

HTML Developer's Guide for ADOBE® AIR®



Legal notices

For legal notices, see http://help.adobe.com/en_US/legalnotices/index.html.

Contents

Chapter 1: About the HTML environment

Overview of the HTML environment	2
AIR and WebKit	5

Chapter 2: Programming HTML and JavaScript in AIR

Creating an HTML-based AIR application	20
An example application and security implications	21
Avoiding security-related JavaScript errors	22
Accessing AIR API classes from JavaScript	27
About URLs in AIR	29
Embedding SWF content in HTML	29
Using ActionScript libraries within an HTML page	32
Converting Date and RegExp objects	34
Cross-scripting content in different security sandboxes	34

Chapter 3: Handling HTML-related events in AIR

HTMLLoader events	39
How AIR class-event handling differs from other event handling in the HTML DOM	40
Adobe AIR event objects	41
Handling runtime events with JavaScript	44

Chapter 4: Scripting the AIR HTML Container

Display properties of HTMLLoader objects	47
Accessing the HTML history list	50
Setting the user agent used when loading HTML content	50
Setting the character encoding to use for HTML content	51
Defining browser-like user interfaces for HTML content	51

Chapter 5: Working with vectors

Basics of vectors	64
Creating vectors	65
Inserting elements into a vector	66
Retrieving values and removing vector elements	66
Properties and methods of Vector objects	67
Example: Using AIR APIs that require vectors	67

Chapter 6: AIR security

AIR security basics	69
Installation and updates	69
HTML security in Adobe AIR	73
Scripting between content in different domains	79
Writing to disk	80
Working securely with untrusted content	81

Contents

Best security practices for developers	82
Code signing	83
Chapter 7: Working with AIR native windows	
Basics of native windows in AIR	84
Creating windows	90
Managing windows	97
Listening for window events	105
Displaying full-screen windows	106
Chapter 8: Display screens in AIR	
Basics of display screens in AIR	109
Enumerating the screens	110
Chapter 9: Working with menus	
Menu basics	113
Creating native menus (AIR)	118
About context menus in HTML (AIR)	120
Displaying pop-up native menus (AIR)	121
Handling menu events	122
Native menu example: Window and application menu (AIR)	123
Using the MenuBuilder framework	126
Chapter 10: Taskbar icons in AIR	
About taskbar icons	140
Dock icons	141
System Tray icons	141
Window taskbar icons and buttons	143
Chapter 11: Working with the file system	
Using the AIR file system API	145
Chapter 12: Drag and drop in AIR	
Drag and drop in HTML	177
Dragging data out of an HTML element	180
Dragging data into an HTML element	181
Example: Overriding the default HTML drag-in behavior	182
Handling file drops in non-application HTML sandboxes	184
Dropping file promises	185
Chapter 13: Copy and paste	
Basics of copy-and-paste	195
Reading from and writing to the system clipboard	196
HTML copy and paste in AIR	196
Clipboard data formats	198
Chapter 14: Working with local SQL databases in AIR	
About local SQL databases	204
Creating and modifying a database	208

Contents

Manipulating SQL database data	212
Using synchronous and asynchronous database operations	232
Using encryption with SQL databases	237
Strategies for working with SQL databases	253
 Chapter 15: Encrypted local storage	
Adding data to the encrypted local store	257
Accessing data in the encrypted local store	258
Removing data from the encrypted local store	258
 Chapter 16: Working with byte arrays	
Reading and writing a ByteArray	259
ByteArray example: Reading a .zip file	265
 Chapter 17: Adding PDF content in AIR	
Detecting PDF Capability	270
Loading PDF content	271
Scripting PDF content	271
Known limitations for PDF content in AIR	273
 Chapter 18: Working with sound	
Basics of working with sound	275
Understanding the sound architecture	276
Loading external sound files	276
Working with embedded sounds	279
Working with streaming sound files	279
Working with dynamically generated audio	280
Playing sounds	281
Working with sound metadata	285
Accessing raw sound data	286
Capturing sound input	289
 Chapter 19: Client system environment	
Basics of the client system environment	293
Using the System class	294
Using the Capabilities class	294
 Chapter 20: AIR application invocation and termination	
Application invocation	296
Capturing command line arguments	297
Invoking an AIR application on user login	299
Invoking an AIR application from the browser	300
Application termination	302
 Chapter 21: Working with AIR runtime and operating system information	
Managing file associations	304
Getting the runtime version and patch level	305
Detecting AIR capabilities	305
Tracking user presence	306

Chapter 22: Sockets

TCP sockets	307
UDP sockets (AIR)	312
IPv6 addresses	314

Chapter 23: HTTP communications

Loading external data	316
Web service requests	323
Opening a URL in another application	328
Sending a URL to a server	330

Chapter 24: Communicating with other Flash Player and AIR instances

About the LocalConnection class	331
Sending messages between two applications	332
Connecting to content in different domains and to AIR applications	332

Chapter 25: ActionScript basics for JavaScript developers

Differences between ActionScript and JavaScript: an overview	335
ActionScript 3.0 data types	336
ActionScript 3.0 classes, packages, and namespaces	337
Required parameters and default values in ActionScript 3.0 functions	339
ActionScript 3.0 event listeners	339

Chapter 26: SQL support in local databases

Supported SQL syntax	342
Data type support	362

Chapter 27: SQL error detail messages, ids, and arguments

Chapter 1: About the HTML environment

Adobe AIR 1.0 and later

AIR uses [WebKit](http://www.webkit.org) (www.webkit.org), also used by the Safari web browser, to parse, layout, and render HTML and JavaScript content. The built-in host classes and objects of AIR provide an API for features traditionally associated with desktop applications. Such features include reading and writing files and managing windows. Adobe AIR also inherits APIs from the Adobe® Flash® Player, which include features like sound and binary sockets.

Important: New versions of the Adobe AIR runtime may include updated versions of WebKit. A WebKit update in a new version of AIR *may* result in unexpected changes in a deployed AIR application. These changes may affect the behavior or appearance of HTML content in an application. For example, improvements or corrections in WebKit rendering may change the layout of elements in an application’s user interface. For this reason, it is highly recommended that you provide an update mechanism in your application. Should you need to update your application due to a change in the WebKit version included in AIR, the AIR update mechanism can prompt the user to install the new version of your application.

The following table lists the version of WebKit used in each release of AIR. The closest corresponding release of the Safari web browser is also given:

AIR version	WebKit version	Safari version
1.0	420	2.04
1.1	523	3.04
1.5	526.9	4.0 Beta
2.0	531.9	4.03
2.5	531.9	4.03
2.6	531.9	4.03

You can always determine the installed version of WebKit by examining the default user agent string returned by a HTMLLoader object:

```
air.trace( window.htmlLoader.userAgent );
```

Keep in mind that the version of WebKit used in AIR is not identical to the open source version. Some features are not supported in AIR and the AIR version can include security and bug fixes not yet available in the corresponding WebKit version. See “[WebKit features not supported in AIR](#)” on page 16.

Using the AIR APIs in HTML content is entirely optional. You can program an AIR application entirely with HTML and JavaScript. Most existing HTML applications should run with few changes (assuming they use HTML, CSS, DOM, and JavaScript features compatible with WebKit).

AIR gives you complete control over the look-and-feel of your application. You can make your application look like a native desktop application. You can turn off the window chrome provided by the operating system and implement your own controls for moving, resizing, and closing windows. You can even run without a window.

Because AIR applications run directly on the desktop, with full access to the file system, the security model is more stringent than the security model of the typical web browser. In AIR, only content loaded from the application installation directory is placed in the *application sandbox*. The application sandbox has the highest level of privilege and allows access to the AIR APIs. AIR places other content into isolated sandboxes based on where that content came from. Files loaded from the file system go into a local sandbox. Files loaded from the network using the http: or https: protocols go into a sandbox based on the domain of the remote server. Content in these non-application sandboxes is prohibited from accessing any AIR API and runs much as it would in a typical web browser.

HTML content in AIR does not display SWF or PDF content if alpha, scaling, or transparency settings are applied. For more information, see “[Considerations when loading SWF or PDF content in an HTML page](#)” on page 48 and “[Window transparency](#)” on page 87.

More Help topics

[Webkit DOM Reference](#)

[Safari HTML Reference](#)

[Safari CSS Reference](#)

www.webkit.org

Overview of the HTML environment

Adobe AIR 1.0 and later

Adobe AIR provides a complete browser-like JavaScript environment with an HTML renderer, document object model, and JavaScript interpreter. The JavaScript environment is represented by the AIR HTMLLoader class. In HTML windows, an HTMLLoader object contains all HTML content, and is, in turn, contained within a NativeWindow object. The NativeWindow object allows an application to script the properties and behavior of native operating system window displayed on the user's desktop.

About the JavaScript environment and its relationship to the AIR host

Adobe AIR 1.0 and later

The following diagram illustrates the relationship between the JavaScript environment and the AIR run-time environment. Although only a single native window is shown, an AIR application can contain multiple windows. (And a single window can contain multiple HTMLLoader objects.)

Important: Only pages installed as part of an application have the `htmlLoader`, `nativeWindow`, or `runtime` properties and only when loaded as the top-level document. These properties are not added when a document is loaded into a frame or iframe. (A child document can access these properties on the parent document as long as it is in the same security sandbox. For example, a document loaded in a frame could access the `runtime` property of its parent with `parent.runtime`.)

About security

Adobe AIR 1.0 and later

AIR executes all code within a security sandbox based on the domain of origin. Application content, which is limited to content loaded from the application installation directory, is placed into the *application* sandbox. Access to the runtime environment and the AIR APIs are only available to HTML and JavaScript running within this sandbox. At the same time, most dynamic evaluation and execution of JavaScript is blocked in the application sandbox after all handlers for the `pageLoad` event have returned.

You can map an application page into a non-application sandbox by loading the page into a frame or iframe and setting the AIR-specific `sandboxRoot` and `documentRoot` attributes of the frame. By setting the `sandboxRoot` value to an actual remote domain, you can enable the sandboxed content to cross-script content in that domain. Mapping pages in this way can be useful when loading and scripting remote content, such as in a *mash-up* application.

Another way to allow application and non-application content to cross-script each other, and the only way to give non-application content access to AIR APIs, is to create a *sandbox bridge*. A *parent-to-child* bridge allows content in a child frame, iframe, or window to access designated methods and properties defined in the application sandbox. Conversely, a *child-to-parent* bridge allows application content to access designated methods and properties defined in the sandbox of the child. Sandbox bridges are established by setting the `parentSandboxBridge` and `childSandboxBridge` properties of the window object. For more information, see [“HTML security in Adobe AIR”](#) on page 73 and [“HTML frame and iframe elements”](#) on page 12.

About plug-ins and embedded objects

Adobe AIR 1.0 and later

AIR supports the Adobe® Acrobat® plug-in. Users must have Acrobat or Adobe® Reader® 8.1 (or better) to display PDF content. The `HTMLLoader` object provides a property for checking whether a user's system can display PDF. SWF file content can also be displayed within the HTML environment, but this capability is built in to AIR and does not use an external plug-in.

No other WebKit plug-ins are supported in AIR.

More Help topics

[“HTML security in Adobe AIR”](#) on page 73

[“HTML Sandboxes”](#) on page 5

[“HTML frame and iframe elements”](#) on page 12

[“JavaScript Window object”](#) on page 10

[“The XMLHttpRequest object”](#) on page 6

[“Adding PDF content in AIR”](#) on page 270

AIR and WebKit

Adobe AIR 1.0 and later

Adobe AIR uses the open source WebKit engine, also used in the Safari web browser. AIR adds several extensions to allow access to the runtime classes and objects as well as for security. In addition, WebKit itself adds features not included in the W3C standards for HTML, CSS, and JavaScript.

Only the AIR additions and the most noteworthy WebKit extensions are covered here; for additional documentation on non-standard HTML, CSS, and JavaScript, see www.webkit.org and developer.apple.com. For standards information, see the [W3C web site](#). Mozilla also provides a valuable [general reference](#) on HTML, CSS, and DOM topics (of course, the WebKit and Mozilla engines are not identical).

JavaScript in AIR

Flash Player 9 and later, Adobe AIR 1.0 and later

AIR makes several changes to the typical behavior of common JavaScript objects. Many of these changes are made to make it easier to write secure applications in AIR. At the same time, these differences in behavior mean that some common JavaScript coding patterns, and existing web applications using those patterns, might not always execute as expected in AIR. For information on correcting these types of issues, see “[Avoiding security-related JavaScript errors](#)” on page 22.

HTML Sandboxes

Adobe AIR 1.0 and later

AIR places content into isolated sandboxes according to the origin of the content. The sandbox rules are consistent with the same-origin policy implemented by most web browsers, as well as the rules for sandboxes implemented by the Adobe Flash Player. In addition, AIR provides a new *application* sandbox type to contain and protect application content. See Security sandboxes for more information on the types of sandboxes you may encounter when developing AIR applications.

Access to the run-time environment and AIR APIs are only available to HTML and JavaScript running within the application sandbox. At the same time, however, dynamic evaluation and execution of JavaScript, in its various forms, is largely restricted within the application sandbox for security reasons. These restrictions are in place whether or not your application actually loads information directly from a server. (Even file content, pasted strings, and direct user input may be untrustworthy.)

The origin of the content in a page determines the sandbox to which it is consigned. Only content loaded from the application directory (the installation directory referenced by the `app:` URL scheme) is placed in the application sandbox. Content loaded from the file system is placed in the *local-with-file system* or the *local-trusted* sandbox, which allows access and interaction with content on the local file system, but not remote content. Content loaded from the network is placed in a remote sandbox corresponding to its domain of origin.

To allow an application page to interact freely with content in a remote sandbox, the page can be mapped to the same domain as the remote content. For example, if you write an application that displays map data from an Internet service, the page of your application that loads and displays content from the service could be mapped to the service domain. The attributes for mapping pages into a remote sandbox and domain are new attributes added to the frame and iframe HTML elements.

To allow content in a non-application sandbox to safely use AIR features, you can set up a parent sandbox bridge. To allow application content to safely call methods and access properties of content in other sandboxes, you can set up a child sandbox bridge. Safety here means that remote content cannot accidentally get references to objects, properties, or methods that are not explicitly exposed. Only simple data types, functions, and anonymous objects can be passed across the bridge. However, you must still avoid explicitly exposing potentially dangerous functions. If, for example, you exposed an interface that allowed remote content to read and write files anywhere on a user's system, then you might be giving remote content the means to do considerable harm to your users.

JavaScript eval() function

Adobe AIR 1.0 and later

Use of the `eval()` function is restricted within the application sandbox once a page has finished loading. Some uses are permitted so that JSON-formatted data can be safely parsed, but any evaluation that results in executable statements results in an error. "[Code restrictions for content in different sandboxes](#)" on page 76 describes the allowed uses of the `eval()` function.

Function constructors

Adobe AIR 1.0 and later

In the application sandbox, function constructors can be used before a page has finished loading. After all `page load` event handlers have finished, new functions cannot be created.

Loading external scripts

Adobe AIR 1.0 and later

HTML pages in the application sandbox cannot use the `script` tag to load JavaScript files from outside of the application directory. For a page in your application to load a script from outside the application directory, the page must be mapped to a non-application sandbox.

The XMLHttpRequest object

Adobe AIR 1.0 and later

AIR provides an XMLHttpRequest (XHR) object that applications can use to make data requests. The following example illustrates a simple data request:

```
xmlhttp = new XMLHttpRequest();
xmlhttp.open("GET", "http://www.example.com/file.data", true);
xmlhttp.onreadystatechange = function() {
    if (xmlhttp.readyState == 4) {
        //do something with data...
    }
}
xmlhttp.send(null);
```

In contrast to a browser, AIR allows content running in the application sandbox to request data from any domain. The result of an XHR that contains a JSON string can be evaluated into data objects unless the result also contains executable code. If executable statements are present in the XHR result, an error is thrown and the evaluation attempt fails.

To prevent accidental injection of code from remote sources, synchronous XHRs return an empty result if made before a page has finished loading. Asynchronous XHRs will always return after a page has loaded.

By default, AIR blocks cross-domain XMLHttpRequests in non-application sandboxes. A parent window in the application sandbox can choose to allow cross-domain requests in a child frame containing content in a non-application sandbox by setting `allowCrossDomainXHR`, an attribute added by AIR, to `true` in the containing frame or `iframe` element:

```
<iframe id="mashup"
  src="http://www.example.com/map.html"
  allowCrossDomainXHR="true"
</iframe>
```

Note: When convenient, the AIR `URLStream` class can also be used to download data.

If you dispatch an XMLHttpRequest to a remote server from a frame or `iframe` containing application content that has been mapped to a remote sandbox, make sure that the mapping URL does not mask the server address used in the XHR. For example, consider the following `iframe` definition, which maps application content into a remote sandbox for the `example.com` domain:

```
<iframe id="mashup"
  src="http://www.example.com/map.html"
  documentRoot="app:/sandbox/"
  sandboxRoot="http://www.example.com/"
  allowCrossDomainXHR="true"
</iframe>
```

Because the `sandboxRoot` attribute remaps the root URL of the `www.example.com` address, all requests are loaded from the application directory and not the remote server. Requests are remapped whether they derive from page navigation or from an XMLHttpRequest.

To avoid accidentally blocking data requests to your remote server, map the `sandboxRoot` to a subdirectory of the remote URL rather than the root. The directory does not have to exist. For example, to allow requests to the `www.example.com` to load from the remote server rather than the application directory, change the previous `iframe` to the following:

```
<iframe id="mashup"
  src="http://www.example.com/map.html"
  documentRoot="app:/sandbox/"
  sandboxRoot="http://www.example.com/air/"
  allowCrossDomainXHR="true"
</iframe>
```

In this case, only content in the `air` subdirectory is loaded locally.

For more information on sandbox mapping see “[HTML frame and iframe elements](#)” on page 12 and “[HTML security in Adobe AIR](#)” on page 73.

Cookies

Adobe AIR 1.0 and later

In AIR applications, only content in remote sandboxes (content loaded from `http:` and `https:` sources) can use cookies (the `document.cookie` property). In the application sandbox, other means for storing persistent data are available, including the `EncryptedLocalStore`, `SharedObject`, and `FileStream` classes.

The Clipboard object

Adobe AIR 1.0 and later

The WebKit Clipboard API is driven with the following events: `copy`, `cut`, and `paste`. The event object passed in these events provides access to the clipboard through the `clipboardData` property. Use the following methods of the `clipboardData` object to read or write clipboard data:

Method	Description
<code>clearData(mimeType)</code>	Clears the clipboard data. Set the <code>mimeType</code> parameter to the MIME type of the data to clear.
<code>getData(mimeType)</code>	Get the clipboard data. This method can only be called in a handler for the <code>paste</code> event. Set the <code>mimeType</code> parameter to the MIME type of the data to return.
<code>setData(mimeType, data)</code>	Copy data to the clipboard. Set the <code>mimeType</code> parameter to the MIME type of the data.

JavaScript code outside the application sandbox can only access the clipboard through these events. However, content in the application sandbox can access the system clipboard directly using the AIR Clipboard class. For example, you could use the following statement to get text format data on the clipboard:

```
var clipping = air.Clipboard.generalClipboard.getData("text/plain",  
    air.ClipboardTransferMode.ORIGINAL_ONLY);
```

The valid data MIME types are:

MIME type	Value
Text	"text/plain"
HTML	"text/html"
URL	"text/uri-list"
Bitmap	"image/x-vnd.adobe.air.bitmap"
File list	"application/x-vnd.adobe.air.file-list"

Important: Only content in the application sandbox can access file data present on the clipboard. If non-application content attempts to access a file object from the clipboard, a security error is thrown.

For more information on using the clipboard, see [“Copy and paste”](#) on page 195 and [Using the Pasteboard from JavaScript \(Apple Developer Center\)](#).

Drag and Drop

Adobe AIR 1.0 and later

Drag-and-drop gestures into and out of HTML produce the following DOM events: `dragstart`, `drag`, `dragend`, `dragenter`, `dragover`, `dragleave`, and `drop`. The event object passed in these events provides access to the dragged data through the `dataTransfer` property. The `dataTransfer` property references an object that provides the same methods as the `clipboardData` object associated with a clipboard event. For example, you could use the following function to get text format data from a `drop` event:

```
function onDrop(dragEvent) {  
    return dragEvent.dataTransfer.getData("text/plain",  
        air.ClipboardTransferMode.ORIGINAL_ONLY);  
}
```

The `dataTransfer` object has the following important members:

Member	Description
<code>clearData(mimeType)</code>	Clears the data. Set the <code>mimeType</code> parameter to the MIME type of the data representation to clear.
<code>getData(mimeType)</code>	Get the dragged data. This method can only be called in a handler for the <code>drop</code> event. Set the <code>mimeType</code> parameter to the MIME type of the data to get.
<code>setData(mimeType, data)</code>	Set the data to be dragged. Set the <code>mimeType</code> parameter to the MIME type of the data.
<code>types</code>	An array of strings containing the MIME types of all data representations currently available in the <code>dataTransfer</code> object.
<code>effectsAllowed</code>	Specifies whether the data being dragged can be copied, moved, linked, or some combination thereof. Set the <code>effectsAllowed</code> property in the handler for the <code>dragstart</code> event.
<code>dropEffect</code>	Specifies which of the allowed drop effects are supported by a drag target. Set the <code>dropEffect</code> property in the handler for the <code>dragEnter</code> event. During the drag, the cursor changes to indicate which effect would occur if the user released the mouse. If no <code>dropEffect</code> is specified, an <code>effectsAllowed</code> property effect is chosen. The copy effect has priority over the move effect, which itself has priority over the link effect. The user can modify the default priority using the keyboard.

For more information on adding support for drag-and-drop to an AIR application see “[Drag and drop in AIR](#)” on page 177 and [Using the Drag-and-Drop from JavaScript \(Apple Developer Center\)](#).

innerHTML and outerHTML properties

Adobe AIR 1.0 and later

AIR places security restrictions on the use of the `innerHTML` and `outerHTML` properties for content running in the application sandbox. Before the page load event, as well as during the execution of any load event handlers, use of the `innerHTML` and `outerHTML` properties is unrestricted. However, once the page has loaded, you can only use `innerHTML` or `outerHTML` properties to add static content to the document. Any statement in the string assigned to `innerHTML` or `outerHTML` that evaluates to executable code is ignored. For example, if you include an event callback attribute in an element definition, the event listener is not added. Likewise, embedded `<script>` tags are not evaluated. For more information, see the “[HTML security in Adobe AIR](#)” on page 73.

Document.write() and Document.writeln() methods

Adobe AIR 1.0 and later

Use of the `write()` and `writeln()` methods is not restricted in the application sandbox before the `load` event of the page. However, once the page has loaded, calling either of these methods does not clear the page or create a new one. In a non-application sandbox, as in most web browsers, calling `document.write()` or `writeln()` after a page has finished loading clears the current page and opens a new, blank one.

Document.designMode property

Adobe AIR 1.0 and later

Set the `document.designMode` property to a value of `on` to make all elements in the document editable. Built-in editor support includes text editing, copy, paste, and drag-and-drop. Setting `designMode` to `on` is equivalent to setting the `contentEditable` property of the `body` element to `true`. You can use the `contentEditable` property on most HTML elements to define which sections of a document are editable. See “[HTML contentEditable attribute](#)” on page 14 for additional information.

unload events (for body and frameset objects)

Adobe AIR 1.0 and later

In the top-level `frameset` or `body` tag of a window (including the main window of the application), do not use the `unload` event to respond to the window (or application) being closed. Instead, use `exiting` event of the `NativeApplication` object (to detect when an application is closing). Or use the `closing` event of the `NativeWindow` object (to detect when a window is closing). For example, the following JavaScript code displays a message ("Goodbye. ") when the user closes the application:

```
var app = air.NativeApplication.nativeApplication;
app.addEventListener(air.Event.EXITING, closeHandler);
function closeHandler(event)
{
    alert("Goodbye. ");
}
```

However, scripts *can* successfully respond to the `unload` event caused by navigation of a frame, `iframe`, or top-level window content.

Note: These limitations may be removed in a future version of Adobe AIR.

JavaScript Window object

Adobe AIR 1.0 and later

The `Window` object remains the global object in the JavaScript execution context. In the application sandbox, AIR adds new properties to the JavaScript `Window` object to provide access to the built-in classes of AIR, as well as important host objects. In addition, some methods and properties behave differently depending on whether they are within the application sandbox or not.

Window.runtime property The `runtime` property allows you to instantiate and use the built-in runtime classes from within the application sandbox. These classes include the AIR and Flash Player APIs (but not, for example, the Flex framework). For example, the following statement creates an AIR file object:

```
var preferencesFile = new window.runtime.flash.filesystem.File();
```

The `AIRAliases.js` file, provided in the AIR SDK, contains alias definitions that allow you to shorten such references. For example, when `AIRAliases.js` is imported into a page, a `File` object can be created with the following statement:

```
var preferencesFile = new air.File();
```

The `window.runtime` property is only defined for content within the application sandbox and only for the parent document of a page with frames or iframes.

See “[Using the AIRAliases.js file](#)” on page 28.

Window.nativeWindow property The `nativeWindow` property provides a reference to the underlying native window object. With this property, you can script window functions and properties such as screen position, size, and visibility, and handle window events such as closing, resizing, and moving. For example, the following statement closes the window:

```
window.nativeWindow.close();
```

Note: The window control features provided by the `NativeWindow` object overlap the features provided by the JavaScript `Window` object. In such cases, you can use whichever method you find most convenient.

The `window.nativeWindow` property is only defined for content within the application sandbox and only for the parent document of a page with frames or iframes.

Window.htmlLoader property The `htmlLoader` property provides a reference to the AIR HTMLLoader object that contains the HTML content. With this property, you can script the appearance and behavior of the HTML environment. For example, you can use the `htmlLoader.paintsDefaultBackground` property to determine whether the control paints a default, white background:

```
window.htmlLoader.paintsDefaultBackground = false;
```

Note: The HTMLLoader object itself has a `window` property, which references the JavaScript Window object of the HTML content it contains. You can use this property to access the JavaScript environment through a reference to the containing HTMLLoader.

The `window.htmlLoader` property is only defined for content within the application sandbox and only for the parent document of a page with frames or iframes.

Window.parentSandboxBridge and Window.childSandboxBridge properties The `parentSandboxBridge` and `childSandboxBridge` properties allow you to define an interface between a parent and a child frame. For more information, see [“Cross-scripting content in different security sandboxes”](#) on page 34.

Window.setTimeout() and Window.setInterval() functions AIR places security restrictions on use of the `setTimeout()` and `setInterval()` functions within the application sandbox. You cannot define the code to be executed as a string when calling `setTimeout()` or `setInterval()`. You must use a function reference. For more information, see [“setTimeout\(\) and setInterval\(\)”](#) on page 25.

Window.open() function When called by code running in a non-application sandbox, the `open()` method only opens a window when called as a result of user interaction (such as a mouse click or keypress). In addition, the window title is prefixed with the application title (to prevent windows opened by remote content from impersonating windows opened by the application). For more information, see the [“Restrictions on calling the JavaScript window.open\(\) method”](#) on page 78.

air.NativeApplication object

Adobe AIR 1.0 and later

The `NativeApplication` object provides information about the application state, dispatches several important application-level events, and provides useful functions for controlling application behavior. A single instance of the `NativeApplication` object is created automatically and can be accessed through the class-defined `NativeApplication.nativeApplication` property.

To access the object from JavaScript code you could use:

```
var app = window.runtime.flash.desktop.NativeApplication.nativeApplication;
```

Or, if the `AIRAliases.js` script has been imported, you could use the shorter form:

```
var app = air.NativeApplication.nativeApplication;
```

The `NativeApplication` object can only be accessed from within the application sandbox. For more information about the `NativeApplication` object, see [“Working with AIR runtime and operating system information”](#) on page 304.

The JavaScript URL scheme

Adobe AIR 1.0 and later

Execution of code defined in a JavaScript URL scheme (as in `href="javascript:alert('Test')"`) is blocked within the application sandbox. No error is thrown.

HTML in AIR

Adobe AIR 1.0 and later

AIR and WebKit define a couple of non-standard HTML elements and attributes, including:

[“HTML frame and iframe elements”](#) on page 12

[“HTML element event handlers”](#) on page 14

HTML frame and iframe elements

Adobe AIR 1.0 and later

AIR adds new attributes to the frame and iframe elements of content in the application sandbox:

sandboxRoot attribute The `sandboxRoot` attribute specifies an alternate, non-application domain of origin for the file specified by the frame `src` attribute. The file is loaded into the non-application sandbox corresponding to the specified domain. Content in the file and content loaded from the specified domain can cross-script each other.

***Important:** If you set the value of `sandboxRoot` to the base URL of the domain, all requests for content from that domain are loaded from the application directory instead of the remote server (whether that request results from page navigation, from an XMLHttpRequest, or from any other means of loading content).*

documentRoot attribute The `documentRoot` attribute specifies the local directory from which to load URLs that resolve to files within the location specified by `sandboxRoot`.

When resolving URLs, either in the frame `src` attribute, or in content loaded into the frame, the part of the URL matching the value specified in `sandboxRoot` is replaced with the value specified in `documentRoot`. Thus, in the following frame tag:

```
<iframe src="http://www.example.com/air/child.html"
        documentRoot="app:/sandbox/"
        sandboxRoot="http://www.example.com/air/" />
```

`child.html` is loaded from the `sandbox` subdirectory of the application installation folder. Relative URLs in `child.html` are resolved based on `sandbox` directory. Note that any files on the remote server at `www.example.com/air` are not accessible in the frame, since AIR would attempt to load them from the `app:/sandbox/` directory.

allowCrossDomainXHR attribute Include `allowCrossDomainXHR="allowCrossDomainXHR"` in the opening frame tag to allow content in the frame to make XMLHttpRequests to any remote domain. By default, non-application content can only make such requests to its own domain of origin. There are serious security implications involved in allowing cross-domain XHRs. Code in the page is able to exchange data with any domain. If malicious content is somehow injected into the page, any data accessible to code in the current sandbox can be compromised. Only enable cross-domain XHRs for pages that you create and control and only when cross-domain data loading is truly necessary. Also, carefully validate all external data loaded by the page to prevent code injection or other forms of attack.

***Important:** If the `allowCrossDomainXHR` attribute is included in a frame or iframe element, cross-domain XHRs are enabled (unless the value assigned is "0" or starts with the letters "f" or "n"). For example, setting `allowCrossDomainXHR` to "deny" would still enable cross-domain XHRs. Leave the attribute out of the element declaration altogether if you do not want to enable cross-domain requests.*

ondominitialize attribute Specifies an event handler for the `dominitialize` event of a frame. This event is an AIR-specific event that fires when the window and document objects of the frame have been created, but before any scripts have been parsed or document elements created.

The frame dispatches the `dominitialize` event early enough in the loading sequence that any script in the child page can reference objects, variables, and functions added to the child document by the `dominitialize` handler. The parent page must be in the same sandbox as the child to directly add or access any objects in a child document. However, a parent in the application sandbox can establish a sandbox bridge to communicate with content in a non-application sandbox.

The following examples illustrate use of the `iframe` tag in AIR:

Place `child.html` in a remote sandbox, without mapping to an actual domain on a remote server:

```
<iframe src="http://localhost/air/child.html"
        documentRoot="app:/sandbox/"
        sandboxRoot="http://localhost/air/" />
```

Place `child.html` in a remote sandbox, allowing XMLHttpRequests only to `www.example.com`:

```
<iframe src="http://www.example.com/air/child.html"
        documentRoot="app:/sandbox/"
        sandboxRoot="http://www.example.com/air/" />
```

Place `child.html` in a remote sandbox, allowing XMLHttpRequests to any remote domain:

```
<iframe src="http://www.example.com/air/child.html"
        documentRoot="app:/sandbox/"
        sandboxRoot="http://www.example.com/air/"
        allowCrossDomainXHR="allowCrossDomainXHR" />
```

Place `child.html` in a local-with-file-system sandbox:

```
<iframe src="file:///templates/child.html"
        documentRoot="app:/sandbox/"
        sandboxRoot="app-storage:/templates/" />
```

Place `child.html` in a remote sandbox, using the `dominitialize` event to establish a sandbox bridge:

```
<html>
<head>
<script>
var bridgeInterface = {};
bridgeInterface.testProperty = "Bridge engaged";
function engageBridge() {
    document.getElementById("sandbox").parentSandboxBridge = bridgeInterface;
}
</script>
</head>
<body>
<iframe id="sandbox"
        src="http://www.example.com/air/child.html"
        documentRoot="app:/"
        sandboxRoot="http://www.example.com/air/"
        ondominitialize="engageBridge()" />
</body>
</html>
```

The following `child.html` document illustrates how child content can access the parent sandbox bridge:

```
<html>
  <head>
    <script>
      document.write(window.parentSandboxBridge.testProperty);
    </script>
  </head>
  <body></body>
</html>
```

For more information, see “[Cross-scripting content in different security sandboxes](#)” on page 34 and “[HTML security in Adobe AIR](#)” on page 73.

HTML element event handlers

Adobe AIR 1.0 and later

DOM objects in AIR and WebKit dispatch some events not found in the standard DOM event model. The following table lists the related event attributes you can use to specify handlers for these events:

Callback attribute name	Description
oncontextmenu	Called when a context menu is invoked, such as through a right-click or command-click on selected text.
oncopy	Called when a selection in an element is copied.
oncut	Called when a selection in an element is cut.
onDOMInitialize	Called when the DOM of a document loaded in a frame or iframe is created, but before any DOM elements are created or scripts parsed.
ondrag	Called when an element is dragged.
ondragend	Called when a drag is released.
ondragenter	Called when a drag gesture enters the bounds of an element.
ondragleave	Called when a drag gesture leaves the bounds of an element.
ondragover	Called continuously while a drag gesture is within the bounds of an element.
ondragstart	Called when a drag gesture begins.
ondrop	Called when a drag gesture is released while over an element.
onerror	Called when an error occurs while loading an element.
oninput	Called when text is entered into a form element.
onpaste	Called when an item is pasted into an element.
onscroll	Called when the content of a scrollable element is scrolled.
onselectstart	Called when a selection begins.

HTML contentEditable attribute

Adobe AIR 1.0 and later

You can add the `contentEditable` attribute to any HTML element to allow users to edit the content of the element. For example, the following example HTML code sets the entire document as editable, except for first `p` element:

```
<html>
<head/>
<body contentEditable="true">
  <h1>de Finibus Bonorum et Malorum</h1>
  <p contentEditable="false">Sed ut perspiciatis unde omnis iste natus error.</p>
  <p>At vero eos et accusamus et iusto odio dignissimos ducimus qui blanditiis.</p>
</body>
</html>
```

Note: If you set the `document.designMode` property to `on`, then all elements in the document are editable, regardless of the setting of `contentEditable` for an individual element. However, setting `designMode` to `off`, does not disable editing of elements for which `contentEditable` is `true`. See “[Document.designMode property](#)” on page 9 for additional information.

Data: URLs

Adobe AIR 2 and later

AIR supports data: URLs for the following elements:

- `img`
- `input type="image"`
- CSS rules allowing images (such as `background-image`)

Data URLs allow you to insert binary image data directly into a CSS or HTML document as a base64-encoded string. The following example uses a data: URL as a repeating background:

```
<html>
<head>
<style>
body {
background-
image:url ('data:image/png;base64,iVBORw0KGgoAAAANSUHEUgAAAGQAAABkCMAAAABHPGVmAAAAGXRFWHR Tb2Z
0d2FyZQB BZG9iZSBjbWFnZVVJlYW R5ccllPAAAAZQTFRF%2F6cA%2F%2F%2F%2Fgxp3lwAAAAJ0Uk5T%2FwDltzBKAAA
BF0lEQVR42uzZQQ7CMAxE0e%2F7X5oNCyRocWzPiJbMBZ6qpIljE%2BnwklgKG7kwUjc2IkIaxkY0CPdEsCCasws6ShX
BgmBBmEagpXQQLAgWBAuSY2gaKaWPYEGwIEwg0FRmECwIFoQeQjJlhJWUEFazjFDJckI5WYRWMgjt fEGYyQnCXD4jTCd
m1zmngFpBFznwVNi5RPSbwbWnpYr%2BBHi%2FtCTfgPLEPL7jBctAKBRptXJ8M%2BprIuZKu%2BUCg4YK1PLz7kx4bS
qHyPaT4d%2B28OCJjRBo4FCQsSA0bziT3XubMgYUG6fc5fatmGBQkL0hoJ1IaZMiQsSFiQ8vRscTj lQOI2iHZwtpHu f
%2BJAYiOiJSkj8Z%2FIQ4ABANvXGLd3%2BZMrAAAAAE1FTkSuQmCC');
background-repeat: repeat;
}
</style>
</head>
<body>
</body>
</html>
```

When using data: URLs, be aware that extra whitespace is significant. For example, the data string must be entered as a single, unbroken line. Otherwise, the line breaks are treated as part of the data and the image cannot be decoded.

CSS in AIR

Adobe AIR 1.0 and later

WebKit supports several extended CSS properties. Many of these extensions use the prefix: `-webkit`. Note that some of these extensions are experimental in nature and may be removed from a future version of WebKit. For more information about the Webkit support for CSS and its extensions to CSS, see [Safari CSS Reference](#).

WebKit features not supported in AIR

Adobe AIR 1.0 and later

AIR does not support the following features available in WebKit or Safari 4:

- Cross-domain messaging via `window.postMessage` (AIR provides its own cross-domain communication APIs)
- CSS variables
- Web Open Font Format (WOFF) and SVG fonts.
- HTML video and audio tags
- Media device queries
- Offline application cache
- Printing (AIR provides its own `PrintJob` API)
- Spelling and grammar checkers
- SVG
- WAI-ARIA
- WebSockets (AIR provides its own socket APIs)
- Web workers
- WebKit SQL API (AIR provides its own API)
- WebKit geolocation API (AIR provides its own geolocation API on supported devices)
- WebKit multi-file upload API
- WebKit touch events (AIR provides its own touch events)
- Wireless Markup Language (WML)

The following lists contain specific JavaScript APIs, HTML elements, and CSS properties and values that AIR does not support:

Unsupported JavaScript Window object members:

- `applicationCache()`
- `console`
- `openDatabase()`
- `postMessage()`
- `document.print()`

Unsupported HTML tags:

- `audio`

- video

Unsupported HTML attributes:

- aria-*
- draggable
- formnovalidate
- list
- novalidate
- onbeforeload
- onhashchange
- onorientationchange
- onpagehide
- onpageshow
- onpopstate
- ontouchstart
- ontouchmove
- ontouchend
- ontouchcancel
- onwebkitbeginfullscreen
- onwebkitendfullscreen
- pattern
- required
- sandbox

Unsupported JavaScript events:

- beforeload
- hashchange
- orientationchange
- pagehide
- pageshow
- popstate
- touchstart
- touchmove
- touchend
- touchcancel
- webkitbeginfullscreen
- webkitendfullscreen

Unsupported CSS properties:

- background-clip
- background-origin (use -webkit-background-origin)
- background-repeat-x
- background-repeat-y
- background-size (use -webkit-background-size)
- border-bottom-left-radius
- border-bottom-right-radius
- border-radius
- border-top-left-radius
- border-top-right-radius
- text-rendering
- -webkit-animation-play-state
- -webkit-background-clip
- -webkit-color-correction
- -webkit-font-smoothing

Unsupported CSS values:

- appearance property values:
 - media-volume-slider-container
 - media-volume-slider
 - media-volume-sliderthumb
 - outer-spin-button
- border-box (background-clip and background-origin)
- contain (background-size)
- content-box (background-clip and background-origin)
- cover (background-size)
- list property values:
 - afar
 - amharic
 - amharic-abegede
 - cjk-earthly-branch
 - cjk-heavenly-stem
 - ethiopic
 - ethiopic-abegede
 - ethiopic-abegede-am-et
 - ethiopic-abegede-gez
 - ethiopic-abegede-ti-er

- ethiopic-abegede-ti-et
- ethiopic-halehame-aa-er
- ethiopic-halehame-aa-et
- ethiopic-halehame-am-et
- ethiopic-halehame-gez
- ethiopic-halehame-om-et
- ethiopic-halehame-sid-et
- ethiopic-halehame-so-et
- ethiopic-halehame-ti-er
- ethiopic-halehame-ti-et
- ethiopic-halehame-tig
- hangul
- hangul-consonant
- lower-norwegian
- oromo
- sidama
- somali
- tigre
- tigrinya-er
- tigrinya-er-abegede
- tigrinya-et
- tigrinya-et-abegede
- upper-greek
- upper-norwegian
- -wap-marquee (display property)

Chapter 2: Programming HTML and JavaScript in AIR

Adobe AIR 1.0 and later

A number of programming topics are unique to developing Adobe® AIR® applications with HTML and JavaScript. The following information is important whether you are programming an HTML-based AIR application or programming a SWF-based AIR application that runs HTML and JavaScript using the HTMLLoader class (or mx:HTML Flex™ component).

Creating an HTML-based AIR application

Adobe AIR 1.0 and later

The process of developing an AIR application is much the same as that of developing an HTML-based web application. Application structure remains page-based, with HTML providing the document structure and JavaScript providing the application logic. In addition, an AIR application requires an application descriptor file, which contains metadata about the application and identifies the root file of the application.

If you are using Adobe® Dreamweaver®, you can test and package an AIR application directly from the Dreamweaver user interface. If you are using the AIR SDK, you can test an AIR application using the command-line ADL utility. ADL reads the application descriptor and launches the application. You can package the application into an AIR installation file using the command-line ADT utility.

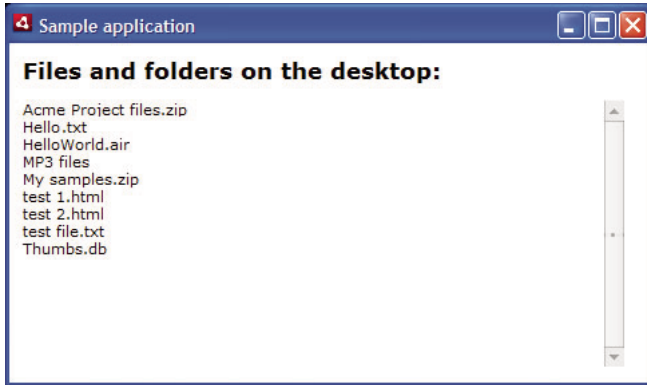
The basic steps to creating an AIR application are:

- 1 Create the application descriptor file. The content element identifies the root page of the application, which is loaded automatically when your application is launched.
- 2 Create the application pages and code.
- 3 Test the application using the ADL utility or Dreamweaver.
- 4 Package the application into an AIR installation file with the ADT utility or Dreamweaver.

An example application and security implications

Adobe AIR 1.0 and later

The following HTML code uses the filesystem APIs to list the files and directories in the user's desktop directory.



Here's the HTML code for the application:

```
<html>
  <head>
    <title>Sample application</title>
    <script type="text/javascript" src="AIRAliases.js"></script>
    <script>
      function getDesktopFileList()
      {
        var log = document.getElementById("log");
        var files = air.File.desktopDirectory.getDirectoryListing();
        for (i = 0; i < files.length; i++)
        {
          log.innerHTML += files[i].name + "<br/>";
        }
      }
    </script>
  </head>
  <body onload="getDesktopFileList();" style="padding: 10px">
    <h2>Files and folders on the desktop:</h2>
    <div id="log" style="width: 450px; height: 200px; overflow-y: scroll;" />
  </body>
</html>
```

You also must set up an application descriptor file and test the application using the AIR Debug Launcher (ADL) application.

You could use most of the sample code in a web browser. However, there are a few lines of code that are specific to the runtime.

The `getDesktopFileList()` method uses the `File` class, which is defined in the runtime APIs. The first `script` tag in the application loads the `AIRAliases.js` file (supplied with the AIR SDK), which lets you easily access the AIR APIs. (For example, the example code accesses the AIR `File` class using the syntax `air.File`.) For details, see [“Using the AIRAliases.js file”](#) on page 28.

The `File.desktopDirectory` property is a `File` object (a type of object defined by the runtime). A `File` object is a reference to a file or directory on the user's computer. The `File.desktopDirectory` property is a reference to the user's desktop directory. The `getDirectoryListing()` method is defined for any `File` object and returns an array of `File` objects. The `File.desktopDirectory.getDirectoryListing()` method returns an array of `File` objects representing files and directories on the user's desktop.

Each `File` object has a `name` property, which is the filename as a string. The `for` loop in the `getDesktopFileList()` method iterates through the files and directories on the user's desktop directory and appends their names to the `innerHTML` property of a `div` object in the application.

Important security rules when using HTML in AIR applications

Adobe AIR 1.0 and later

The files you install with the AIR application have access to the AIR APIs. For security reasons, content from other sources do not. For example, this restriction prevents content from a remote domain (such as `http://example.com`) from reading the contents the user's desktop directory (or worse).

Because there are security loopholes that can be exploited through calling the `eval()` function (and related APIs), content installed with the application, by default, is restricted from using these methods. However, some Ajax frameworks use the calling the `eval()` function and related APIs.

To properly structure content to work in an AIR application, you must take the rules for the security restrictions on content from different sources into account. Content from different sources is placed in separate security classifications, called sandboxes (see Security sandboxes). By default, content installed with the application is installed in a sandbox known as the *application* sandbox, and this grants it access to the AIR APIs. The application sandbox is generally the most secure sandbox, with restrictions designed to prevent the execution of untrusted code.

The runtime allows you to load content installed with your application into a sandbox other than the application sandbox. Content in non-application sandboxes operates in a security environment similar to that of a typical web browser. For example, code in non-application sandboxes can use `eval()` and related methods (but at the same time is not allowed to access the AIR APIs). The runtime includes ways to have content in different sandboxes communicate securely (without exposing AIR APIs to non-application content, for example). For details, see [“Cross-scripting content in different security sandboxes”](#) on page 34.

If you call code that is restricted from use in a sandbox for security reasons, the runtime dispatches a JavaScript error: “Adobe AIR runtime security violation for JavaScript code in the application security sandbox.”

To avoid this error, follow the coding practices described in the next section, [“Avoiding security-related JavaScript errors”](#) on page 22.

For more information, see [“HTML security in Adobe AIR”](#) on page 73.

Avoiding security-related JavaScript errors

Adobe AIR 1.0 and later

If you call code that is restricted from use in a sandbox due to these security restrictions, the runtime dispatches a JavaScript error: “Adobe AIR runtime security violation for JavaScript code in the application security sandbox.” To avoid this error, follow these coding practices.

Causes of security-related JavaScript errors

Adobe AIR 1.0 and later

Code executing in the application sandbox is restricted from most operations that involve evaluating and executing strings once the document `load` event has fired and any `load` event handlers have exited. Attempting to use the following types of JavaScript statements that evaluate and execute potentially insecure strings generates JavaScript errors:

- [eval\(\) function](#)
- [setTimeout\(\) and setInterval\(\)](#)
- [Function constructor](#)

In addition, the following types of JavaScript statements fail without generating an unsafe JavaScript error:

- [javascript: URLs](#)
- [Event callbacks assigned through onevent attributes in innerHTML and outerHTML statements](#)
- [Loading JavaScript files from outside the application installation directory](#)
- [document.write\(\) and document.writeln\(\)](#)
- [Synchronous XMLHttpRequests before the load event or during a load event handler](#)
- [Dynamically created script elements](#)

Note: In some restricted cases, evaluation of strings is permitted. See “[Code restrictions for content in different sandboxes](#)” on page 76 for more information.

Adobe maintains a list of Ajax frameworks known to support the application security sandbox, at <http://www.adobe.com/go/airappsandboxframeworks>.

The following sections describe how to rewrite scripts to avoid these unsafe JavaScript errors and silent failures for code running in the application sandbox.

Mapping application content to a different sandbox

Adobe AIR 1.0 and later

In most cases, you can rewrite or restructure an application to avoid security-related JavaScript errors. However, when rewriting or restructuring is not possible, you can load the application content into a different sandbox using the technique described in “[Loading application content into a non-application sandbox](#)” on page 35. If that content also must access AIR APIs, you can create a sandbox bridge, as described in “[Setting up a sandbox bridge interface](#)” on page 36.

eval() function

Flash Player 9 and later, Adobe AIR 1.0 and later

In the application sandbox, the `eval()` function can only be used before the page `load` event or during a `load` event handler. After the page has loaded, calls to `eval()` will not execute code. However, in the following cases, you can rewrite your code to avoid the use of `eval()`.

Assigning properties to an object

Adobe AIR 1.0 and later

Instead of parsing a string to build the property accessor:

```
eval("obj." + propName + " = " + val);
```

access properties with bracket notation:

```
obj[propName] = val;
```

Creating a function with variables available in context

Adobe AIR 1.0 and later

Replace statements such as the following:

```
function compile(var1, var2){
    eval("var fn = function(){ this."+var1+"(var2) }");
    return fn;
}
```

with:

```
function compile(var1, var2){
    var self = this;
    return function(){ self[var1](var2) };
}
```

Creating an object using the name of the class as a string parameter

Adobe AIR 1.0 and later

Consider a hypothetical JavaScript class defined with the following code:

```
var CustomClass =
{
    Utils:
    {
        Parser: function(){ alert('constructor') }
    },
    Data:
    {
    }
};

var constructorClassName = "CustomClass.Utils.Parser";
```

The simplest way to create an instance would be to use `eval()`:

```
var myObj;
eval('myObj=new ' + constructorClassName + '()')
```

However, you could avoid the call to `eval()` by parsing each component of the class name and building the new object using bracket notation:

```
function getter(str)
{
    var obj = window;
    var names = str.split('.');
    for(var i=0;i<names.length;i++){
        if(typeof obj[names[i]]=='undefined'){
            var undefstring = names[0];
            for(var j=1;j<=i;j++){
                undefstring+="."+names[j];
                throw new Error(undefstring+" is undefined");
            }
            obj = obj[names[i]];
        }
    }
    return obj;
}
```

To create the instance, use:

```
try{
    var Parser = getter(constructorClassName);
    var a = new Parser();
}catch(e){
    alert(e);
}
```

setTimeout() and setInterval()

Adobe AIR 1.0 and later

Replace the string passed as the handler function with a function reference or object. For example, replace a statement such as:

```
setTimeout("alert('Timeout')", 100);
```

with:

```
setTimeout(function(){alert('Timeout')}, 100);
```

Or, when the function requires the `this` object to be set by the caller, replace a statement such as:

```
this.appTimer = setInterval("obj.customFunction();", 100);
```

with the following:

```
var _self = this;
this.appTimer = setInterval(function(){obj.customFunction.apply(_self);}, 100);
```

Function constructor

Adobe AIR 1.0 and later

Calls to `new Function(param, body)` can be replaced with an inline function declaration or used only before the page load event has been handled.

javascript: URLs

Adobe AIR 1.0 and later

The code defined in a link using the javascript: URL scheme is ignored in the application sandbox. No unsafe JavaScript error is generated. You can replace links using javascript: URLs, such as:

```
<a href="javascript:code()">Click Me</a>
```

with:

```
<a href="#" onclick="code()">Click Me</a>
```

Event callbacks assigned through onevent attributes in innerHTML and outerHTML statements

Adobe AIR 1.0 and later

When you use innerHTML or outerHTML to add elements to the DOM of a document, any event callbacks assigned within the statement, such as onclick or onmouseover, are ignored. No security error is generated. Instead, you can assign an id attribute to the new elements and set the event handler callback functions using the addEventListener() method.

For example, given a target element in a document, such as:

```
<div id="container"></div>
```

Replace statements such as:

```
document.getElementById('container').innerHTML =  
    '<a href="#" onclick="code()">Click Me.</a>';
```

with:

```
document.getElementById('container').innerHTML = '<a href="#" id="smith">Click Me.</a>';  
document.getElementById('smith').addEventListener("click", function() { code(); });
```

Loading JavaScript files from outside the application installation directory

Adobe AIR 1.0 and later

Loading script files from outside the application sandbox is not permitted. No security error is generated. All script files that run in the application sandbox must be installed in the application directory. To use external scripts in a page, you must map the page to a different sandbox. See [“Loading application content into a non-application sandbox”](#) on page 35.

document.write() and document.writeln()

Adobe AIR 1.0 and later

Calls to document.write() or document.writeln() are ignored after the page load event has been handled. No security error is generated. As an alternative, you can load a new file, or replace the body of the document using DOM manipulation techniques.

Synchronous XMLHttpRequests before the load event or during a load event handler

Adobe AIR 1.0 and later

Synchronous XMLHttpRequests initiated before the page `load` event or during a `load` event handler do not return any content. Asynchronous XMLHttpRequests can be initiated, but do not return until after the `load` event. After the `load` event has been handled, synchronous XMLHttpRequests behave normally.

Dynamically created script elements

Adobe AIR 1.0 and later

Dynamically created script elements, such as when created with `innerHTML` or `document.createElement()` method are ignored.

Accessing AIR API classes from JavaScript

Adobe AIR 1.0 and later

In addition to the standard and extended elements of Webkit, HTML and JavaScript code can access the host classes provided by the runtime. These classes let you access the advanced features that AIR provides, including:

- Access to the file system
- Use of local SQL databases
- Control of application and window menus
- Access to sockets for networking
- Use of user-defined classes and objects
- Sound capabilities

For example, the AIR file API includes a `File` class, contained in the `flash.filesystem` package. You can create a `File` object in JavaScript as follows:

```
var myFile = new window.runtime.flash.filesystem.File();
```

The `runtime` object is a special JavaScript object, available to HTML content running in AIR in the application sandbox. It lets you access runtime classes from JavaScript. The `flash` property of the `runtime` object provides access to the `flash` package. In turn, the `flash.filesystem` property of the `runtime` object provides access to the `flash.filesystem` package (and this package includes the `File` class). Packages are a way of organizing classes used in ActionScript.

***Note:** The `runtime` property is not automatically added to the window objects of pages loaded in a frame or iframe. However, as long as the child document is in the application sandbox, the child can access the `runtime` property of the parent.*

Because the package structure of the runtime classes would require developers to type long strings of JavaScript code strings to access each class (as in `window.runtime.flash.desktop.NativeApplication`), the AIR SDK includes an `AIRAliases.js` file that lets you access runtime classes much more easily (for instance, by simply typing `air.NativeApplication`).

The AIR API classes are discussed throughout this guide. Other classes from the Flash Player API, which may be of interest to HTML developers, are described in the *Adobe AIR API Reference for HTML Developers*. ActionScript is the language used in SWF (Flash Player) content. However, JavaScript and ActionScript syntax are similar. (They are both based on versions of the ECMAScript language.) All built-in classes are available in both JavaScript (in HTML content) and ActionScript (in SWF content).

Note: JavaScript code cannot use the *Dictionary*, *XML*, and *XMLList* classes, which are available in ActionScript.

Note: For more information, see “[ActionScript 3.0 classes, packages, and namespaces](#)” on page 337 and “[ActionScript basics for JavaScript developers](#)” on page 335.

Using the AIRAliases.js file

Adobe AIR 1.0 and later

The runtime classes are organized in a package structure, as in the following:

- `window.runtime.flash.desktop.NativeApplication`
- `window.runtime.flash.desktop.ClipboardManager`
- `window.runtime.flash.filesystem.FileStream`
- `window.runtime.flash.data.SQLDatabase`

Included in the AIR SDK is an AIRAliases.js file that provide “alias” definitions that let you access the runtime classes with less typing. For example, you can access the classes listed above by simply typing the following:

- `air.NativeApplication`
- `air.Clipboard`
- `air.FileStream`
- `air.SQLDatabase`

This list is just a short subset of the classes in the AIRAliases.js file. The complete list of classes and package-level functions is provided in the *Adobe AIR API Reference for HTML Developers*.

In addition to commonly used runtime classes, the AIRAliases.js file includes aliases for commonly used package-level functions: `window.runtime.trace()`, `window.runtime.flash.net.navigateToURL()`, and `window.runtime.flash.net.sendToURL()`, which are aliased as `air.trace()`, `air.navigateToURL()`, and `air.sendToURL()`.

To use the AIRAliases.js file, include the following `script` reference in your HTML page:

```
<script src="AIRAliases.js"></script>
```

Adjust the path in the `src` reference, as needed.

Important: Except where noted, the JavaScript example code in this documentation assumes that you have included the AIRAliases.js file in your HTML page.

About URLs in AIR

Adobe AIR 1.0 and later

In HTML content running in AIR, you can use any of the following URL schemes in defining `src` attributes for `img`, `frame`, `iframe`, and `script` tags, in the `href` attribute of a link tag, or anywhere else you can provide a URL.

URL scheme	Description	Example
file	A path relative to the root of the file system.	file:///c:/AIR Test/test.txt
app	A path relative to the root directory of the installed application.	app:/images
app-storage	A path relative to the application store directory. For each installed application, AIR defines a unique application store directory, which is a useful place to store data specific to that application.	app-storage:/settings/prefs.xml
http	A standard HTTP request.	http://www.adobe.com
https	A standard HTTPS request.	https://secure.example.com

For more information about using URL schemes in AIR, see “[URI schemes](#)” on page 318.

Many of AIR APIs, including the File, Loader, URLStream, and Sound classes, use a URLRequest object rather than a string containing the URL. The URLRequest object itself is initialized with a string, which can use any of the same url schemes. For example, the following statement creates a URLRequest object that can be used to request the Adobe home page:

```
var urlReq = new air.URLRequest("http://www.adobe.com/");
```

For information about URLRequest objects see “[HTTP communications](#)” on page 316.

Embedding SWF content in HTML

Adobe AIR 1.0 and later

You can embed SWF content in HTML content within an AIR application just as you would in a browser. Embed the SWF content using an `object` tag, an `embed` tag, or both.

Note: A common web development practice is to use both an `object` tag and an `embed` tag to display SWF content in an HTML page. This practice has no benefit in AIR. You can use the W3C-standard `object` tag by itself in content to be displayed in AIR. At the same time, you can continue to use the `object` and `embed` tags together, if necessary, for HTML content that is also displayed in a browser.

If you have enabled transparency in the NativeWindow object displaying the HTML and SWF content, then AIR does not display the SWF content when window mode (`wmode`) used to embed the content is set to the value: `window`. To display SWF content in an HTML page of a transparent window, set the `wmode` parameter to `opaque` or `transparent`. The `window` is the default value for `wmode`, so if you do not specify a value, your content may not be displayed.

The following example illustrates the use of the HTML `object` tag to display a SWF file within HTML content. The `wmode` parameter is set to `opaque` so that the content is displayed, even if the underlying `NativeWindow` object is transparent. The SWF file is loaded from the application directory, but you can use any of the URL schemes supported by AIR. (The location from which the SWF file is loaded determines the security sandbox in which AIR places the content.)

```
<object type="application/x-shockwave-flash" width="100%" height="100%">
  <param name="movie" value="app:/SWFFile.swf"></param>
  <param name="wmode" value="opaque"></param>
</object>
```

You can also use a script to load content dynamically. The following example creates an `object` node to display the SWF file specified in the `urlString` parameter. The example adds the node as a child of the page element with the ID specified by the `elementID` parameter:

```
<script>
function showSWF(urlString, elementID) {
    var displayContainer = document.getElementById(elementID);
    var flash = createSWFObject(urlString, 'opaque', 650, 650);
    displayContainer.appendChild(flash);
}
function createSWFObject(urlString, wmodeString, width, height) {
    var SWFObject = document.createElement("object");
    SWFObject.setAttribute("type", "application/x-shockwave-flash");
    SWFObject.setAttribute("width", "100%");
    SWFObject.setAttribute("height", "100%");
    var movieParam = document.createElement("param");
    movieParam.setAttribute("name", "movie");
    movieParam.setAttribute("value", urlString);
    SWFObject.appendChild(movieParam);
    var wmodeParam = document.createElement("param");
    wmodeParam.setAttribute("name", "wmode");
    wmodeParam.setAttribute("value", wmodeString);
    SWFObject.appendChild(wmodeParam);
    return SWFObject;
}
</script>
```

SWF content is not displayed if the `HTMLLoader` object is scaled or rotated, or if the `alpha` property is set to a value other than 1.0. Prior to AIR 1.5.2, SWF content was not displayed in a transparent window no matter which `wmode` value was set.

Note: When an embedded SWF object attempts to load an external asset like a video file, the SWF content may not be rendered properly if an absolute path to the video file is not provided in the HTML file. However, an embedded SWF object can load an external image file using a relative path.

The following example depicts how external assets can be loaded through a SWF object embedded in an HTML content:

```
var imageLoader;

function showSWF(urlString, elementID){
    var displayContainer = document.getElementById(elementID);
    imageLoader = createSWFObject(urlString,650,650);
    displayContainer.appendChild(imageLoader);
}

function createSWFObject(urlString, width, height){

    var SWFObject = document.createElement("object");
    SWFObject.setAttribute("type","application/x-shockwave-flash");
    SWFObject.setAttribute("width","100%");
    SWFObject.setAttribute("height","100%");

    var movieParam = document.createElement("param");
    movieParam.setAttribute("name","movie");
    movieParam.setAttribute("value",urlString);
    SWFObject.appendChild(movieParam);

    var flashVars = document.createElement("param");
    flashVars.setAttribute("name","FlashVars");

    //Load the asset inside the SWF content.
    flashVars.setAttribute("value","imgPath=air.jpg");
    SWFObject.appendChild(flashVars);

    return SWFObject;
}
function loadImage()
{
    showSWF("ImageLoader.swf", "imageSpot");
}
}
```

In the following ActionScript example, the image path passed by the HTML file is read and the image is loaded on stage:

```
package
{
    import flash.display.Sprite;
    import flash.display.LoaderInfo;
    import flash.display.StageScaleMode;
    import flash.display.StageAlign;
    import flash.display.Loader;
    import flash.net.URLRequest;

    public class ImageLoader extends Sprite
    {
        public function ImageLoader()
        {

            var flashvars = LoaderInfo(this.loaderInfo).parameters;

            if(flashvars.imgPath){
                var imageLoader = new Loader();
                var image = new URLRequest(flashvars.imgPath);
                imageLoader.load(image);
                addChild(imageLoader);
                imageLoader.x = 0;
                imageLoader.y = 0;
                stage.scaleMode=StageScaleMode.NO_SCALE;
                stage.align=StageAlign.TOP_LEFT;
            }
        }
    }
}
```

Using ActionScript libraries within an HTML page

Adobe AIR 1.0 and later

AIR extends the HTML script element so that a page can import ActionScript classes in a compiled SWF file. For example, to import a library named, *myClasses.swf*, located in the `lib` subdirectory of the root application folder, include the following script tag within an HTML file:

```
<script src="lib/myClasses.swf" type="application/x-shockwave-flash"></script>
```

Important: The *type* attribute must be `type="application/x-shockwave-flash"` for the library to be properly loaded.

If the SWF content is compiled as a Flash Player 10 or AIR 1.5 SWF, you must set the XML namespace of the application descriptor file to the AIR 1.5 namespace.

The `lib` directory and `myClasses.swf` file must also be included when the AIR file is packaged.

Access the imported classes through the `runtime` property of the JavaScript Window object:

```
var libraryObject = new window.runtime.LibraryClass();
```

If the classes in the SWF file are organized in packages, you must include the package name as well. For example, if the `LibraryClass` definition was in a package named *utilities*, you would create an instance of the class with the following statement:

```
var libraryObject = new window.runtime.utilities.LibraryClass();
```

Note: To compile an ActionScript SWF library for use as part of an HTML page in AIR, use the `acompc` compiler. The `acompc` utility is part of the Flex SDK and is described in the Flex SDK documentation.

Accessing the HTML DOM and JavaScript objects from an imported ActionScript file

Adobe AIR 1.0 and later

To access objects in an HTML page from ActionScript in a SWF file imported into the page using the `<script>` tag, pass a reference to a JavaScript object, such as `window` or `document`, to a function defined in the ActionScript code. Use the reference within the function to access the JavaScript object (or other objects accessible through the passed-in reference).

For example, consider the following HTML page:

```
<html>
  <script src="ASLibrary.swf" type="application/x-shockwave-flash"></script>
  <script>
    num = 254;
    function getStatus() {
      return "OK.";
    }
    function runASFunction(window) {
      var obj = new runtime.ASClass();
      obj.accessDOM(window);
    }
  </script>
  <body onload="runASFunction">
    <p id="p1">Body text.</p>
  </body>
</html>
```

This simple HTML page has a JavaScript variable named `num` and a JavaScript function named `getStatus()`. Both of these are properties of the `window` object of the page. Also, the `window.document` object includes a named `P` element (with the ID `p1`).

The page loads an ActionScript file, “ASLibrary.swf,” that contains a class, `ASClass`. `ASClass` defines a function named `accessDOM()` that simply traces the values of these JavaScript objects. The `accessDOM()` method takes the JavaScript Window object as an argument. Using this Window reference, it can access other objects in the page including variables, functions, and DOM elements as illustrated in the following definition:

```
public class ASClass{
  public function accessDOM(window:*) :void {
    trace(window.num); // 254
    trace(window.document.getElementById("p1").innerHTML); // Body text..
    trace(window.getStatus()); // OK.
  }
}
```

You can both get and set properties of the HTML page from an imported ActionScript class. For example, the following function sets the contents of the `p1` element on the page and it sets the value of the `foo` JavaScript variable on the page:

```
public function modifyDOM(window:*) :void {
    window.document.getElementById("p1").innerHTML = "Bye";
    window.foo = 66;
```

Converting Date and RegExp objects

Adobe AIR 1.0 and later

The JavaScript and ActionScript languages both define Date and RegExp classes, but objects of these types are not automatically converted between the two execution contexts. You must convert Date and RegExp objects to the equivalent type before using them to set properties or function parameters in the alternate execution context.

For example, the following ActionScript code converts a JavaScript Date object named `jsDate` to an ActionScript Date object:

```
var asDate:Date = new Date(jsDate.getMilliseconds());
```

The following ActionScript code converts a JavaScript RegExp object named `jsRegExp` to an ActionScript RegExp object:

```
var flags:String = "";
if (jsRegExp.dotAll) flags += "s";
if (jsRegExp.extended) flags += "x";
if (jsRegExp.global) flags += "g";
if (jsRegExp.ignoreCase) flags += "i";
if (jsRegExp.multiline) flags += "m";
var asRegExp:RegExp = new RegExp(jsRegExp.source, flags);
```

Cross-scripting content in different security sandboxes

Adobe AIR 1.0 and later

The runtime security model isolates code from different origins. By cross-scripting content in different security sandboxes, you can allow content in one security sandbox to access selected properties and methods in another sandbox.

AIR security sandboxes and JavaScript code

Adobe AIR 1.0 and later

AIR enforces a same-origin policy that prevents code in one domain from interacting with content in another. All files are placed in a sandbox based on their origin. Ordinarily, content in the application sandbox cannot violate the same-origin principle and cross-script content loaded from outside the application install directory. However, AIR provides a few techniques that let you cross-script non-application content.

One technique uses frames or iframes to map application content into a different security sandbox. Any pages loaded from the sandboxed area of the application behave as if they were loaded from the remote domain. For example, by mapping application content to the *example.com* domain, that content could cross-script pages loaded from *example.com*.

Since this technique places the application content into a different sandbox, code within that content is also no longer subject to the restrictions on the execution of code in evaluated strings. You can use this sandbox mapping technique to ease these restrictions even when you don't need to cross-script remote content. Mapping content in this way can be especially useful when working with one of the many JavaScript frameworks or with existing code that relies on evaluating strings. However, you should consider and guard against the additional risk that untrusted content could be injected and executed when content is run outside the application sandbox.

At the same time, application content mapped to another sandbox loses its access to the AIR APIs, so the sandbox mapping technique cannot be used to expose AIR functionality to code executed outside the application sandbox.

Another cross-scripting technique lets you create an interface called a *sandbox bridge* between content in a non-application sandbox and its parent document in the application sandbox. The bridge allows the child content to access properties and methods defined by the parent, the parent to access properties and methods defined by the child, or both.

Finally, you can also perform cross-domain XMLHttpRequests from the application sandbox and, optionally, from other sandboxes.

For more information, see “[HTML frame and iframe elements](#)” on page 12, “[HTML security in Adobe AIR](#)” on page 73, and “[The XMLHttpRequest object](#)” on page 6.

Loading application content into a non-application sandbox

Adobe AIR 1.0 and later

To allow application content to safely cross-script content loaded from outside the application install directory, you can use `frame` or `iframe` elements to load application content into the same security sandbox as the external content. If you do not need to cross-script remote content, but still wish to load a page of your application outside the application sandbox, you can use the same technique, specifying `http://localhost/` or some other innocuous value, as the domain of origin.

AIR adds the new attributes, `sandboxRoot` and `documentRoot`, to the `frame` element that allow you to specify whether an application file loaded into the frame should be mapped to a non-application sandbox. Files resolving to a path underneath the `sandboxRoot` URL are loaded instead from the `documentRoot` directory. For security purposes, the application content loaded in this way is treated as if it was actually loaded from the `sandboxRoot` URL.

The `sandboxRoot` property specifies the URL to use for determining the sandbox and domain in which to place the frame content. The `file:`, `http:`, or `https:` URL schemes must be used. If you specify a relative URL, the content remains in the application sandbox.

The `documentRoot` property specifies the directory from which to load the frame content. The `file:`, `app:`, or `app-storage:` URL schemes must be used.

The following example maps content installed in the `sandbox` subdirectory of the application to run in the remote sandbox and the `www.example.com` domain:

```
<iframe
  src="http://www.example.com/local/ui.html"
  sandboxRoot="http://www.example.com/local/"
  documentRoot="app:/sandbox/" >
</iframe>
```

The `ui.html` page could load a javascript file from the `local`, `sandbox` folder using the following script tag:

```
<script src="http://www.example.com/local/ui.js"></script>
```

It could also load content from a directory on the remote server using a script tag such as the following:

```
<script src="http://www.example.com/remote/remote.js"></script>
```

The `sandboxRoot` URL will mask any content at the same URL on the remote server. In the above example, you would not be able to access any remote content at `www.example.com/local/` (or any of its subdirectories) because AIR remaps the request to the local application directory. Requests are remapped whether they derive from page navigation, from an XMLHttpRequest, or from any other means of loading content.

Setting up a sandbox bridge interface

Adobe AIR 1.0 and later

You can use a sandbox bridge when content in the application sandbox must access properties or methods defined by content in a non-application sandbox, or when non-application content must access properties and methods defined by content in the application sandbox. Create a bridge with the `childSandboxBridge` and `parentSandboxBridge` properties of the `window` object of any child document.

Establishing a child sandbox bridge

Adobe AIR 1.0 and later

The `childSandboxBridge` property allows the child document to expose an interface to content in the parent document. To expose an interface, you set the `childSandbox` property to a function or object in the child document. You can then access the object or function from content in the parent document. The following example shows how a script running in a child document can expose an object containing a function and a property to its parent:

```
var interface = {};  
interface.calculatePrice = function() {  
    return ".45 cents";  
}  
interface.storeID = "abc"  
window.childSandboxBridge = interface;
```

If this child content was loaded into an `iframe` assigned an `id` of "child", you could access the interface from parent content by reading the `childSandboxBridge` property of the frame:

```
var childInterface = document.getElementById("child").contentWindow.childSandboxBridge;  
air.trace(childInterface.calculatePrice()); //traces ".45 cents"  
air.trace(childInterface.storeID); //traces "abc"
```

Establishing a parent sandbox bridge

Adobe AIR 1.0 and later

The `parentSandboxBridge` property allows the parent document to expose an interface to content in a child document. To expose an interface, the parent document sets the `parentSandbox` property of the child document to a function or object defined in the parent document. You can then access the object or function from content in the child. The following example shows how a script running in a parent frame can expose an object containing a function to a child document:

```
var interface = {};  
interface.save = function(text){  
    var saveFile = air.File("app-storage:/save.txt");  
    //write text to file  
}  
document.getElementById("child").contentWindow.parentSandboxBridge = interface;
```

Using this interface, content in the child frame could save text to a file named `save.txt`, but would not have any other access to the file system. The child content could call the `save` function as follows:

```
var textToSave = "A string."  
window.parentSandboxBridge.save(textToSave);
```

Application content should expose the narrowest interface possible to other sandboxes. Non-application content should be considered inherently untrustworthy since it may be subject to accidental or malicious code injection. You must put appropriate safeguards in place to prevent misuse of the interface you expose through the parent sandbox bridge.

Accessing a parent sandbox bridge during page loading

Adobe AIR 1.0 and later

In order for a script in a child document to access a parent sandbox bridge, the bridge must be set up before the script is run. Window, frame and iframe objects dispatch a `dominitialize` event when a new page DOM has been created, but before any scripts have been parsed, or DOM elements added. You can use the `dominitialize` event to establish the bridge early enough in the page construction sequence that all scripts in the child document can access it.

The following example illustrates how to create a parent sandbox bridge in response to the `dominitialize` event dispatched from the child frame:

```
<html>  
<head>  
<script>  
var bridgeInterface = {};  
bridgeInterface.testProperty = "Bridge engaged";  
function engageBridge(){  
    document.getElementById("sandbox").contentWindow.parentSandboxBridge = bridgeInterface;  
}  
</script>  
</head>  
<body>  
<iframe id="sandbox"  
    src="http://www.example.com/air/child.html"  
    documentRoot="app:/"  
    sandboxRoot="http://www.example.com/air/"  
    ondominitialize="engageBridge()"/>  
</body>  
</html>
```

The following `child.html` document illustrates how child content can access the parent sandbox bridge:

```
<html>
  <head>
    <script>
      document.write(window.parentSandboxBridge.testProperty);
    </script>
  </head>
  <body></body>
</html>
```

To listen for the `dominitialize` event on a child window, rather than a frame, you must add the listener to the new child window object created by the `window.open()` function:

```
var childWindow = window.open();
childWindow.addEventListener("dominitialize", engageBridge());
childWindow.document.location = "http://www.example.com/air/child.html";
```

In this case, there is no way to map application content into a non-application sandbox. This technique is only useful when `child.html` is loaded from outside the application directory. You can still map application content in the window to a non-application sandbox, but you must first load an intermediate page that itself uses frames to load the child document and map it to the desired sandbox.

If you use the `HTMLLoader` class `createRootWindow()` function to create a window, the new window is not a child of the document from which `createRootWindow()` is called. Thus, you cannot create a sandbox bridge from the calling window to non-application content loaded into the new window. Instead, you must use load an intermediate page in the new window that itself uses frames to load the child document. You can then establish the bridge from the parent document of the new window to the child document loaded into the frame.

Chapter 3: Handling HTML-related events in AIR

Adobe AIR 1.0 and later

An event-handling system allows programmers to respond to user input and system events in a convenient way. The Adobe® AIR® event model is not only convenient, but also standards-compliant. Based on the Document Object Model (DOM) Level 3 Events Specification, an industry-standard event-handling architecture, the event model provides a powerful, yet intuitive, event-handling tool for programmers.

HTMLLoader events

Adobe AIR 1.0 and later

An HTMLLoader object dispatches the following Adobe® ActionScript® 3.0 events:

Event	Description
<code>htmlDOMInitialize</code>	Dispatched when the HTML document is created, but before any scripts are parsed or DOM nodes are added to the page.
<code>complete</code>	Dispatched when the HTML DOM has been created in response to a load operation, immediately after the <code>onload</code> event in the HTML page.
<code>htmlBoundsChanged</code>	Dispatched when one or both of the <code>contentWidth</code> and <code>contentHeight</code> properties have changed.
<code>locationChange</code>	Dispatched when the location property of the HTMLLoader has changed.
<code>locationChanging</code>	<p>Dispatched before the location of the HTMLLoader changes because of user navigation, a JavaScript call, or a redirect. The <code>locationChanging</code> event is not dispatched when you call the <code>load()</code>, <code>loadString()</code>, <code>reload()</code>, <code>historyGo()</code>, <code>historyForward()</code>, or <code>historyBack()</code> methods.</p> <p>Calling the <code>preventDefault()</code> method of the dispatched event object cancels navigation.</p> <p>If a link is opened in the system browser, a <code>locationChanging</code> event is not dispatched since the HTMLLoader does not change location.</p>
<code>scroll</code>	Dispatched anytime the HTML engine changes the scroll position. Scroll events can be because of navigation to anchor links (# links) in the page or because of calls to the <code>window.scrollTo()</code> method. Entering text in a text input or text area can also cause a scroll event.
<code>uncaughtScriptException</code>	Dispatched when a JavaScript exception occurs in the HTMLLoader and the exception is not caught in JavaScript code.

How AIR class-event handling differs from other event handling in the HTML DOM

Adobe AIR 1.0 and later

The HTML DOM provides a few different ways to handle events:

- Defining an `on` event handler within an HTML element opening tag, as in:

```
<div id="myDiv" onclick="myHandler()">
```

- Callback function properties, such as:

```
document.getElementById("myDiv").onclick
```

- Event listeners that you register using the `addEventListener()` method, as in:

```
document.getElementById("myDiv").addEventListener("click", clickHandler)
```

However, since runtime objects do not appear in the DOM, you can only add event listeners by calling the `addEventListener()` method of an AIR object.

As in JavaScript, events dispatched by AIR objects can be associated with default behaviors. (A *default behavior* is an action that AIR executes as the normal consequence of certain events.)

The event objects dispatched by runtime objects are an instance of the `Event` class or one of its subclasses. An event object not only stores information about a specific event, but also contains methods that facilitate manipulation of the event object. For example, when AIR detects an I/O error event when reading a file asynchronously, it creates an event object (an instance of the `IOErrorEvent` class) to represent that particular I/O error event.

Any time you write event handler code, it follows the same basic structure:

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

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

This code does two things. First, it defines a handler function, which is the way to specify the actions to be performed in response to the event. Next, it calls the `addEventListener()` method of the source object, in essence subscribing the function to the specified event so that when the event happens, the handler actions are carried out. When the event actually happens, the event target checks its list of all the functions and methods that are registered with event listeners. It then calls each one in turn, passing the event object as a parameter.

Default behaviors

Adobe AIR 1.0 and later

Developers are usually responsible for writing code that responds to events. In some cases, however, a behavior is so commonly associated with an event that AIR automatically executes the behavior unless the developer adds code to cancel it. Because AIR automatically exhibits the behavior, such behaviors are called default behaviors.

For example, when a user clicks the close box of a window of an application, the expectation that the window will close is so common that the behavior is built into AIR. If you do not want this default behavior to occur, you can cancel it using the event-handling system. When a user clicks the close box of a window, the `NativeWindow` object that represents the window dispatches a `closing` event. To prevent the runtime from closing the window, you must call the `preventDefault()` method of the dispatched event object.

Not all default behaviors can be prevented. For example, the runtime generates an `OutputProgressEvent` object as a `FileStream` object writes data to a file. The default behavior, which cannot be prevented, is that the content of the file is updated with the new data.

Many types of event objects do not have associated default behaviors. For example, a `Sound` object dispatches an `id3` event when enough data from an MP3 file is read to provide ID3 information, but there is no default behavior associated with it. The API documentation for the `Event` class and its subclasses lists each type of event and describes any associated default behavior, and whether that behavior can be prevented.

***Note:** Default behaviors are associated only with event objects dispatched by the runtime directly, and do not exist for event objects dispatched programmatically through JavaScript. For example, you can use the methods of the `EventDispatcher` class to dispatch an event object, but dispatching the event does not trigger the default behavior.*

The event flow

Adobe AIR 1.0 and later

SWF file content running in AIR uses the ActionScript 3.0 display list architecture to display visual content. The ActionScript 3.0 display list provides a parent-child relationship for content, and events (such as mouse-click events) in SWF file content that propagates between parent and child display objects. The HTML DOM has its own, separate event flow that traverses only the DOM elements. When writing HTML-based applications for AIR, you primarily use the HTML DOM instead of the ActionScript 3.0 display list, so you can generally disregard the information on event phases that appears in the AIR reference documentation.

Adobe AIR event objects

Adobe AIR 1.0 and later

Event objects serve two main purposes in the event-handling system. First, event objects represent actual events by storing information about specific events in a set of properties. Second, event objects contain a set of methods that allow you to manipulate event objects and affect the behavior of the event-handling system.

The AIR API defines an `Event` class that serves as the base class for all event objects dispatched by the AIR API classes. The `Event` class defines a fundamental set of properties and methods that are common to all event objects.

To use `Event` objects, it's important to first understand the `Event` class properties and methods and why subclasses of the `Event` class exist.

Understanding Event class properties

Adobe AIR 1.0 and later

The Event class defines several read-only properties and constants that provide important information about an event. The following are especially important:

- `Event.type` describes the type of event that an event object represents.
- `Event.cancelable` is a Boolean value that reports whether the default behavior associated with the event, if any, can be canceled.
- Event flow information is contained in the remaining properties, and is only of interest when using ActionScript 3.0 in SWF content in AIR.

Event object types

Adobe AIR 1.0 and later

Every event object has an associated event type. Event types are stored in the `Event.type` property as string values. It is useful to know the type of an event object so that your code can distinguish objects of different types from one another. For example, the following code registers a `fileReadHandler()` listener function to respond to a `complete` event dispatched by `myFileStream`:

```
myFileStream.addEventListener(Event.COMPLETE, fileReadHandler);
```

The AIR Event class defines many class constants, such as `COMPLETE`, `CLOSING`, and `ID3`, to represent the types of events dispatched by runtime objects. These constants are listed in the Event class page of the [Adobe AIR API Reference for HTML Developers](#).

Event constants provide an easy way to refer to specific event types. Using a constant instead of the string value helps you identify typographical errors more quickly. If you misspell a constant name in your code, the JavaScript parser will catch the mistake. If you instead misspell an event string, the event handler will be registered for a type of event that will never be dispatched. Thus, when adding an event listener, it is a better practice to use the following code:

```
myFileStream.addEventListener(Event.COMPLETE, htmlRenderHandler);
```

rather than:

```
myFileStream.addEventListener("complete", htmlRenderHandler);
```

Default behavior information

Adobe AIR 1.0 and later

Your code can check whether the default behavior for any given event object can be prevented by accessing the `cancelable` property. The `cancelable` property holds a Boolean value that indicates whether a default behavior can be prevented. You can prevent, or cancel, the default behavior associated with a small number of events using the `preventDefault()` method. For more information, see “[Canceling default event behavior](#)” on page 43.

Understanding Event class methods

Adobe AIR 1.0 and later

There are three categories of Event class methods:

- Utility methods, which can create copies of an event object or convert it to a string.

- Event flow methods, which remove event objects from the event flow (primarily of use when using ActionScript 3.0 in SWF content for the runtime—see “[The event flow](#)” on page 41).
- Default behavior methods, which prevent default behavior or check whether it has been prevented.

Event class utility methods

Adobe AIR 1.0 and later

The Event class has two utility methods. The `clone()` method allows you to create copies of an event object. The `toString()` method allows you to generate a string representation of the properties of an event object along with their values.

Canceling default event behavior

Adobe AIR 1.0 and later

The two methods that pertain to canceling default behavior are the `preventDefault()` method and the `isDefaultPrevented()` method. Call the `preventDefault()` method to cancel the default behavior associated with an event. Check whether `preventDefault()` has already been called on an event object, with the `isDefaultPrevented()` method.

The `preventDefault()` method works only if the event's default behavior can be canceled. You can check whether an event has behavior that can be canceled by referring to the API documentation, or by examining the `cancelable` property of the event object.

Canceling the default behavior has no effect on the progress of an event object through the event flow. Use the event flow methods of the Event class to remove an event object from the event flow.

Subclasses of the Event class

Adobe AIR 1.0 and later

For many events, the common set of properties defined in the Event class is sufficient. Representing other events, however, requires properties not available in the Event class. For these events, the AIR API defines several subclasses of the Event class.

Each subclass provides additional properties and event types that are unique to that category of events. For example, events related to mouse input provide properties describing the mouse location when the event occurred. Likewise, the `InvokeEvent` class adds properties containing the file path of the invoking file and any arguments passed as parameters in the command-line invocation.

An Event subclass frequently defines additional constants to represent the event types that are associated with the subclass. For example, the `FileListEvent` class defines constants for the `directoryListing` and `selectMultiple` event types.

Handling runtime events with JavaScript

Adobe AIR 1.0 and later

The runtime classes support adding event handlers with the `addEventListener()` method. To add a handler function for an event, call the `addEventListener()` method of the object that dispatches the event, providing the event type and the handling function. For example, to listen for the `closing` event dispatched when a user clicks the window close button on the title bar, use the following statement:

```
window.nativeWindow.addEventListener(air.NativeWindow.CLOSING, handleWindowClosing);
```

The type parameter of the `addEventListener()` method is a string, but the AIR APIs define constants for all runtime event types. Using these constants can help pinpoint typographic errors entered in the type parameter more quickly than using the string version.

Creating an event handler function

Adobe AIR 1.0 and later

The following code creates a simple HTML file that displays information about the position of the main window. A handler function named `moveHandler()`, listens for a move event (defined by the `NativeWindowBoundsEvent` class) of the main window.

```
<html>
  <script src="AIRAliases.js" />
  <script>
    function init() {
      writeValues();
      window.nativeWindow.addEventListener(air.NativeWindowBoundsEvent.MOVE,
                                           moveHandler);
    }
    function writeValues() {
      document.getElementById("xText").value = window.nativeWindow.x;
      document.getElementById("yText").value = window.nativeWindow.y;
    }
    function moveHandler(event) {
      air.trace(event.type); // move
      writeValues();
    }
  </script>
  <body onload="init()" />
    <table>
      <tr>
        <td>Window X:</td>
        <td><textarea id="xText"></textarea></td>
      </tr>
      <tr>
        <td>Window Y:</td>
        <td><textarea id="yText"></textarea></td>
      </tr>
    </table>
  </body>
</html>
```

When a user moves the window, the textarea elements display the updated X and Y positions of the window:

Notice that the event object is passed as an argument to the `moveHandler()` method. The event parameter allows your handler function to examine the event object. In this example, you use the event object's `type` property to report that the event is a `move` event.

Note: Do not use parentheses when you specify the `listener` parameter. For example, the `moveHandler()` function is specified without parentheses in the following call to the `addEventListener()` method:
`addEventListener(Event.MOVE, moveHandler).`

The `addEventListener()` method has three other parameters, described in the [Adobe AIR API Reference for HTML Developers](#); these parameters are `useCapture`, `priority`, and `useWeakReference`.

Removing event listeners

Adobe AIR 1.0 and later

You can use the `removeEventListener()` method to remove an event listener that you no longer need. It is a good idea to remove any listeners that will no longer be used. Required parameters include the `eventName` and `listener` parameters, which are the same as the required parameters for the `addEventListener()` method.

Removing event listeners in HTML pages that navigate

Adobe AIR 1.0 and later

When HTML content navigates, or when HTML content is discarded because a window that contains it is closed, the event listeners that reference objects on the unloaded page are not automatically removed. When an object dispatches an event to a handler that has already been unloaded, you see the following error message: “The application attempted to reference a JavaScript object in an HTML page that is no longer loaded.”

To avoid this error, remove JavaScript event listeners in an HTML page before it goes away. In the case of page navigation (within an `HTMLLoader` object), remove the event listener during the `unload` event of the `window` object.

For example, the following JavaScript code removes an event listener for an `uncaughtScriptException` event:

```
window.onunload = cleanup;
window.htmlLoader.addEventListener('uncaughtScriptException', uncaughtScriptExceptionHandler);
function cleanup()
{
    window.htmlLoader.removeEventListener('uncaughtScriptException',
        uncaughtScriptExceptionHandler);
}
```

To prevent the error from occurring when closing windows that contain HTML content, call a cleanup function in response to the `closing` event of the `NativeWindow` object (`window.nativeWindow`). For example, the following JavaScript code removes an event listener for an `uncaughtScriptException` event:

```
window.nativeWindow.addEventListener(air.Event.CLOSING, cleanup);
function cleanup()
{
    window.htmlLoader.removeEventListener('uncaughtScriptException',
        uncaughtScriptExceptionHandler);
}
```

You can also prevent this error from occurring by removing an event listener as soon as it runs (if the event only needs to be handled once). For example, the following JavaScript code creates an HTML window by calling the `createRootWindow()` method of the `HTMLLoader` class and adds an event listener for the `complete` event. When the `complete` event handler is called, it removes its own event listener using the `removeEventListener()` function:

```
var html = runtime.flash.html.HTMLLoader.createRootWindow(true);
html.addEventListener('complete', htmlCompleteListener);
function htmlCompleteListener()
{
    html.removeEventListener(complete, arguments.callee)
    // handler code..
}
html.load(new runtime.flash.net.URLRequest("second.html"));
```

Removing unneeded event listeners also allows the system garbage collector to reclaim any memory associated with those listeners.

Checking for existing event listeners

Adobe AIR 1.0 and later

The `hasEventListener()` method lets you check for the existence of an event listener on an object.

Error events without listeners

Adobe AIR 1.0 and later

Exceptions, rather than events, are the primary mechanism for error handling in the runtime classes. However, exception handling does not work for asynchronous operations such as loading files. If an error occurs during an asynchronous operation, the runtime dispatches an error event object. If you do not create a listener for the error event, the AIR Debug Launcher presents a dialog box with information about the error.

Most error events are based on the `ErrorEvent` class, and have a property named `text` that is used to store a descriptive error message. An exception is the `StatusEvent` class, which has a `level` property instead of a `text` property. When the value of the `level` property is `error`, the `StatusEvent` is considered to be an error event.

An error event does not cause an application to stop executing. It manifests only as a dialog box on the AIR Debug Launcher. It does not manifest at all in the installed AIR application running in the runtime.

Chapter 4: Scripting the AIR HTML Container

Adobe AIR 1.0 and later

The `HTMLLoader` class serves as the container for HTML content in Adobe® AIR®. The class provides many properties and methods for controlling the behavior and appearance of the HTML content. In addition, the class defines properties and methods for such tasks as loading and interacting with HTML content and managing history.

The `HTMLHost` class defines a set of default behaviors for an `HTMLLoader`. When you create an `HTMLLoader` object, no `HTMLHost` implementation is provided. Thus when HTML content triggers one of the default behaviors, such as changing the window location, or the window title, nothing happens. You can extend the `HTMLHost` class to define the behaviors desired for your application.

A default implementation of the `HTMLHost` is provided for HTML windows created by AIR. You can assign the default `HTMLHost` implementation to another `HTMLLoader` object by setting the `htmlHost` property of the object using a new `HTMLHost` object created with the `defaultBehavior` parameter set to `true`.

The `HTMLHost` class can only be extended using ActionScript. In an HTML-based application, you can import a compiled SWF file containing an implementation of the `HTMLHost` class. Assign the host class implementation using the `window.htmlLoader` property:

```
<script src="HTMLHostLibrary.swf" type="application/x-shockwave-flash"></script>
<script>
    window.htmlLoader.htmlHost = new window.runtime.HTMLHostImplementation();
</script>
```

Display properties of HTMLLoader objects

Adobe AIR 1.0 and later

An `HTMLLoader` object inherits the display properties of the Adobe® Flash® Player Sprite class. You can resize, move, hide, and change the background color, for example. Or you can apply advanced effects like filters, masks, scaling, and rotation. When applying effects, consider the impact on legibility. SWF and PDF content loaded into an HTML page cannot be displayed when some effects are applied.

HTML windows contain an `HTMLLoader` object that renders the HTML content. This object is constrained within the area of the window, so changing the dimensions, position, rotation, or scale factor does not always produce desirable results.

Basic display properties

Adobe AIR 1.0 and later

The basic display properties of the `HTMLLoader` allow you to position the control within its parent display object, to set the size, and to show or hide the control. You should not change these properties for the `HTMLLoader` object of an HTML window.

The basic properties include:

Property	Notes
x, y	Positions the object within its parent container.
width, height	Changes the dimensions of the display area.
visible	Controls the visibility of the object and any content it contains.

Outside an HTML window, the `width` and `height` properties of an `HTMLLoader` object default to 0. You must set the width and height before the loaded HTML content can be seen. HTML content is drawn to the `HTMLLoader` size, laid out according to the HTML and CSS properties in the content. Changing the `HTMLLoader` size reflows the content.

When loading content into a new `HTMLLoader` object (with `width` still set to 0), it can be tempting to set the display width and height of the `HTMLLoader` using the `contentWidth` and `contentHeight` properties. This technique works for pages that have a reasonable minimum width when laid out according to the HTML and CSS flow rules. However, some pages flow into a long and narrow layout in the absence of a reasonable width provided by the `HTMLLoader`.

Note: When you change the width and height of an `HTMLLoader` object, the `scaleX` and `scaleY` values do not change, as would happen with most other types of display objects.

Transparency of HTMLLoader content

Adobe AIR 1.0 and later

The `paintsDefaultBackground` property of an `HTMLLoader` object, which is `true` by default, determines whether the `HTMLLoader` object draws an opaque background. When `paintsDefaultBackground` is `false`, the background is clear. The display object container or other display objects below the `HTMLLoader` object are visible behind the foreground elements of the HTML content.

If the body element or any other element of the HTML document specifies a background color (using `style="background-color:gray"`, for example), then the background of that portion of the HTML is opaque and rendered with the specified background color. If you set the `opaqueBackground` property of the `HTMLLoader` object, and `paintsDefaultBackground` is `false`, then the color set for the `opaqueBackground` is visible.

Note: You can use a transparent, PNG-format graphic to provide an alpha-blended background for an element in an HTML document. Setting the opacity style of an HTML element is not supported.

Scaling HTMLLoader content

Adobe AIR 1.0 and later

Avoid scaling an `HTMLLoader` object beyond a scale factor of 1.0. Text in `HTMLLoader` content is rendered at a specific resolution and appears pixelated if the `HTMLLoader` object is scaled up.

Considerations when loading SWF or PDF content in an HTML page

Adobe AIR 1.0 and later

SWF and PDF content loaded into in an `HTMLLoader` object disappears in the following conditions:

- If you scale the `HTMLLoader` object to a factor other than 1.0.

- If you set the alpha property of the HTMLLoader object to a value other than 1.0.
- If you rotate the HTMLLoader content.

The content reappears if you remove the offending property setting and remove the active filters.

In addition, the runtime cannot display PDF content in transparent windows. The runtime only displays SWF content embedded in an HTML page when the `wmode` parameter of the object or `embed` tag is set to `opaque` or `transparent`. Since the default value of `wmode` is `window`, SWF content is not displayed in transparent windows unless you explicitly set the `wmode` parameter.

Note: Prior to AIR 1.5.2, SWF embedded in HTML could not be displayed no matter which `wmode` value was used.

For more information on loading these types of media in an HTMLLoader, see “[Embedding SWF content in HTML](#)” on page 29 and “[Adding PDF content in AIR](#)” on page 270.

Advanced display properties

Adobe AIR 1.0 and later

The HTMLLoader class inherits several methods that can be used for special effects. In general, these effects have limitations when used with the HTMLLoader display, but they can be useful for transitions or other temporary effects. For example, if you display a dialog window to gather user input, you could blur the display of the main window until the user closes the dialog. Likewise, you could fade the display out when closing a window.

The advanced display properties include:

Property	Limitations
<code>alpha</code>	Can reduce the legibility of HTML content
<code>filters</code>	In an HTML Window, exterior effects are clipped by the window edge
<code>graphics</code>	Shapes drawn with graphics commands appear below HTML content, including the default background. The <code>paintsDefaultBackground</code> property must be false for the drawn shapes to be visible.
<code>opaqueBackground</code>	Does not change the color of the default background. The <code>paintsDefaultBackground</code> property must be false for this color layer to be visible.
<code>rotation</code>	The corners of the rectangular HTMLLoader area can be clipped by the window edge. SWF and PDF content loaded in the HTML content is not displayed.
<code>scaleX, scaleY</code>	The rendered display can appear pixelated at scale factors greater than 1. SWF and PDF content loaded in the HTML content is not displayed.
<code>transform</code>	Can reduce legibility of HTML content. The HTML display can be clipped by the window edge. SWF and PDF content loaded in the HTML content is not displayed if the transform involves rotation, scaling, or skewing.

The following example illustrates how to set the `filters` array to blur the entire HTML display:

```
var blur = new window.runtime.flash.filters.BlurFilter();
var filters = [blur];
window.htmlLoader.filters = filters;
```

Note: Display object classes, such as `Sprite` and `BlurFilter`, are not commonly used in HTML-based applications. They are not listed in the [Adobe AIR API Reference for HTML Developers](#) nor aliased in the `AIRAliases.js` file. For documentation about these classes, consult the [ActionScript 3.0 Reference for the Adobe Flash Platform](#).

Accessing the HTML history list

Adobe AIR 1.0 and later

As new pages are loaded in an `HTMLLoader` object, the runtime maintains a history list for the object. The history list corresponds to the `window.history` object in the HTML page. The `HTMLLoader` class includes the following properties and methods that let you work with the HTML history list:

Class member	Description
<code>historyLength</code>	The overall length of the history list, including back and forward entries.
<code>historyPosition</code>	The current position in the history list. History items before this position represent "back" navigation, and items after this position represent "forward" navigation.
<code>getHistoryAt()</code>	Returns the <code>URLRequest</code> object corresponding to the history entry at the specified position in the history list.
<code>historyBack()</code>	Navigates back in the history list, if possible.
<code>historyForward()</code>	Navigates forward in the history list, if possible.
<code>historyGo()</code>	Navigates the indicated number of steps in the browser history. Navigates forward if positive, backward if negative. Navigating to zero reloads the page. Specifying a position beyond the end navigates to the end of the list.

Items in the history list are stored as objects of type [HTMLHistoryItem](#). The `HTMLHistoryItem` class has the following properties:

Property	Description
<code>isPost</code>	Set to <code>true</code> if the HTML page includes POST data.
<code>originalUrl</code>	The original URL of the HTML page, before any redirects.
<code>title</code>	The title of the HTML page.
<code>url</code>	The URL of the HTML page.

Setting the user agent used when loading HTML content

Adobe AIR 1.0 and later

The `HTMLLoader` class has a `userAgent` property, which lets you set the user agent string used by the `HTMLLoader`. Set the `userAgent` property of the `HTMLLoader` object before calling the `load()` method. If you set this property on the `HTMLLoader` instance, then the `userAgent` property of the `URLRequest` passed to the `load()` method is *not* used.

You can set the default user agent string used by all `HTMLLoader` objects in an application domain by setting the `URLRequestDefaults.userAgent` property. The static `URLRequestDefaults` properties apply as defaults for all `URLRequest` objects, not only `URLRequest`s used with the `load()` method of `HTMLLoader` objects. Setting the `userAgent` property of an `HTMLLoader` overrides the default `URLRequestDefaults.userAgent` setting.

If you do not set a user agent value for either the `userAgent` property of the `HTMLLoader` object or for `URLRequestDefaults.userAgent`, then the default AIR user agent value is used. This default value varies depending on the runtime operating system (such as Mac OS or Windows), the runtime language, and the runtime version, as in the following two examples:

- "Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/420+ (KHTML, like Gecko) AdobeAIR/1.0"
- "Mozilla/5.0 (Windows; U; en) AppleWebKit/420+ (KHTML, like Gecko) AdobeAIR/1.0"

Setting the character encoding to use for HTML content

Adobe AIR 1.0 and later

An HTML page can specify the character encoding it uses by including `meta` tag, such as the following:

```
meta http-equiv="content-type" content="text/html" charset="ISO-8859-1";
```

Override the page setting to ensure that a specific character encoding is used by setting the `textEncodingOverride` property of the `HTMLLoader` object:

```
window.htmlLoader.textEncodingOverride = "ISO-8859-1";
```

Specify the character encoding for the `HTMLLoader` content to use when an HTML page does not specify a setting with the `textEncodingFallback` property of the `HTMLLoader` object:

```
window.htmlLoader.textEncodingFallback = "ISO-8859-1";
```

The `textEncodingOverride` property overrides the setting in the HTML page. And the `textEncodingOverride` property and the setting in the HTML page override the `textEncodingFallback` property.

Set the `textEncodingOverride` property or the `textEncodingFallback` property before loading the HTML content.

Defining browser-like user interfaces for HTML content

Adobe AIR 1.0 and later

JavaScript provides several APIs for controlling the window displaying the HTML content. In AIR, these APIs can be overridden by implementing a custom [HTMLHost](#) class.

Important: You can only create a custom implementation of the `HTMLHost` class using `ActionScript`. You can import and use a compiled `ActionScript` (SWF) file containing a custom implementation in an HTML page. See [“Using ActionScript libraries within an HTML page”](#) on page 32 for more information about importing `ActionScript` libraries into HTML.

About extending the HTMLHost class

Adobe AIR 1.0 and later

The AIR HTMLHost class controls the following JavaScript properties and methods:

- `window.status`
- `window.document.title`
- `window.location`
- `window.blur()`
- `window.close()`
- `window.focus()`
- `window.moveBy()`
- `window.moveTo()`
- `window.open()`
- `window.resizeBy()`
- `window.resizeTo()`

When you create an HTMLLoader object using `new HTMLLoader()`, the listed JavaScript properties or methods are not enabled. The HTMLHost class provides a default, browser-like implementation of these JavaScript APIs. You can also extend the HTMLHost class to customize the behavior. To create an HTMLHost object supporting the default behavior, set the `defaultBehaviors` parameter to `true` in the HTMLHost constructor:

```
var defaultHost = new HTMLHost(true);
```

When you create an HTML window in AIR with the HTMLLoader class `createRootWindow()` method, an HTMLHost instance supporting the default behaviors is assigned automatically. You can change the host object behavior by assigning a different HTMLHost implementation to the `htmlHost` property of the HTMLLoader, or you can assign `null` to disable the features entirely.

***Note:** AIR assigns a default HTMLHost object to the initial window created for an HTML-based AIR application and any windows created by the default implementation of the JavaScript `window.open()` method.*

Example: Extending the HTMLHost class

Adobe AIR 1.0 and later

The following example shows how to customize the way that an HTMLLoader object affects the user interface, by extending the HTMLHost class:

Flex example:

- 1 Create a class that extends the HTMLHost class (a subclass).
- 2 Override methods of the new class to handle changes in the user interface-related settings. For example, the following class, `CustomHost`, defines behaviors for calls to `window.open()` and changes to `window.document.title`. Calls to `window.open()` open the HTML page in a new window, and changes to `window.document.title` (including the setting of the `<title>` element of an HTML page) set the title of that window.

```
package
{
    import flash.html.*;
    import flash.display.StageScaleMode;
    import flash.display.NativeWindow;
    import flash.display.NativeWindowInitOptions;

    public class CustomHost extends HTMLHost
    {
        import flash.html.*;
        override public function
            createWindow(windowCreateOptions:HTMLWindowCreateOptions):HTMLLoader
        {
            var initOptions:NativeWindowInitOptions = new NativeWindowInitOptions();
            var bounds:Rectangle = new Rectangle(windowCreateOptions.x,
                windowCreateOptions.y,
                windowCreateOptions.width,
                windowCreateOptions.height);
            var htmlControl:HTMLLoader = HTMLLoader.createRootWindow(true, initOptions,
                windowCreateOptions.scrollBarsVisible, bounds);
            htmlControl.htmlHost = new HTMLHostImplementation();
            if(windowCreateOptions.fullscreen){
                htmlControl.stage.displayState =
                    StageDisplayState.FULL_SCREEN_INTERACTIVE;
            }
            return htmlControl;
        }
        override public function updateTitle(title:String):void
        {
            htmlLoader.stage.nativeWindow.title = title;
        }
    }
}
```

- 3 In the code that contains the HTMLLoader (not the code of the new subclass of HTMLHost), create an object of the new class. Assign the new object to the `htmlHost` property of the HTMLLoader. The following Flex code uses the CustomHost class defined in the previous step:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication
  xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="vertical"
  applicationComplete="init()" >
  <mx:Script>
    <![CDATA[
      import flash.html.HTMLLoader;
      import CustomHost;
      private function init():void
      {
        var html:HTMLLoader = new HTMLLoader();
        html.width = container.width;
        html.height = container.height;
        var urlReq:URLRequest = new URLRequest("Test.html");
        html.htmlHost = new CustomHost();
        html.load(urlReq);
        container.addChild(html);
      }
    ]]>
  </mx:Script>
  <mx:UIComponent id="container" width="100%" height="100%"/>
</mx:WindowedApplication>
```

To test the code described here, include an HTML file with the following content in the application directory:

```
<html>
  <head>
    <title>Test</title>
  </head>
  <script>
    function openWindow()
    {
      window.runtime.trace("in");
      document.title = "foo"
      window.open('Test.html');
      window.runtime.trace("out");
    }
  </script>
  <body>
    <a href="#" onclick="openWindow()">window.open('Test.html')</a>
  </body>
</html>
```

Flash Professional example:

- 1 Create a Flash file for AIR. Set its document class to CustomHostExample and then save the file as CustomHostExample.fla.
- 2 Create an ActionScript file called CustomHost.as containing a class that extends the HTMLHost class (a subclass). This class overrides certain methods of the new class to handle changes in the user interface-related settings. For example, the following class, CustomHost, defines behaviors for calls to `window.open()` and changes to `window.document.title`. Calls to the `window.open()` method open the HTML page in a new window, and changes to the `window.document.title` property (including the setting of the `<title>` element of an HTML page) set the title of that window.

```
package
{
    import flash.display.StageScaleMode;
    import flash.display.NativeWindow;
    import flash.display.NativeWindowInitOptions;
    import flash.events.Event;
    import flash.events.NativeWindowBoundsEvent;
    import flash.geom.Rectangle;
    import flash.html.HTMLLoader;
    import flash.html.HTMLHost;
    import flash.html.HTMLWindowCreateOptions;
    import flash.text.TextField;

    public class CustomHost extends HTMLHost
    {
        public var statusField:TextField;

        public function CustomHost(defaultBehaviors:Boolean=true)
        {
            super(defaultBehaviors);
        }
        override public function windowClose():void
        {
            htmlLoader.stage.nativeWindow.close();
        }
        override public function createWindow(
            windowCreateOptions:HTMLWindowCreateOptions ):HTMLLoader
        {
            var initOptions:NativeWindowInitOptions = new NativeWindowInitOptions();
            var bounds:Rectangle = new Rectangle(windowCreateOptions.x,
                windowCreateOptions.y,
                windowCreateOptions.width,
                windowCreateOptions.height);
            var htmlControl:HTMLLoader = HTMLLoader.createRootWindow(true, initOptions,
                windowCreateOptions.scrollBarsVisible, bounds);
            htmlControl.htmlHost = new HTMLHostImplementation();
            if(windowCreateOptions.fullscreen){
                htmlControl.stage.displayState =
                    StageDisplayState.FULL_SCREEN_INTERACTIVE;
            }
            return htmlControl;
        }
        override public function updateLocation(locationURL:String):void
        {
            trace(locationURL);
        }
        override public function set windowRect(value:Rectangle):void
        {
            htmlLoader.stage.nativeWindow.bounds = value;
        }
    }
}
```

```
    }  
    override public function updateStatus(status:String):void  
    {  
        statusField.text = status;  
        trace(status);  
    }  
    override public function updateTitle(title:String):void  
    {  
        htmlLoader.stage.nativeWindow.title = title + "- Example Application";  
    }  
    override public function windowBlur():void  
    {  
        htmlLoader.alpha = 0.5;  
    }  
    override public function windowFocus():void  
    {  
        htmlLoader.alpha = 1;  
    }  
    }  
}
```

- 3 Create another ActionScript file named CustomHostExample.as to contain the document class for the application. This class creates an HTMLLoader object and sets its host property to an instance of the CustomHost class defined in the previous step:

```
package
{
    import flash.display.Sprite;
    import flash.html.HTMLLoader;
    import flash.net.URLRequest;
    import flash.text.TextField;

    public class CustomHostExample extends Sprite
    {
        function CustomHostExample():void
        {
            var html:HTMLLoader = new HTMLLoader();
            html.width = 550;
            html.height = 380;
            var host:CustomHost = new CustomHost();
            html.htmlHost = host;

            var urlReq:URLRequest = new URLRequest("Test.html");
            html.load(urlReq);

            addChild(html);

            var statusTxt:TextField = new TextField();
            statusTxt.y = 380;
            statusTxt.height = 20;
            statusTxt.width = 550;
            statusTxt.background = true;
            statusTxt.backgroundColor = 0xEEEEEEEE;
            addChild(statusTxt);

            host.statusField = statusTxt;
        }
    }
}
```

To test the code described here, include an HTML file with the following content in the application directory:

```
<html>
  <head>
    <title>Test</title>
    <script>
      function openWindow()
      {
        document.title = "Test"
        window.open('Test.html');
      }
    </script>
  </head>
  <body bgColor="#EEEEEE">
    <a href="#" onclick="window.open('Test.html') ">window.open('Test.html') </a>
    <br/><a href="#" onclick="window.document.location='http://www.adobe.com' ">
    window.document.location = 'http://www.adobe.com' </a>
    <br/><a href="#" onclick="window.moveBy(6, 12) ">moveBy(6, 12) </a>
    <br/><a href="#" onclick="window.close() ">window.close() </a>
    <br/><a href="#" onclick="window.blur() ">window.blur() </a>
    <br/><a href="#" onclick="window.focus() ">window.focus() </a>
    <br/><a href="#" onclick="window.status = new Date().toString() ">window.status=new
Date().toString() </a>
  </body>
</html>
```

- 1 Create an ActionScript file, such as `HTMLHostImplementation.as`.
- 2 In this file, define a class extending the `HTMLHost` class.
- 3 Override methods of the new class to handle changes in the user interface-related settings. For example, the following class, `CustomHost`, defines behaviors for calls to `window.open()` and changes to `window.document.title`. Calls to `window.open()` open the HTML page in a new window, and changes to `window.document.title` (including the setting of the `<title>` element of an HTML page) set the title of that window.


```
package {
    import flash.html.HTMLHost;
    import flash.html.HTMLLoader;
    import flash.html.HTMLWindowCreateOptions;
    import flash.geom.Rectangle;
    import flash.display.NativeWindowInitOptions;
    import flash.display.StageDisplayState;

    public class HTMLHostImplementation extends HTMLHost{
        public function HTMLHostImplementation(defaultBehaviors:Boolean = true):void{
            super(defaultBehaviors);
        }

        override public function updateTitle(title:String):void{
            htmlLoader.stage.nativeWindow.title = title + " - New Host";
        }

        override public function
createWindow(windowCreateOptions:HTMLWindowCreateOptions):HTMLLoader{
            var initOptions:NativeWindowInitOptions = new NativeWindowInitOptions();
            var bounds:Rectangle = new Rectangle(windowCreateOptions.x,
                                                windowCreateOptions.y,
                                                windowCreateOptions.width,
                                                windowCreateOptions.height);

            var htmlControl:HTMLLoader = HTMLLoader.createRootWindow(true, initOptions,
                                                                    windowCreateOptions.scrollBarsVisible, bounds);

            htmlControl.htmlHost = new HTMLHostImplementation();

            if(windowCreateOptions.fullscreen){
                htmlControl.stage.displayState =
                    StageDisplayState.FULL_SCREEN_INTERACTIVE;
            }

            return htmlControl;
        }
    }
}
```

- 4 Compile the class into a SWF file using the `acompc` component compiler.

```
acompc -source-path . -include-classes HTMLHostImplementation -output Host.zip
```

Note: The `acompc` compiler is included with the Flex SDK (but not the AIR SDK, which is targeted for HTML developers who do not generally need to compile SWF files.) Instructions for using `acompc` are provided in the [Using `acompc`, the component compiler](#).

- 5 Open the `Host.zip` file and extract the `Library.swf` file inside.
- 6 Rename `Library.swf` to `HTMLHostLibrary.swf`. This SWF file is the library to import into the HTML page.
- 7 Import the library into the HTML page using a `<script>` tag:

```
<script src="HTMLHostLibrary.swf" type="application/x-shockwave-flash"></script>
```

- 8 Assign a new instance of the HTMLHost implementation to the HTMLLoader object of the page.

```
window.htmlLoader.htmlHost = new window.runtime.HTMLHostImplementation();
```

The following HTML page illustrates how to load and use the HTMLHost implementation. You can test the `updateTitle()` and `createWindow()` implementations by clicking the button to open a new, fullscreen window.

```
<html>
  <head>
    <title>HTMLHost Example</title>
    <script src="HTMLHostLibrary.swf" type="application/x-shockwave-flash"></script>
    <script language="javascript">
      window.htmlLoader.htmlHost = new window.runtime.HTMLHostImplementation();

      function test(){
        window.open('child.html', 'Child', 'fullscreen');
      }
    </script>
  </head>
  <body>
    <button onClick="test()">Create Window</button>
  </body>
</html>
```

To run this example, provide an HTML file named `child.html` in the application directory.

Handling changes to the `window.location` property

Adobe AIR 1.0 and later

Override the `locationChange()` method to handle changes of the URL of the HTML page. The `locationChange()` method is called when JavaScript in a page changes the value of `window.location`. The following example simply loads the requested URL:

```
override public function updateLocation(locationURL:String):void
{
    htmlLoader.load(new URLRequest(locationURL));
}
```

Note: You can use the `htmlLoader` property of the `HTMLHost` object to reference the current `HTMLLoader` object.

Handling JavaScript calls to `window.moveBy()`, `window.moveTo()`, `window.resizeTo()`, and `window.resizeBy()`

Adobe AIR 1.0 and later

Override the `set windowRect()` method to handle changes in the bounds of the HTML content. The `set windowRect()` method is called when JavaScript in a page calls `window.moveBy()`, `window.moveTo()`, `window.resizeTo()`, or `window.resizeBy()`. The following example simply updates the bounds of the desktop window:

```
override public function set windowRect(value:Rectangle):void
{
    htmlLoader.stage.nativeWindow.bounds = value;
}
```

Handling JavaScript calls to window.open()

Adobe AIR 1.0 and later

Override the `createWindow()` method to handle JavaScript calls to `window.open()`. Implementations of the `createWindow()` method are responsible for creating and returning a new `HTMLLoader` object. Typically, you would display the `HTMLLoader` in a new window, but creating a window is not required.

The following example illustrates how to implement the `createWindow()` function using the `HTMLLoader.createRootWindow()` to create both the window and the `HTMLLoader` object. You can also create a `NativeWindow` object separately and add the `HTMLLoader` to the window stage.

```

override public function createWindow(windowCreateOptions:HTMLWindowCreateOptions):HTMLLoader{
    var initOptions:NativeWindowInitOptions = new NativeWindowInitOptions();
    var bounds:Rectangle = new Rectangle(windowCreateOptions.x, windowCreateOptions.y,
        windowCreateOptions.width, windowCreateOptions.height);
    var htmlControl:HTMLLoader = HTMLLoader.createRootWindow(true, initOptions,
        windowCreateOptions.scrollBarsVisible, bounds);
    htmlControl.htmlHost = new HTMLHostImplementation();
    if(windowCreateOptions.fullscreen){
        htmlControl.stage.displayState = StageDisplayState.FULL_SCREEN_INTERACTIVE;
    }
    return htmlControl;
}
    
```

Note: This example assigns the custom `HTMLHost` implementation to any new windows created with `window.open()`. You can also use a different implementation or set the `htmlHost` property to null for new windows, if desired.

The object passed as a parameter to the `createWindow()` method is an `HTMLWindowCreateOptions` object. The `HTMLWindowCreateOptions` class includes properties that report the values set in the `features` parameter string in the call to `window.open()`:

HTMLWindowCreateOptions property	Corresponding setting in the features string in the JavaScript call to window.open()
fullscreen	fullscreen
height	height
locationBarVisible	location
menuBarVisible	menubar
resizeable	resizable
scrollBarsVisible	scrollbars
statusBarVisible	status
toolBarVisible	toolbar
width	width
x	left or screenX
y	top or screenY

The `HTMLLoader` class does not implement all the features that can be specified in the feature string. Your application must provide scroll bars, location bars, menu bars, status bars, and toolbars when appropriate.

The other arguments to the JavaScript `window.open()` method are handled by the system. A `createWindow()` implementation should not load content in the `HTMLLoader` object, or set the window title.

Handling JavaScript calls to `window.close()`

Adobe AIR 1.0 and later

Override the `windowClose()` to handle JavaScript calls to `window.close()` method. The following example closes the desktop window when the `window.close()` method is called:

```
override public function windowClose():void
{
    htmlLoader.stage.nativeWindow.close();
}
```

JavaScript calls to `window.close()` do not have to close the containing window. You could, for example, remove the `HTMLLoader` from the display list, leaving the window (which may have other content) open, as in the following code:

```
override public function windowClose():void
{
    htmlLoader.parent.removeChild(htmlLoader);
}
```

Handling changes of the `window.status` property

Adobe AIR 1.0 and later

Override the `updateStatus()` method to handle JavaScript changes to the value of `window.status`. The following example traces the status value:

```
override public function updateStatus(status:String):void
{
    trace(status);
}
```

The requested status is passed as a string to the `updateStatus()` method.

The `HTMLLoader` object does not provide a status bar.

Handling changes of the `window.document.title` property

Adobe AIR 1.0 and later

Override the `updateTitle()` method to handle JavaScript changes to the value of `window.document.title`. The following example changes the window title and appends the string, "Sample," to the title:

```
override public function updateTitle(title:String):void
{
    htmlLoader.stage.nativeWindow.title = title + " - Sample";
}
```

When `document.title` is set on an HTML page, the requested title is passed as a string to the `updateTitle()` method.

Changes to `document.title` do not have to change the title of the window containing the `HTMLLoader` object. You could, for example, change another interface element, such as a text field.

Handling JavaScript calls to `window.blur()` and `window.focus()`

Adobe AIR 1.0 and later

Override the `windowBlur()` and `windowFocus()` methods to handle JavaScript calls to `window.blur()` and `window.focus()`, as shown in the following example:

```
override public function windowBlur():void
{
    htmlLoader.alpha = 0.5;
}
override public function windowFocus():void
{
    htmlLoader.alpha = 1.0;
    NativeApplication.nativeApplication.activate(htmlLoader.stage.nativeWindow);
}
```

Note: AIR does not provide an API for deactivating a window or application.

Creating windows with scrolling HTML content

Adobe AIR 1.0 and later

The `HTMLLoader` class includes a static method, `HTMLLoader.createRootWindow()`, which lets you open a new window (represented by a `NativeWindow` object) that contains an `HTMLLoader` object and define some user interface settings for that window. The method takes four parameters, which let you define the user interface:

Parameter	Description
<code>visible</code>	A Boolean value that specifies whether the window is initially visible (<code>true</code>) or not (<code>false</code>).
<code>windowInitOptions</code>	A <code>NativeWindowInitOptions</code> object. The <code>NativeWindowInitOptions</code> class defines initialization options for a <code>NativeWindow</code> object, including the following: whether the window is minimizable, maximizable, or resizable, whether the window has system chrome or custom chrome, whether the window is transparent or not (for windows that do not use system chrome), and the type of window.
<code>scrollBarsVisible</code>	Whether there are scroll bars (<code>true</code>) or not (<code>false</code>).
<code>bounds</code>	A <code>Rectangle</code> object defining the position and size of the new window.

For example, the following code uses the `HTMLLoader.createRootWindow()` method to create a window with `HTMLLoader` content that uses scroll bars:

```
var initOptions = new air.NativeWindowInitOptions();
var bounds = new air.Rectangle(10, 10, 600, 400);
var html2 = air.HTMLLoader.createRootWindow(true, initOptions, true, bounds);
var urlReq2 = new air.URLRequest("http://www.example.com");
html2.load(urlReq2);
html2.stage.nativeWindow.activate();
```

Note: Windows created by calling `createRootWindow()` directly in JavaScript remain independent from the opening HTML window. The JavaScript window opener and parent properties, for example, are `null`. However, if you call `createRootWindow()` indirectly by overriding the `HTMLHost.createWindow()` method to call `createRootWindow()`, then opener and parent do reference the opening HTML window.

Chapter 5: Working with vectors

Adobe AIR 1.5 and later

A *Vector* instance is a *typed array*, which means that all the elements in a *Vector* instance always have the same data type. Some AIR APIs, such as *NativeProcess* and *NetworkInfo*, use *Vectors* as data types for properties or methods.

In JavaScript code running in Adobe AIR, the *Vector* class is referenced as *air.Vector* (in the *AIRAliases.js* file).

Basics of vectors

Adobe AIR 1.5 and later

When you declare a *Vector* variable or instantiate a *Vector* object, you explicitly specify the data type of the objects that the *Vector* can contain. The specified data type is known as the *Vector's base type*. At run time, any code that sets or retrieves a value of a *Vector* is checked. If the data type of the object being added or retrieved doesn't match the *Vector's* base type, an error occurs.

In addition to the data type restriction, the *Vector* class has other restrictions that distinguish it from the *Array* class:

- A *Vector* is a dense array. An *Array* object may have values in indices 0 and 7 even if it has no values in positions 1 through 6. However, a *Vector* must have a value (or `null`) in each index.
- A *Vector* can optionally be fixed length. This means that the number of elements the *Vector* contains can't change.
- Access to a *Vector's* elements is bounds-checked. You can never read a value from an index greater than the final element (`length - 1`). You can never set a value with an index more than one beyond the current final index. (In other words, you can only set a value at an existing index or at index `[length]`.)

As a result of its restrictions, a *Vector* has three primary benefits over an *Array* instance whose elements are all instances of a single class:

- **Performance:** array element access and iteration are much faster when using a *Vector* instance than when using an *Array* instance.
- **Type safety:** examples of such errors include assigning a value of the incorrect data type to a *Vector* or expecting the wrong data type when reading a value from a *Vector*. At run time, data types are checked when adding data to or reading data from a *Vector* object.
- **Reliability:** run-time range checking (or fixed-length checking) increases reliability significantly over *Arrays*.

Aside from the additional restrictions and benefits, the *Vector* class is very much like the *Array* class. The properties and methods of a *Vector* object are similar—usually identical—to the properties and methods of an *Array*. In most situations where you would use an *Array* in which all the elements have the same data type, a *Vector* instance is preferable.

Important concepts and terms

The following reference list contains important terms to know when programming array and vector handling routines:

Array access (`[]`) operator A pair of square brackets surrounding an index or key that uniquely identifies an array element. This syntax is used after a vector variable name to specify a single element of the vector rather than the entire vector.

Base type The data type of the objects that a Vector instance is allowed to store.

Element A single item in a vector.

Index The numeric “address” used to identify a single element in an indexed array.

T The standard convention that’s used in this documentation to represent the base type of a Vector instance, whatever that base type happens to be. The T convention is used to represent a class name, as shown in the Type parameter description. (“T” stands for “type,” as in “data type.”).

Type parameter The syntax that’s used with the Vector class name to specify the Vector’s base type (the data type of the objects that it stores). The syntax consists of a period (.), then the data type name surrounded by angle brackets (<>). Put together, it looks like this: `vector.<T>`. In this documentation, the class specified in the type parameter is represented generically as T.

Vector A type of array whose elements are all instances of the same data type.

Creating vectors

AIR 1.5 and later

You create a Vector instance by calling the `air.Vector["<T>"]()` constructor. When you call this constructor, you specify the base type of the Vector variable. You specify the Vector’s base type using type parameter syntax. The type parameter immediately follows the word `vector` in the code. It consists of a left bracket, then a string containing the base class name surrounded by angle brackets (<>), followed by a right bracket. This example shows this syntax:

```
var v = new air.Vector["<String>"]();
```

In this example, the variable `v` is declared as a vector of String objects. In other words, it represents an indexed array that can only hold String instances.

If you use the `air.Vector["<T>"]()` constructor without any arguments, it creates an empty Vector instance. You can test that a Vector is empty by checking its `length` property. For example, the following code calls the `Vector["<T>"]()` constructor with no arguments:

```
var names = new air.Vector["<String>"]();  
air.trace(names.length); // output: 0
```

If you know ahead of time how many elements a Vector initially needs, you can pre-define the number of elements in the Vector. To create a Vector with a certain number of elements, pass the number of elements as the first parameter (the `length` parameter). Because Vector elements can’t be empty, the elements are filled with instances of the base type. If the base type is a reference type that allows `null` values, the elements all contain `null`. Otherwise, the elements all contain the default value for the class. For example, a Number variable can’t be `null`. Consequently, in the following code listing the Vector named `ages` is created with three elements, each containing the default Number value `NaN`:

```
var ages = new air.Vector["<Number>"](3);  
air.trace(ages); // output: NaN, NaN, NaN
```

Using the `Vector["<T>"]()` constructor you can also create a fixed-length Vector by passing `true` for the second parameter (the `fixed` parameter). In that case the Vector is created with the specified number of elements and the number of elements can’t be changed. Note, however, that you can still change the values of the elements of a fixed-length Vector.

If you create a Vector of AIR runtime objects (classes defined in the `window.runtime` object), reference the class’s fully qualified ActionScript 3.0 name when calling the Vector constructor. For example, the following code creates a Vector of File objects:

```
var files = new air.Vector["flash.filesystem.File"](3);
```

Inserting elements into a vector

Adobe AIR 1.5 and later

The most basic way to add an element to vector is to use the array access (`[]`) operator:

```
songTitles[5] = "Happy Birthday";
```

If the Vector doesn't already have an element at that index, the index is created and the value is stored there.

With a Vector object, you can only assign a value to an existing index or to the next available index. The next available index corresponds to the Vector object's `length` property. The safest way to add a new element to a Vector object is to use code like this listing:

```
myVector[myVector.length] = valueToAdd;
```

As with arrays, three of the Vector class methods—`push()`, `unshift()`, and `splice()`—allow you to insert elements into a vector.

Note: If a Vector object's `fixed` property is `true`, the total number of elements in the Vector can't change. If you try to add a new element to a fixed-length Vector using the `push()` method or other means, an error occurs.

Retrieving values and removing vector elements

Adobe AIR 1.5 and later

The simplest way to retrieve the value of an element from vector is to use the array access (`[]`) operator. To retrieve the value of an vector element, use the vector object name and index number on the right side of an assignment statement:

```
var myFavoriteSong = songTitles[3];
```

It's possible to attempt to retrieve a value from a vector using an index where no element exists. In that case, a Vector throws a `RangeError` exception.

Three methods of the Array and Vector classes—`pop()`, `shift()`, and `splice()`—allow you to remove elements.

```
var vegetables = new air.Vector["<String>"];  
vegetables.push("spinach");  
vegetables.push("green pepper");  
vegetables.push("cilantro");  
vegetables.push("onion");  
var spliced = vegetables.splice(2, 2);  
air.trace(spliced); // output: spinach,green pepper
```

You can truncate a vector using the `length` property.

If you set the `length` property of a vector to a length that is less than the current length of the vector, the vector is truncated. Any elements stored at index numbers higher than the new value of `length` minus 1 are removed.

Note: If a Vector object's `fixed` property is `true`, the total number of elements in the Vector can't change. If you try to remove an element from or truncate a fixed-length Vector using the techniques described here, an error occurs.

Properties and methods of Vector objects

Adobe AIR 1.5 and later

Many of the same methods and properties of Array objects are available for vector object. For example, you can call the `reverse()` method to change the order of elements of a Vector. You can call the `sort()` method to sort the elements of a Vector. However, the Vector class does not include a `sortOn()` method.

For details on supported properties and methods, see the Vector class documentation in the Adobe AIR Language Reference for HTML Developers.

Example: Using AIR APIs that require vectors

Adobe AIR 1.5 and later

Some Adobe AIR runtime classes use vectors as properties or method return values. For example, the `findInterfaces()` method of the `NetworkInfo` class returns an array of `NetworkInterface` objects. The `arguments` property `NativeProcessStartupInfo` class is a vector of strings.

Accessing AIR APIs that return vector objects

Adobe AIR 2.0 and later

The `findInterfaces()` method of the `NetworkInfo` class returns an array of `NetworkInterface` objects. For example, the following code lists the computer's network interfaces:

```
var netInfo = air.NetworkInfo;
var interfaces = netInfo.findInterfaces();
for (i = 0; i < interfaces.length; i++)
{
    air.trace(interfaces[i].name);
    air.trace(" hardware address: ", interface.hardwareAddress);
}
```

You iterate through the vector of `NetworkInfo` objects just as you would iterate through an array. You use a `for` loop and use square brackets to access indexed elements of the vector.

The `interfaces` property of the `NetworkInterface` object is a vector of `InterfaceAddress` objects. The following code extends the previous example, adding a function to enumerate interface addresses for each network interface:

```
var netInfo = air.NetworkInfo;
var interfaces = netInfo.findInterfaces();
for (i = 0; i < interfaces.length; i++)
{
    air.trace(interfaces[i].name);
    air.trace(" hardware address: ", interface.hardwareAddress);
    air.trace(" addresses: ", traceAddresses(i));
}

function traceAddresses(i)
{
    returnString = new String();
    for (j = 0; j < interfaces[i].addresses.length; j++)
        returnString += interfaces[i].addresses[j].address + " ";
}
}
```

Setting AIR APIs that are vectors

Adobe AIR 2.0 and later

The `arguments` property `NativeProcessStartupInfo` class is a vector of strings. To set this property, create a vector of strings using the `air.Vector()` constructor. You can use the `push()` method to add strings to the vector:

```
var arguments = new air.Vector["<String>"]();

arguments.push("test");
arguments.push("44");

var startupInfo = new air.NativeProcessStartupInfo();
startupInfo.arguments = arguments;
startupInfo.executable = File.applicationDirectory.resolvePath("myApplication.exe");

process = new air.NativeProcess();
process.start(startupInfo);
```

For more information on using the native process API, see “Communicating with Native Processes” in *Networking and communication*.

Chapter 6: AIR security

Adobe AIR 1.0 and later

AIR security basics

Adobe AIR 1.0 and later

AIR applications run with the same security restrictions as native applications. In general, AIR applications, like native applications, have broad access to operating system capabilities such as reading and writing files, starting applications, drawing to the screen, and communicating with the network. Operating system restrictions that apply to native applications, such as user-specific privileges, equally apply to AIR applications.

Although the Adobe® AIR® security model is an evolution of the Adobe® Flash® Player security model, the security contract is different from the security contract applied to content in a browser. This contract offers developers a secure means of broader functionality for rich experiences with freedoms that would be inappropriate for a browser-based application.

AIR applications are written using either compiled bytecode (SWF content) or interpreted script (JavaScript, HTML) so that the runtime provides memory management. This minimizes the chances of AIR applications being affected by vulnerabilities related to memory management, such as buffer overflows and memory corruption. These are some of the most common vulnerabilities affecting desktop applications written in native code.

Installation and updates

Adobe AIR 1.0 and later

AIR applications are distributed via AIR installer files which use the `air` extension or via native installers, which use the file format and extension of the native platform. For example, the native installer format of Windows is an EXE file, and for Android the native format is an APK file.

When Adobe AIR is installed and an AIR installer file is opened, the AIR runtime administers the installation process. When a native installer is used, the operating system administers the installation process.

Note: Developers can specify a version, and application name, and a publisher source, but the initial application installation workflow itself cannot be modified. This restriction is advantageous for users because all AIR applications share a secure, streamlined, and consistent installation procedure administered by the runtime. If application customization is necessary, it can be provided when the application is first executed.

Runtime installation location

Adobe AIR 1.0 and later

AIR applications first require the runtime to be installed on a user's computer, just as SWF files first require the Flash Player browser plug-in to be installed.

The runtime is installed to the following location on desktop computers:

- Mac OS: `/Library/Frameworks/`
- Windows: `C:\Program Files\Common Files\Adobe AIR`
- Linux: `/opt/Adobe AIR/`

On Mac OS, to install an updated version of an application, the user must have adequate system privileges to install to the application directory. On Windows and Linux, a user must have administrative privileges.

Note: On iOS, the AIR runtime is not installed separately; every AIR app is a self-contained application.

The runtime can be installed in two ways: using the seamless install feature (installing directly from a web browser) or via a manual install.

Seamless install (runtime and application)

Adobe AIR 1.0 and later

The seamless install feature provides developers with a streamlined installation experience for users who do not have Adobe AIR installed yet. In the seamless install method, the developer creates a SWF file that presents the application for installation. When a user clicks in the SWF file to install the application, the SWF file attempts to detect the runtime. If the runtime cannot be detected it is installed, and the runtime is activated immediately with the installation process for the developer's application.

Manual install

Adobe AIR 1.0 and later

Alternatively, the user can manually download and install the runtime before opening an AIR file. The developer can then distribute an AIR file by different means (for instance, via e-mail or an HTML link on a website). When the AIR file is opened, the runtime begins to process the application installation.

Application installation flow

Adobe AIR 1.0 and later

The AIR security model allows users to decide whether to install an AIR application. The AIR install experience provides several improvements over native application install technologies that make this trust decision easier for users:

- The runtime provides a consistent installation experience on all operating systems, even when an AIR application is installed from a link in a web browser. Most native application install experiences depend upon the browser or other application to provide security information, if it is provided at all.
- The AIR application install experience identifies the source of the application and information about what privileges are available to the application (if the user allows the installation to proceed).
- The runtime administers the installation process of an AIR application. An AIR application cannot manipulate the installation process the runtime uses.

In general, users should not install any desktop application that comes from a source that they do not trust, or that cannot be verified. The burden of proof on security for native applications is equally true for AIR applications as it is for other installable applications.

Application destination

Adobe AIR 1.0 and later

The installation directory can be set using one of the following two options:

- 1 The user customizes the destination during installation. The application installs to wherever the user specifies.
- 2 If the user does not change the install destination, the application installs to the default path as determined by the runtime:
 - Mac OS: `~/Applications/`
 - Windows XP and earlier: `C:\Program Files\`
 - Windows Vista: `~/Apps/`
 - Linux: `/opt/`

If the developer specifies an `installFolder` setting in the application descriptor file, the application is installed to a subpath of this directory.

The AIR file system

Adobe AIR 1.0 and later

The install process for AIR applications copies all files that the developer has included within the AIR installer file onto the user's local computer. The installed application is composed of:

- Windows: A directory containing all files included in the AIR installer file. The runtime also creates an exe file during the installation of the AIR application.
- Linux: A directory containing all files included in the AIR installer file. The runtime also creates a bin file during the installation of the AIR application.
- Mac OS: An `app` file that contains all of the contents of the AIR installer file. It can be inspected using the "Show Package Contents" option in Finder. The runtime creates this app file as part of the installation of the AIR application.

An AIR application is run by:

- Windows: Running the `.exe` file in the install folder, or a shortcut that corresponds to this file (such as a shortcut on the Start Menu or desktop).
- Linux: Launching the `.bin` file in the install folder, choosing the application from the Applications menu, or running from an alias or desktop shortcut.
- Mac OS: Running the `.app` file or an alias that points to it.

The application file system also includes subdirectories related to the function of the application. For example, information written to encrypted local storage is saved to a subdirectory in a directory named after the application identifier of the application.

AIR application storage

Adobe AIR 1.0 and later

AIR applications have privileges to write to any location on the user's hard drive; however, developers are encouraged to use the `app-storage:/` path for local storage related to their application. Files written to `app-storage:/` from an application are located in a standard location depending on the user's operating system:

- On Mac OS: the storage directory of an application is `<appData>/<appId>/Local Store/` where `<appData>` is the user's "preferences folder," typically: `/Users/<user>/Library/Preferences`
- On Windows: the storage directory of an application is `<appData>\<appId>\Local Store\` where `<appData>` is the user's CSIDL_APPDATA "Special Folder," typically: `C:\Documents and Settings\<user>\Application Data`
- On Linux: `<appData>/<appId>/Local Store/` where `<appData>` is `/home/<user>/ .appdata`

You can access the application storage directory via the `air.File.applicationStorageDirectory` property. You can access its contents using the `resolvePath()` method of the `File` class. For details, see ["Working with the file system"](#) on page 145.

Updating Adobe AIR

Adobe AIR 1.0 and later

When the user installs an AIR application that requires an updated version of the runtime, the runtime automatically installs the required runtime update.

To update the runtime, a user must have administrative privileges for the computer.

Updating AIR applications

Adobe AIR 1.0 and later

Development and deployment of software updates are one of the biggest security challenges facing native code applications. The AIR API provides a mechanism to improve this: the `Updater.update()` method can be invoked upon launch to check a remote location for an AIR file. If an update is appropriate, the AIR file is downloaded, installed, and the application restarts. Developers can use this class not only to provide new functionality but also respond to potential security vulnerabilities.

The `Updater` class can only be used to update applications distributed as AIR files. Applications distributed as native applications must use the update facilities, if any, of the native operating system.

Note: Developers can specify the version of an application by setting the `versionNumber` property of the application descriptor file.

Uninstalling an AIR application

Adobe AIR 1.0 and later

Removing an AIR application removes all files in the application directory. However, it does not remove all files that the application may have written to outside of the application directory. Removing AIR applications does not revert changes the AIR application has made to files outside of the application directory.

Windows registry settings for administrators

Adobe AIR 1.0 and later

On Windows, administrators can configure a machine to prevent (or allow) AIR application installation and runtime updates. These settings are contained in the Windows registry under the following key: HKLM\Software\Policies\Adobe\AIR. They include the following:

Registry setting	Description
AppInstallDisabled	Specifies that AIR application installation and uninstallation are allowed. Set to 0 for "allowed," set to 1 for "disallowed."
UntrustedAppInstallDisabled	Specifies that installation of untrusted AIR applications (applications that do not include a trusted certificate) is allowed. Set to 0 for "allowed," set to 1 for "disallowed."
UpdateDisabled	Specifies that updating the runtime is allowed, either as a background task or as part of an explicit installation. Set to 0 for "allowed," set to 1 for "disallowed."

HTML security in Adobe AIR

Adobe AIR 1.0 and later

This topic describes the AIR HTML security architecture and how to use iframes, frames, and the sandbox bridge to set up HTML-based applications and safely integrate HTML content into SWF-based applications.

The runtime enforces rules and provides mechanisms for overcoming possible security vulnerabilities in HTML and JavaScript. The same rules are enforced whether your application is primarily written in JavaScript or whether you load the HTML and JavaScript content into a SWF-based application. Content in the application sandbox and the non-application security sandbox have different privileges. When loading content into an iframe or frame, the runtime provides a secure *sandbox bridge* mechanism that allows content in the frame or iframe to communicate securely with content in the application security sandbox.

The AIR SDK provides three classes for rendering HTML content.

The HTMLLoader class provides close integration between JavaScript code and the AIR APIs.

The StageWebView class is an HTML rendering class and has very limited integration with the host AIR application. Content loaded by the StageWebView class is never placed in the application security sandbox and cannot access data or call functions in the host AIR application. On desktop platforms, the StageWebView class uses the built-in AIR HTML engine, based on Webkit, which is also used by the HTMLLoader class. On mobile platforms, the StageWebView class uses the HTML control provided by the operating system. Thus, on mobile platforms the StageWebView class has the same security considerations and vulnerabilities as the system web browser.

The TextField class can display strings of HTML text. No JavaScript can be executed, but the text can include links and externally loaded images.

For more information, see "[Avoiding security-related JavaScript errors](#)" on page 22.

Overview on configuring your HTML-based application

Adobe AIR 1.0 and later

Frames and iframes provide a convenient structure for organizing HTML content in AIR. Frames provide a means both for maintaining data persistence and for working securely with remote content.

Because HTML in AIR retains its normal, page-based organization, the HTML environment completely refreshes if the top frame of your HTML content “navigates” to a different page. You can use frames and iframes to maintain data persistence in AIR, much the same as you would for a web application running in a browser. Define your main application objects in the top frame and they persist as long as you don't allow the frame to navigate to a new page. Use child frames or iframes to load and display the transient parts of the application. (There are a variety of ways to maintain data persistence that can be used in addition to, or instead of, frames. These include cookies, local shared objects, local file storage, the encrypted file store, and local database storage.)

Because HTML in AIR retains its normal, blurred line between executable code and data, AIR puts content in the top frame of the HTML environment into the application sandbox. After the `page load` event, AIR restricts any operations, such as `eval()`, that can convert a string of text into an executable object. This restriction is enforced even when an application does not load remote content. To allow HTML content to execute these restricted operations, you must use frames or iframes to place the content into a non-application sandbox. (Running content in a sandboxed child frame may be necessary when using some JavaScript application frameworks that rely on the `eval()` function.) For a complete list of the restrictions on JavaScript in the application sandbox, see [“Code restrictions for content in different sandboxes”](#) on page 76.

Because HTML in AIR retains its ability to load remote, possibly insecure content, AIR enforces a same-origin policy that prevents content in one domain from interacting with content in another. To allow interaction between application content and content in another domain, you can set up a bridge to serve as the interface between a parent and a child frame.

Setting up a parent-child sandbox relationship

Adobe AIR 1.0 and later

AIR adds the `sandboxRoot` and `documentRoot` attributes to the HTML frame and iframe elements. These attributes let you treat application content as if it came from another domain:

Attribute	Description
<code>sandboxRoot</code>	The URL to use for determining the sandbox and domain in which to place the frame content. The <code>file:</code> , <code>http:</code> , or <code>https:</code> URL schemes must be used.
<code>documentRoot</code>	The URL from which to load the frame content. The <code>file:</code> , <code>app:</code> , or <code>app-storage:</code> URL schemes must be used.

The following example maps content installed in the `sandbox` subdirectory of the application to run in the remote sandbox and the `www.example.com` domain:

```
<iframe
  src="ui.html"
  sandboxRoot="http://www.example.com/local/"
  documentRoot="app:/sandbox/" >
</iframe>
```


Setting up a bridge between parent and child frames in different sandboxes or domains

Adobe AIR 1.0 and later

AIR adds the `childSandboxBridge` and `parentSandboxBridge` properties to the `window` object of any child frame. These properties let you define bridges to serve as interfaces between a parent and a child frame. Each bridge goes in one direction:

`childSandboxBridge` — The `childSandboxBridge` property allows the child frame to expose an interface to content in the parent frame. To expose an interface, you set the `childSandbox` property to a function or object in the child frame. You can then access the object or function from content in the parent frame. The following example shows how a script running in a child frame can expose an object containing a function and a property to its parent:

```
var interface = {};  
interface.calculatePrice = function(){  
    return .45 + 1.20;  
}  
interface.storeID = "abc"  
window.childSandboxBridge = interface;
```

If this child content is in an `iframe` assigned an `id` of "child", you can access the interface from parent content by reading the `childSandboxBridge` property of the frame:

```
var childInterface = document.getElementById("child").childSandboxBridge;  
air.trace(childInterface.calculatePrice()); //traces "1.65"  
air.trace(childInterface.storeID); //traces "abc"
```

`parentSandboxBridge` — The `parentSandboxBridge` property allows the parent frame to expose an interface to content in the child frame. To expose an interface, you set the `parentSandbox` property of the child frame to a function or object in the parent frame. You can then access the object or function from content in the child frame. The following example shows how a script running in the parent frame can expose an object containing a save function to a child:

```
var interface = {};  
interface.save = function(text){  
    var saveFile = air.File("app-storage:/save.txt");  
    //write text to file  
}  
document.getElementById("child").parentSandboxBridge = interface;
```

Using this interface, content in the child frame could save text to a file named `save.txt`. However, it would not have any other access to the file system. In general, application content should expose the narrowest possible interface to other sandboxes. The child content could call the save function as follows:

```
var textToSave = "A string.";  
window.parentSandboxBridge.save(textToSave);
```

If child content attempts to set a property of the `parentSandboxBridge` object, the runtime throws a `SecurityError` exception. If parent content attempts to set a property of the `childSandboxBridge` object, the runtime throws a `SecurityError` exception.

Code restrictions for content in different sandboxes

Adobe AIR 1.0 and later

As discussed in the introduction to this topic, “[HTML security in Adobe AIR](#)” on page 73, the runtime enforces rules and provides mechanisms for overcoming possible security vulnerabilities in HTML and JavaScript. This topic lists those restrictions. If code attempts to call these restricted APIs, the runtime throws an error with the message “Adobe AIR runtime security violation for JavaScript code in the application security sandbox.”

For more information, see “[Avoiding security-related JavaScript errors](#)” on page 22.

Restrictions on using the JavaScript eval() function and similar techniques

Adobe AIR 1.0 and later

For HTML content in the application security sandbox, there are limitations on using APIs that can dynamically transform strings into executable code after the code is loaded (after the `onload` event of the `body` element has been dispatched and the `onload` handler function has finished executing). This is to prevent the application from inadvertently injecting (and executing) code from non-application sources (such as potentially insecure network domains).

For example, if your application uses string data from a remote source to write to the `innerHTML` property of a DOM element, the string could include executable (JavaScript) code that could perform insecure operations. However, while the content is loading, there is no risk of inserting remote strings into the DOM.

One restriction is in the use of the JavaScript `eval()` function. Once code in the application sandbox is loaded and after processing of the `onload` event handler, you can only use the `eval()` function in limited ways. The following rules apply to the use of the `eval()` function *after* code is loaded from the application security sandbox:

- Expressions involving literals are allowed. For example:

```
eval("null");  
eval("3 + .14");  
eval("'foo'");
```

- Object literals are allowed, as in the following:

```
{ prop1: val1, prop2: val2 }
```

- Object literal setter/getters are *prohibited*, as in the following:

```
{ get prop1() { ... }, set prop1(v) { ... } }
```

- Array literals are allowed, as in the following:

```
[ val1, val2, val3 ]
```

- Expressions involving property reads are *prohibited*, as in the following:

```
a.b.c
```

- Function invocation is *prohibited*.
- Function definitions are *prohibited*.
- Setting any property is *prohibited*.
- Function literals are *prohibited*.

However, while the code is loading, before the `onload` event, and during execution the `onload` event handler function, these restrictions do not apply to content in the application security sandbox.

For example, after code is loaded, the following code results in the runtime throwing an exception:

```
eval("alert(44)");  
eval("myFunction(44)");  
eval("NativeApplication.applicationID");
```

Dynamically generated code, such as that which is made when calling the `eval()` function, would pose a security risk if allowed within the application sandbox. For example, an application may inadvertently execute a string loaded from a network domain, and that string may contain malicious code. For example, this could be code to delete or alter files on the user's computer. Or it could be code that reports back the contents of a local file to an untrusted network domain.

Ways to generate dynamic code are the following:

- Calling the `eval()` function.
- Using `innerHTML` properties or DOM functions to insert script tags that load a script outside of the application directory.
- Using `innerHTML` properties or DOM functions to insert script tags that have inline code (rather than loading a script via the `src` attribute).
- Setting the `src` attribute for a script tags to load a JavaScript file that is outside of the application directory.
- Using the `javascript` URL scheme (as in `href="javascript:alert('Test')"`).
- Using the `setInterval()` or `setTimeout()` function where the first parameter (defining the function to run asynchronously) is a string (to be evaluated) rather than a function name (as in `setTimeout('x = 4', 1000)`).
- Calling `document.write()` or `document.writeln()`.

Code in the application security sandbox can only use these methods while content is loading.

These restrictions do *not* prevent using `eval()` with JSON object literals. This lets your application content work with the JSON JavaScript library. However, you are restricted from using overloaded JSON code (with event handlers).

For other Ajax frameworks and JavaScript code libraries, check to see if the code in the framework or library works within these restrictions on dynamically generated code. If they do not, include any content that uses the framework or library in a non-application security sandbox. For details, see Restrictions for JavaScript inside AIR and “[Scripting between application and non-application content](#)” on page 81. Adobe maintains a list of Ajax frameworks known to support the application security sandbox, at <http://www.adobe.com/products/air/develop/ajax/features/>.

Unlike content in the application security sandbox, JavaScript content in a non-application security sandbox *can* call the `eval()` function to execute dynamically generated code at any time.

Restrictions on access to AIR APIs (for non-application sandboxes)

Adobe AIR 1.0 and later

JavaScript code in a non-application sandbox does not have access to the `window.runtime` object, and as such this code cannot execute AIR APIs. If content in a non-application security sandbox calls the following code, the application throws a `TypeError` exception:

```
try {  
    window.runtime.flash.system.NativeApplication.nativeApplication.exit();  
}  
catch (e)  
{  
    alert(e);  
}
```

The exception type is `TypeError` (undefined value), because content in the non-application sandbox does not recognize the `window.runtime` object, so it is seen as an undefined value.

You can expose runtime functionality to content in a non-application sandbox by using a script bridge. For details, see and [“Scripting between application and non-application content”](#) on page 81.

Restrictions on using XMLHttpRequest calls

Adobe AIR 1.0 and later

HTML content in the application security sandbox cannot use synchronous XMLHttpRequest methods to load data from outside of the application sandbox while the HTML content is loading and during `onLoad` event.

By default, HTML content in non-application security sandboxes are not allowed to use the JavaScript XMLHttpRequest object to load data from domains other than the domain calling the request. A `frame` or `iframe` tag can include an `allowcrossdomainxhr` attribute. Setting this attribute to any non-null value allows the content in the frame or iframe to use the JavaScript XMLHttpRequest object to load data from domains other than the domain of the code calling the request:

```
<iframe id="UI"
  src="http://example.com/ui.html"
  sandboxRoot="http://example.com/"
  allowcrossDomainxhr="true"
  documentRoot="app:/">
</iframe>
```

For more information, see [“Scripting between content in different domains”](#) on page 79.

Restrictions on loading CSS, frame, iframe, and img elements (for content in non-application sandboxes)

Adobe AIR 1.0 and later

HTML content in remote (network) security sandboxes can only load CSS, `frame`, `iframe`, and `img` content from remote sandboxes (from network URLs).

HTML content in local-with-filesystem, local-with-networking, or local-trusted sandboxes can only load CSS, `frame`, `iframe`, and `img` content from local sandboxes (not from application or remote sandboxes).

Restrictions on calling the JavaScript window.open() method

Adobe AIR 1.0 and later

If a window that is created via a call to the JavaScript `window.open()` method displays content from a non-application security sandbox, the window's title begins with the title of the main (launching) window, followed by a colon character. You cannot use code to move that portion of the title of the window off screen.

Content in non-application security sandboxes can only successfully call the JavaScript `window.open()` method in response to an event triggered by a user mouse or keyboard interaction. This prevents non-application content from creating windows that might be used deceptively (for example, for phishing attacks). Also, the event handler for the mouse or keyboard event cannot set the `window.open()` method to execute after a delay (for example by calling the `setTimeout()` function).

Content in remote (network) sandboxes can only use the `window.open()` method to open content in remote network sandboxes. It cannot use the `window.open()` method to open content from the application or local sandboxes.

Content in the local-with-filesystem, local-with-networking, or local-trusted sandboxes (see Security sandboxes) can only use the `window.open()` method to open content in local sandboxes. It cannot use `window.open()` to open content from the application or remote sandboxes.

Errors when calling restricted code

Adobe AIR 1.0 and later

If you call code that is restricted from use in a sandbox due to these security restrictions, the runtime dispatches a JavaScript error: "Adobe AIR runtime security violation for JavaScript code in the application security sandbox."

For more information, see "[Avoiding security-related JavaScript errors](#)" on page 22.

Sandbox protection when loading HTML content from a string

Adobe AIR 1.0 and later

The `loadString()` method of the `HTMLLoader` class lets you create HTML content at run time. However, data that you use as the HTML content can be corrupted if the data is loaded from an insecure Internet source. For this reason, by default, HTML created using the `loadString()` method is not placed in the application sandbox and it has no access to AIR APIs. However, you can set the `placeLoadStringContentInApplicationSandbox` property of an `HTMLLoader` object to `true` to place HTML created using the `loadString()` method into the application sandbox. For more information, see [Loading HTML content from a string](#).

Scripting between content in different domains

Adobe AIR 1.0 and later

AIR applications are granted special privileges when they are installed. It is crucial that the same privileges not be leaked to other content, including remote files and local files that are not part of the application.

About the AIR sandbox bridge

Adobe AIR 1.0 and later

Normally, content from other domains cannot call scripts in other domains.

There are still cases where the main AIR application requires content from a remote domain to have controlled access to scripts in the main AIR application, or vice versa. To accomplish this, the runtime provides a *sandbox bridge* mechanism, which serves as a gateway between the two sandboxes. A sandbox bridge can provide explicit interaction between remote and application security sandboxes.

The sandbox bridge exposes two objects that both loaded and loading scripts can access:

- The `parentSandboxBridge` object lets loading content expose properties and functions to scripts in the loaded content.
- The `childSandboxBridge` object lets loaded content expose properties and function to scripts in the loading content.

Objects exposed via the sandbox bridge are passed by value, not by reference. All data is serialized. This means that the objects exposed by one side of the bridge cannot be set by the other side, and that objects exposed are all untyped. Also, you can only expose simple objects and functions; you cannot expose complex objects.

If child content attempts to set a property of the `parentSandboxBridge` object, the runtime throws a `SecurityError` exception. Similarly, if parent content attempts to set a property of the `childSandboxBridge` object, the runtime throws a `SecurityError` exception.

Sandbox bridge example (HTML)

Adobe AIR 1.0 and later

In HTML content, the `parentSandboxBridge` and `childSandboxBridge` properties are added to the JavaScript window object of a child document. For an example of how to set up bridge functions in HTML content, see [“Setting up a sandbox bridge interface”](#) on page 36.

Limiting API exposure

Adobe AIR 1.0 and later

When exposing sandbox bridges, it's important to expose high-level APIs that limit the degree to which they can be abused. Keep in mind that the content calling your bridge implementation may be compromised (for example, via a code injection). So, for example, exposing a `readFile(path)` method (that reads the contents of an arbitrary file) via a bridge is vulnerable to abuse. It would be better to expose a `readApplicationSetting()` API that doesn't take a path and reads a specific file. The more semantic approach limits the damage that an application can do once part of it is compromised.

More Help topics

[“Cross-scripting content in different security sandboxes”](#) on page 34

Writing to disk

Adobe AIR 1.0 and later

Applications running in a web browser have only limited interaction with the user's local file system. Web browsers implement security policies that ensure that a user's computer cannot be compromised as a result of loading web content. For example, SWF files running through Flash Player in a browser cannot directly interact with files already on a user's computer. Shared objects and cookies can be written to a user's computer for the purpose of maintaining user preferences and other data, but this is the limit of file system interaction. Because AIR applications are natively installed, they have a different security contract, one which includes the capability to read and write across the local file system.

This freedom comes with high responsibility for developers. Accidental application insecurities jeopardize not only the functionality of the application, but also the integrity of the user's computer. For this reason, developers should read [“Best security practices for developers”](#) on page 82.

AIR developers can access and write files to the local file system using several URL scheme conventions:

URL scheme	Description
app:/	An alias to the application directory. Files accessed from this path are assigned the application sandbox and have the full privileges granted by the runtime.
app-storage:/	An alias to the local storage directory, standardized by the runtime. Files accessed from this path are assigned a non-application sandbox.
file:///	An alias that represents the root of the user's hard disk. A file accessed from this path is assigned an application sandbox if the file exists in the application directory, and a non-application sandbox otherwise.

Note: AIR applications cannot modify content using the `app:` URL scheme. Also, the application directory may be read only because of administrator settings.

Unless there are administrator restrictions to the user's computer, AIR applications are privileged to write to any location on the user's hard drive. Developers are advised to use the `app-storage:/` path for local storage related to their application. Files written to `app-storage:/` from an application are put in a standard location:

- On Mac OS: the storage directory of an application is `<appData>/<appId>/Local Store/` where `<appData>` is the user's preferences folder. This is typically `/Users/<user>/Library/Preferences`
- On Windows: the storage directory of an application is `<appData>\<appId>\Local Store\` where `<appData>` is the user's CSIDL_APPDATA Special Folder. This is typically `C:\Documents and Settings\<userName>\Application Data`
- On Linux: `<appData>/<appId>/Local Store/` where `<appData>` is `/home/<user>/ .appdata`

If an application is designed to interact with existing files in the user's file system, be sure to read “[Best security practices for developers](#)” on page 82.

Working securely with untrusted content

Adobe AIR 1.0 and later

Content not assigned to the application sandbox can provide additional scripting functionality to your application, but only if it meets the security criteria of the runtime. This topic explains the AIR security contract with non-application content.

Scripting between application and non-application content

Adobe AIR 1.0 and later

AIR applications that script between application and non-application content have more complex security arrangements. Files that are not in the application sandbox are only allowed to access the properties and methods of files in the application sandbox through the use of a sandbox bridge. A sandbox bridge acts as a gateway between application content and non-application content, providing explicit interaction between the two files. When used correctly, sandbox bridges provide an extra layer of security, restricting non-application content from accessing object references that are part of application content.

The benefit of sandbox bridges is best illustrated through example. Suppose an AIR music store application wants to provide an API to advertisers who want to create their own SWF files, with which the store application can then communicate. The store wants to provide advertisers with methods to look up artists and CDs from the store, but also wants to isolate some methods and properties from the third-party SWF file for security reasons.

A sandbox bridge can provide this functionality. By default, content loaded externally into an AIR application at runtime does not have access to any methods or properties in the main application. With a custom sandbox bridge implementation, a developer can provide services to the remote content without exposing these methods or properties. Consider the sandbox bridge as a pathway between trusted and untrusted content, providing communication between loader and loadee content without exposing object references.

For more information on how to securely use sandbox bridges, see [“Scripting between content in different domains”](#) on page 79.

Best security practices for developers

Adobe AIR 1.0 and later

Although AIR applications are built using web technologies, it is important for developers to note that they are not working within the browser security sandbox. This means that it is possible to build AIR applications that can do harm to the local system, either intentionally or unintentionally. AIR attempts to minimize this risk, but there are still ways where vulnerabilities can be introduced. This topic covers important potential insecurities.

Risk from importing files into the application security sandbox

Adobe AIR 1.0 and later

Files that exist in the application directory are assigned to the application sandbox and have the full privileges of the runtime. Applications that write to the local file system are advised to write to `app-storage:/`. This directory exists separately from the application files on the user's computer, hence the files are not assigned to the application sandbox and present a reduced security risk. Developers are advised to consider the following:

- Include a file in an AIR file (in the installed application) only if it is necessary.
- Include a scripting file in an AIR file (in the installed application) only if its behavior is fully understood and trusted.
- Do not write to or modify content in the application directory. The runtime prevents applications from writing or modifying files and directories using the `app:/` URL scheme by throwing a `SecurityError` exception.
- Do not use data from a network source as parameters to methods of the AIR API that may lead to code execution. This includes use of the `Loader.loadBytes()` method and the JavaScript `eval()` function.

Risk from using an external source to determine paths

Adobe AIR 1.0 and later

An AIR application can be compromised when using external data or content. For this reason, take special care when using data from the network or file system. The onus of trust is ultimately on the developer and the network connections they make, but loading foreign data is inherently risky, and should not be used for input into sensitive operations. Developers are advised against the following:

- Using data from a network source to determine a file name
- Using data from a network source to construct a URL that the application uses to send private information

Risk from using, storing, or transmitting insecure credentials

Adobe AIR 1.0 and later

Storing user credentials on the user's local file system inherently introduces the risk that these credentials may be compromised. Developers are advised to consider the following:

- If credentials must be stored locally, encrypt the credentials when writing to the local file system. The runtime provides an encrypted storage unique to each installed application, via the `EncryptedLocalStore` class. For details, see [“Encrypted local storage”](#) on page 256.
- Do not transmit unencrypted user credentials to a network source unless that source is trusted and the transmission uses the HTTPS: or Transport Layer Security (TLS) protocols.
- Never specify a default password in credential creation — let users create their own. Users who leave the default unchanged expose their credentials to an attacker who already knows the default password.

Risk from a downgrade attack

Adobe AIR 1.0 and later

During application install, the runtime checks to ensure that a version of the application is not currently installed. If an application is already installed, the runtime compares the version string against the version that is being installed. If this string is different, the user can choose to upgrade their installation. The runtime does not guarantee that the newly installed version is newer than the older version, only that it is different. An attacker can distribute an older version to the user to circumvent a security weakness. For this reason, the developer is advised to make version checks when the application is run. It is a good idea to have applications check the network for required updates. That way, even if an attacker gets the user to run an old version, that old version will recognize that it needs to be updated. Also, using a clear versioning scheme for your application makes it more difficult to trick users into installing a downgraded version.

Code signing

Adobe AIR 1.0 and later

All AIR installer files are required to be code signed. Code signing is a cryptographic process of confirming that the specified origin of software is accurate. AIR applications can be signed using either by a certificate issued by an external certificate authority (CA) or by a self-signed certificate you create yourself. A commercial certificate from a well-known CA is strongly recommended and provides assurance to your users that they are installing your application, not a forgery. However, self-signed certificates can be created using `adt` from the SDK or using either Flash, Flash Builder, or another application that uses `adt` for certificate generation. Self-signed certificates do not provide any assurance that the application being installed is genuine and should only be used for testing an application prior to public release.

Chapter 7: Working with AIR native windows

Adobe AIR 1.0 and later

You use the classes provided by the Adobe® AIR® native window API to create and manage desktop windows.

Basics of native windows in AIR

Adobe AIR 1.0 and later

For quick explanations and code examples of working with native windows in AIR, see the following quick start articles on the Adobe Developer Connection:

- [Customizing the look and feel of a window](#)

AIR provides an easy-to-use, cross-platform window API for creating native operating system windows using Flash®, Flex™, and HTML programming techniques.

With AIR, you have a wide latitude in developing the appearance of your application. The windows you create can look like a standard desktop application, matching Apple style when run on the Mac, conforming to Microsoft conventions when run on Windows, and harmonizing with the window manager on Linux—all without including a line of platform-specific code. Or you can use the skinnable, extensible chrome provided by the Flex framework to establish your own style no matter where your application is run. You can even draw your own window chrome with vector and bitmap artwork with full support for transparency and alpha blending against the desktop. Tired of rectangular windows? Draw a round one.

Windows in AIR

Adobe AIR 1.0 and later

AIR supports three distinct APIs for working with windows:

- The ActionScript-oriented `NativeWindow` class provides the lowest level window API. Use `NativeWindows` in ActionScript and Flash Professional-authored applications. Consider extending the `NativeWindow` class to specialize the windows used in your application.
- In the HTML environment, you can use the JavaScript `Window` class, just as you would in a browser-based web application. Calls to JavaScript `Window` methods are forwarded to the underlying native window object.
- The Flex framework `mx:WindowedApplication` and `mx:Window` classes provide a Flex “wrapper” for the `NativeWindow` class. The `WindowedApplication` component replaces the `Application` component when you create an AIR application with Flex and must always be used as the initial window in your Flex application.

ActionScript windows

When you create windows with the `NativeWindow` class, use the Flash Player stage and display list directly. To add a visual object to a `NativeWindow`, add the object to the display list of the window stage or to another display object container on the stage.

HTML windows

When you create HTML windows, you use HTML, CSS, and JavaScript to display content. To add a visual object to an HTML window, you add that content to the HTML DOM. HTML windows are a special category of `NativeWindow`. The AIR host defines a `nativeWindow` property in HTML windows that provides access to the underlying `NativeWindow` instance. You can use this property to access the `NativeWindow` properties, methods, and events described here.

***Note:** The JavaScript `Window` object also has methods for scripting the containing window, such as `moveTo()` and `close()`. Where overlapping methods are available, you can use whichever method that is convenient.*

Flex Framework windows

The Flex Framework defines its own window components. These components, `mx:WindowedApplication` and `mx:Window`, cannot be used outside the framework and thus cannot be used in HTML-based AIR applications.

The initial application window

The first window of your application is automatically created for you by AIR. AIR sets the properties and content of the window using the parameters specified in the `initialWindow` element of the application descriptor file.

If the root content is a SWF file, AIR creates a `NativeWindow` instance, loads the SWF file, and adds it to the window stage. If the root content is an HTML file, AIR creates an HTML window and loads the HTML.

Native window classes

Adobe AIR 1.0 and later

The native window API contains the following classes:

Package	Classes
flash.display	<ul style="list-style-type: none">• <code>NativeWindow</code>• <code>NativeWindowInitOptions</code>• <code>NativeWindowDisplayState</code>• <code>NativeWindowResize</code>• <code>NativeWindowSystemChrome</code>• <code>NativeWindowType</code>
flash.events	<ul style="list-style-type: none">• <code>NativeWindowBoundsEvent</code>• <code>NativeWindowDisplayStateEvent</code>

Native window event flow

Adobe AIR 1.0 and later

Native windows dispatch events to notify interested components that an important change is about to occur or has already occurred. Many window-related events are dispatched in pairs. The first event warns that a change is about to happen. The second event announces that the change has been made. You can cancel a warning event, but not a notification event. The following sequence illustrates the flow of events that occurs when a user clicks the maximize button of a window:

- 1 The `NativeWindow` object dispatches a `displayStateChanging` event.
- 2 If no registered listeners cancel the event, the window maximizes.
- 3 The `NativeWindow` object dispatches a `displayStateChange` event.

In addition, the `NativeWindow` object also dispatches events for related changes to the window size and position. The window does not dispatch warning events for these related changes. The related events are:

- a A `move` event is dispatched if the top, left corner of the window moved because of the maximize operation.
- b A `resize` event is dispatched if the window size changed because of the maximize operation.

A `NativeWindow` object dispatches a similar sequence of events when minimizing, restoring, closing, moving, and resizing a window.

The warning events are only dispatched when a change is initiated through window chrome or other operating-system controlled mechanism. When you call a window method to change the window size, position, or display state, the window only dispatches an event to announce the change. You can dispatch a warning event, if desired, using the window `dispatchEvent()` method, then check to see if your warning event has been canceled before proceeding with the change.

For detailed information about the window API classes, methods, properties, and events, see the [Adobe AIR API Reference for HTML Developers](#).

Properties controlling native window style and behavior

Flash Player 9 and later, Adobe AIR 1.0 and later

The following properties control the basic appearance and behavior of a window:

- `type`
- `systemChrome`
- `transparent`
- `owner`

When you create a window, you set these properties on the `NativeWindowInitOptions` object passed to the window constructor. AIR reads the properties for the initial application window from the application descriptor. (Except the `type` property, which cannot be set in the application descriptor and is always set to `normal`.) The properties cannot be changed after window creation.

Some settings of these properties are mutually incompatible: `systemChrome` cannot be set to `standard` when either `transparent` is `true` or `type` is `lightweight`.

Window types

Adobe AIR 1.0 and later

The AIR window types combine chrome and visibility attributes of the native operating system to create three functional types of window. Use the constants defined in the `NativeWindowType` class to reference the type names in code. AIR provides the following window types:

Type	Description
Normal	A typical window. Normal windows use the full-size style of chrome and appear on the Windows taskbar and the Mac OS X window menu.
Utility	A tool palette. Utility windows use a slimmer version of the system chrome and do not appear on the Windows taskbar and the Mac OS X window menu.
Lightweight	Lightweight windows have no chrome and do not appear on the Windows taskbar or the Mac OS X window menu. In addition, lightweight windows do not have the System (Alt+Space) menu on Windows. Lightweight windows are suitable for notification bubbles and controls such as combo-boxes that open a short-lived display area. When the <code>lightweight</code> type is used, <code>systemChrome</code> must be set to <code>none</code> .

Window chrome

Adobe AIR 1.0 and later

Window chrome is the set of controls that allow users to manipulate a window in the desktop environment. Chrome elements include the title bar, title bar buttons, border, and resize grippers.

System chrome

You can set the `systemChrome` property to `standard` or `none`. Choose `standard` system chrome to give your window the set of standard controls created and styled by the user's operating system. Choose `none` to provide your own chrome for the window. Use the constants defined in the `NativeWindowSystemChrome` class to reference the system chrome settings in code.

System chrome is managed by the system. Your application has no direct access to the controls themselves, but can react to the events dispatched when the controls are used. When you use standard chrome for a window, the `transparent` property must be set to `false` and the `type` property must be `normal` or `utility`.

Custom chrome

When you create a window with no system chrome, then you must add your own chrome controls to handle the interactions between a user and the window. You are also free to make transparent, non-rectangular windows.

Window transparency

Adobe AIR 1.0 and later

To allow alpha blending of a window with the desktop or other windows, set the window `transparent` property to `true`. The `transparent` property must be set before the window is created and cannot be changed.

A transparent window has no default background. Any window area not containing an object drawn by the application is invisible. If a displayed object has an alpha setting of less than one, then anything below the object shows through, including other display objects in the same window, other windows, and the desktop.

Transparent windows are useful when you want to create applications with borders that are irregular in shape or that "fade out" or appear to be invisible. However, rendering large alpha-blended areas can be slow, so the effect should be used conservatively.

Important: On Linux, mouse events do not pass through fully transparent pixels. You should avoid creating windows with large, fully transparent areas since you may invisibly block the user's access to other windows or items on their desktop. On Mac OS X and Windows, mouse events do pass through fully transparent pixels.

Transparency cannot be used with windows that have system chrome. In addition, SWF and PDF content in HTML may not display in transparent windows. For more information, see [“Considerations when loading SWF or PDF content in an HTML page”](#) on page 48.

The static `NativeWindow.supportsTransparency` property reports whether window transparency is available. When transparency is not supported, the application is composited against a black background. In these cases, any transparent areas of the application display as an opaque black. It is a good practice to provide a fallback in case this property tests `false`. For example, you could display a warning dialog to the user, or display a rectangular, non-transparent user interface.

Note that transparency is always supported by the Mac and Windows operating systems. Support on Linux operating systems requires a compositing window manager, but even when a compositing window manager is active, transparency can be unavailable because of user display options or hardware configuration.

Transparency in an HTML application window

Adobe AIR 1.0 and later

By default the background of HTML content displayed in HTML windows and `HTMLLoader` objects is opaque, even if the containing window is transparent. To turn off the default background displayed for HTML content, set the `paintsDefaultBackground` property to `false`. The following example creates an `HTMLLoader` and turns off the default background:

```
var htmlView:HTMLLoader = new HTMLLoader();  
htmlView.paintsDefaultBackground = false;
```

This example uses JavaScript to turn off the default background of an HTML window:

```
window.htmlLoader.paintsDefaultBackground = false;
```

If an element in the HTML document sets a background color, the background of that element is not transparent. Setting a partial transparency (or opacity) value is not supported. However, you can use a transparent PNG-format graphic as the background for a page or a page element to achieve a similar visual effect.

Window ownership

One window can *own* one or more other windows. These owned windows always appear in front of the master window, are minimized and restored along with the master window, and are closed when the master window is closed. Window ownership cannot be transferred to another window or removed. A window can only be owned by one master window, but can own any number of other windows.



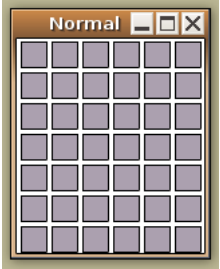
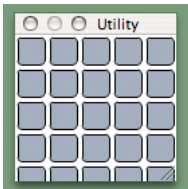


You can use window ownership to make it easier to manage windows used for tool palettes and dialogs. For example, if you displayed a Save dialog in association with a document window, making the document window own the dialog will keep the dialog in front of the document window automatically.

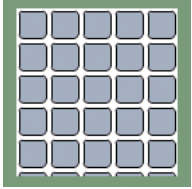
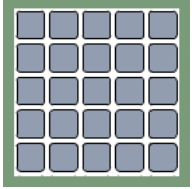
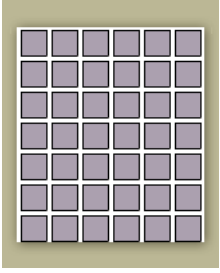
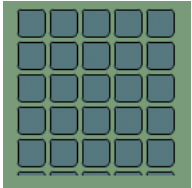
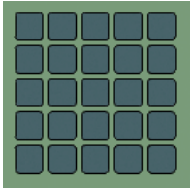
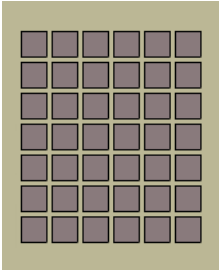


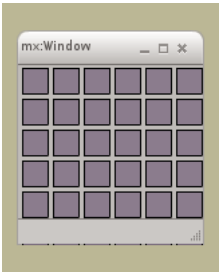
- [NativeWindow.owner](#)
- [Christian Cantrell: Owned windows in AIR 2.6](#)

A visual window catalog

Adobe AIR 1.0 and later

The following table illustrates the visual effects of different combinations of window property settings on the Mac OS X, Windows, and Linux operating systems:

Window settings	Mac OS X	Microsoft Windows	Linux*
Type: normal SystemChrome: standard Transparent: false	 A screenshot of a normal window on Mac OS X. The title bar is light gray with three window control buttons (red, yellow, green) on the left. The window content is a 5x5 grid of blue squares.	 A screenshot of a normal window on Microsoft Windows. The title bar is blue with a red 'N...' icon on the left and minimize, maximize, and close buttons on the right. The window content is a 5x5 grid of blue squares.	 A screenshot of a normal window on Linux. The title bar is brown with the text 'Normal' on the left and minimize, maximize, and close buttons on the right. The window content is a 5x5 grid of blue squares.
Type: utility SystemChrome: standard Transparent: false	 A screenshot of a utility window on Mac OS X. The title bar is light gray with three window control buttons (red, yellow, green) on the left. The window content is a 5x5 grid of blue squares.	 A screenshot of a utility window on Microsoft Windows. The title bar is blue with the text 'Utility' on the left and a close button on the right. The window content is a 5x5 grid of blue squares.	 A screenshot of a utility window on Linux. The title bar is brown with the text 'Utility' on the left and a close button on the right. The window content is a 5x5 grid of blue squares.

Window settings	Mac OS X	Microsoft Windows	Linux*
Type: Any SystemChrome: none Transparent: false			
Type: Any SystemChrome: none Transparent: true			
mxWindowedApplication or mx:Window Type: Any SystemChrome: none Transparent: true			

*Ubuntu with Compiz window manager

Note: The following system chrome elements are not supported by AIR: the Mac OS X Toolbar, the Mac OS X Proxy Icon, Windows title bar icons, and alternate system chrome.

Creating windows

Adobe AIR 1.0 and later

AIR automatically creates the first window for an application, but you can create any additional windows you need. To create a native window, use the `NativeWindow` constructor method.

To create an HTML window, either use the `HTMLLoader` `createRootWindow()` method or, from an HTML document, call the JavaScript `window.open()` method. The window created is a `NativeWindow` object whose display list contains an `HTMLLoader` object. The `HTMLLoader` object interprets and displays the HTML and JavaScript content for the window. You can access the properties of the underlying `NativeWindow` object from JavaScript using the `window.nativeWindow` property. (This property is only accessible to code running in the AIR application sandbox.)

When you initialize a window—including the initial application window—you should consider creating the window in the invisible state, loading content or executing any graphical updates, and then making the window visible. This sequence prevents any jarring visual changes from being visible to your users. You can specify that the initial window of your application should be created in the invisible state by specifying the `<visible>false</visible>` tag in the application descriptor (or by leaving the tag out altogether since `false` is the default value). New `NativeWindows` are invisible by default. When you create an HTML window with the `HTMLLoader.createRootWindow()` method, you can set the `visible` argument to `false`. Call the `NativeWindow.activate()` method or set the `visible` property to `true` to make a window visible.

Specifying window initialization properties

Adobe AIR 1.0 and later

The initialization properties of a native window cannot be changed after the desktop window is created. These immutable properties and their default values include:

Property	Default value
<code>systemChrome</code>	<code>standard</code>
<code>type</code>	<code>normal</code>
<code>transparent</code>	<code>false</code>
<code>owner</code>	<code>null</code>
<code>maximizable</code>	<code>true</code>
<code>minimizable</code>	<code>true</code>
<code>resizable</code>	<code>true</code>

Set the properties for the initial window created by AIR in the application descriptor file. The main window of an AIR application is always `type`, `normal`. (Additional window properties can be specified in the descriptor file, such as `visible`, `width`, and `height`, but these properties can be changed at any time.)

Set the properties for other native and HTML windows created by your application using the `NativeWindowInitOptions` class. When you create a window, you must pass a `NativeWindowInitOptions` object specifying the window properties to either the `NativeWindow` constructor function or the `HTMLLoader.createRootWindow()` method.

The following code creates a `NativeWindowInitOptions` object for a utility window:

```
var options = new air.NativeWindowInitOptions();
options.systemChrome = air.NativeWindowSystemChrome.STANDARD;
options.type = air.NativeWindowType.UTILITY;
options.transparent = false;
options.resizable = false;
options.maximizable = false;
```

Setting `systemChrome` to `standard` when `transparent` is `true` or `type` is `lightweight` is *not supported*.

Note: You cannot set the initialization properties for a window created with the JavaScript `window.open()` function. You can, however, override how these windows are created by implementing your own `HTMLHost` class. See [“Handling JavaScript calls to `window.open\(\)`”](#) on page 61 for more information.

Creating the initial application window

Adobe AIR 1.0 and later

Use a standard HTML page for the initial window of your application. This page is loaded from the application install directory and placed in the application sandbox. The page serves as the initial entry point for your application.

When your application launches, AIR creates a window, sets up the HTML environment, and loads your HTML page. Before parsing any scripts or adding any elements to the HTML DOM, AIR adds the `runtime`, `htmlLoader`, and `nativeWindow` properties to the JavaScript Window object. You can use these properties to access the runtime classes from JavaScript. The `nativeWindow` property gives you direct access to the properties and methods of the desktop window.

The following example illustrates the basic skeleton for the main page of an AIR application built with HTML. The page waits for the JavaScript window `load` event and then shows the native window.

```
<html>
  <head>
    <script language="javascript" type="text/javascript" src="AIRAliases.js"></script>
    <script language="javascript">
      window.onload=init;

      function init(){
        window.nativeWindow.activate();
      }
    </script>
  </head>
  <body></body>
</html>
```

Note: The `AIRAliases.js` file referenced in this example is a script file that defines convenient alias variables for the commonly used built-in AIR classes. The file is available inside the `frameworks` directory of the AIR SDK.

Creating a NativeWindow

Adobe AIR 1.0 and later

To create a `NativeWindow`, pass a `NativeWindowInitOptions` object to the `NativeWindow` constructor:

```
var options = new air.NativeWindowInitOptions();
options.systemChrome = air.NativeWindowSystemChrome.STANDARD;
options.transparent = false;
var newWindow = new air.NativeWindow(options);
```

The window is not shown until you set the `visible` property to `true` or call the `activate()` method.

Once the window is created, you can initialize its properties and load content into the window using the stage property and Flash display list techniques.

In almost all cases, you should set the stage `scaleMode` property of a new native window to `noScale` (use the `StageScaleMode.NO_SCALE` constant). The Flash scale modes are designed for situations in which the application author does not know the aspect ratio of the application display space in advance. The scale modes let the author choose the least-bad compromise: clip the content, stretch or squash it, or pad it with empty space. Since you control the display space in AIR (the window frame), you can size the window to the content or the content to the window without compromise.

The scale mode for HTML windows is set to `noScale` automatically.

Note: To determine the maximum and minimum window sizes allowed on the current operating system, use the following static `NativeWindow` properties:

```
var maxOSSize = air.NativeWindow.systemMaxSize;  
var minOSSize = air.NativeWindow.systemMinSize;
```

Creating an HTML window

Adobe AIR 1.0 and later

To create an HTML window, you can either call the JavaScript `window.open()` method, or you can call the AIR `HTMLLoader` class `createRootWindow()` method.

HTML content in any security sandbox can use the standard JavaScript `window.open()` method. If the content is running outside the application sandbox, the `open()` method can only be called in response to user interaction, such as a mouse click or keypress. When `open()` is called, a window with system chrome is created to display the content at the specified URL. For example:

```
newWindow = window.open("xml.html", "logWindow", "height=600, width=400, top=10, left=10");
```

Note: You can extend the `HTMLHost` class in ActionScript to customize the window created with the JavaScript `window.open()` function. See [“About extending the HTMLHost class”](#) on page 52.

Content in the application security sandbox has access to the more powerful method of creating windows, `HTMLLoader.createRootWindow()`. With this method, you can specify all the creation options for a new window. For example, the following JavaScript code creates a lightweight type window without system chrome that is 300x400 pixels in size:

```
var options = new air.NativeWindowInitOptions();  
options.systemChrome = "none";  
options.type = "lightweight";  
  
var windowBounds = new air.Rectangle(200,250,300,400);  
newHTMLLoader = air.HTMLLoader.createRootWindow(true, options, true, windowBounds);  
newHTMLLoader.load(new air.URLRequest("xml.html"));
```

Note: If the content loaded by a new window is outside the application security sandbox, the window object does not have the AIR properties: `runtime`, `nativeWindow`, or `htmlLoader`.

If you create a transparent window, then SWF content embedded in the HTML loaded into that window is not always displayed. You must set the `wmode` parameter of the object or embed tag used to reference the SWF file to either `opaque` or `transparent`. The default value of `wmode` is `window`, so, by default, SWF content is not displayed in transparent windows. PDF content cannot be displayed in transparent windows, no matter which `wmode` value is set. (Prior to AIR 1.5.2, SWF content could not be displayed in transparent windows, either.)

Windows created with the `createRootWindow()` method remain independent from the opening window. The `parent` and `opener` properties of the JavaScript `Window` object are `null`. The opening window can access the `Window` object of the new window using the `HTMLLoader` reference returned by the `createRootWindow()` function. In the context of the previous example, the statement `newHTMLLoader.window` would reference the JavaScript `Window` object of the created window.

Note: The `createRootWindow()` function can be called from both JavaScript and ActionScript.

Creating a mx:Window

Adobe AIR 1.0 and later

To create a mx:Window, you can create an MXML file using mx:Window as the root tag, or you can call the Window class constructor directly.

The following example creates and shows a mx:Window by calling the Window constructor:

```
var newWindow:Window = new Window();
newWindow.systemChrome = NativeWindowSystemChrome.NONE;
newWindow.transparent = true;
newWindow.title = "New Window";
newWindow.width = 200;
newWindow.height = 200;
newWindow.open(true);
```

Adding content to a window

Adobe AIR 1.0 and later

How you add content to an AIR window depends on the type of window. For example, MXML and HTML let you declaratively define the basic content of the window. You can embed resources in the application SWF files or you can load them from separate application files. Flex, Flash, and HTML content can all be created on the fly and added to a window dynamically.

When you load SWF content, or HTML content containing JavaScript, you must take the AIR security model into consideration. Any content in the application security sandbox, that is, content installed with your application and loadable with the app: URL scheme, has full privileges to access all the AIR APIs. Any content loaded from outside this sandbox cannot access the AIR APIs. JavaScript content outside the application sandbox is not able to use the runtime, nativeWindow, or htmlLoader properties of the JavaScript Window object.

To allow safe cross-scripting, you can use a sandbox bridge to provide a limited interface between application content and non-application content. In HTML content, you can also map pages of your application into a non-application sandbox to allow the code on that page to cross-script external content. See “[AIR security](#)” on page 69.

Loading HTML content into a NativeWindow

To load HTML content into a NativeWindow, you can either add an HTMLLoader object to the window stage and load the HTML content into the HTMLLoader, or create a window that already contains an HTMLLoader object by using the HTMLLoader.createRootWindow() method. The following example displays HTML content within a 300 by 500 pixel display area on the stage of a native window:

```
//newWindow is a NativeWindow instance
var htmlView:HTMLLoader = new HTMLLoader();
htmlView.width = 300;
htmlView.height = 500;

//set the stage so display objects are added to the top-left and not scaled
newWindow.stage.align = "TL";
newWindow.stage.scaleMode = "noScale";
newWindow.stage.addChild( htmlView );

//urlString is the URL of the HTML page to load
htmlView.load( new URLRequest( urlString ) );
```

To load an HTML page into a Flex application, you can use the Flex HTML component.

SWF content in an HTML file is not displayed if the window uses transparency (that is the `transparent` property of the window is `true`) unless the `wmode` parameter of the object or embed tag used to reference the SWF file is set to either `opaque` or `transparent`. Since the default `wmode` value is `window`, by default, SWF content is not displayed in a transparent window. PDF content is not displayed in a transparent window no matter what `wmode` value is used.

Also, neither SWF nor PDF content is displayed if the `HTMLLoader` control is scaled, rotated, or if the `HTMLLoader.alpha` property is set to a value other than 1.0.

Adding SWF content as an overlay on an HTML window

Because HTML windows are contained within a `NativeWindow` instance, you can add Flash display objects both above and below the HTML layer in the display list.

To add a display object above the HTML layer, use the `addChild()` method of the `window.nativeWindow.stage` property. The `addChild()` method adds content layered above any existing content in the window.

To add a display object below the HTML layer, use the `addChildAt()` method of the `window.nativeWindow.stage` property, passing in a value of zero for the `index` parameter. Placing an object at the zero index moves existing content, including the HTML display, up one layer and insert the new content at the bottom. For content layered underneath the HTML page to be visible, you must set the `paintsDefaultBackground` property of the `HTMLLoader` object to `false`. In addition, any elements of the page that set a background color, will not be transparent. If, for example, you set a background color for the body element of the page, none of the page will be transparent.

The following example illustrates how to add a Flash display objects as overlays and underlays to an HTML page. The example creates two simple shape objects, adds one below the HTML content and one above. The example also updates the shape position based on the `enterFrame` event.

```
<html>
<head>
<title>Bouncers</title>
<script src="AIRAliases.js" type="text/javascript"></script>
<script language="JavaScript" type="text/javascript">
air.Shape = window.runtime.flash.display.Shape;

function Bouncer(radius, color){
    this.radius = radius;
    this.color = color;

    //velocity
    this.vX = -1.3;
    this.vY = -1;

    //Create a Shape object and draw a circle with its graphics property
    this.shape = new air.Shape();
    this.shape.graphics.lineStyle(1,0);
    this.shape.graphics.beginFill(this.color, .9);
    this.shape.graphics.drawCircle(0,0,this.radius);
    this.shape.graphics.endFill();

    //Set the starting position
    this.shape.x = 100;
    this.shape.y = 100;

    //Moves the sprite by adding (vX,vY) to the current position
```

```
    this.update = function(){
        this.shape.x += this.vX;
        this.shape.y += this.vY;

        //Keep the sprite within the window
        if( this.shape.x - this.radius < 0){
            this.vX = -this.vX;
        }
        if( this.shape.y - this.radius < 0){
            this.vY = -this.vY;
        }
        if( this.shape.x + this.radius > window.nativeWindow.stage.stageWidth){
            this.vX = -this.vX;
        }
        if( this.shape.y + this.radius > window.nativeWindow.stage.stageHeight){
            this.vY = -this.vY;
        }
    };
}

function init(){
    //turn off the default HTML background
    window.htmlLoader.paintsDefaultBackground = false;
    var bottom = new Bouncer(60,0xff2233);
    var top = new Bouncer(30,0x2441ff);

    //listen for the enterFrame event
    window.htmlLoader.addEventListener("enterFrame",function(evt){
        bottom.update();
        top.update();
    });

    //add the bouncing shapes to the window stage
    window.nativeWindow.stage.addChildAt(bottom.shape,0);
    window.nativeWindow.stage.addChild(top.shape);
}
</script>
<body onload="init();">
<h1>de Finibus Bonorum et Malorum</h1>
<p>Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium
doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis
et quasi architecto beatae vitae dicta sunt explicabo.</p>
<p style="background-color:#FFFF00; color:#660000;">This paragraph has a background color.</p>
<p>At vero eos et accusamus et iusto odio dignissimos ducimus qui blanditiis
praesentium voluptatum deleniti atque corrupti quos dolores et quas molestias
excepturi sint occaecati cupiditate non provident, similique sunt in culpa qui
officia deserunt mollitia animi, id est laborum et dolorum fuga.</p>
</body>
</html>
```

This example provides a rudimentary introduction to some advanced techniques that cross over the boundaries between JavaScript and ActionScript in AIR. If you are unfamiliar with using ActionScript display objects, refer to Display programming in the *ActionScript 3.0 Developer's Guide*.

Note: To access the runtime, `nativeWindow` and `htmlLoader` properties of the JavaScript `Window` object, the HTML page must be loaded from the application directory. This will always be the case for the root page in an HTML-based application, but may not be true for other content. In addition, documents loaded into frames even within the application sandbox do not receive these properties, but can access those of the parent document.

Example: Creating a native window

Adobe AIR 1.0 and later

The following example illustrates how to create a native window:

```
function createNativeWindow() {
    //create the init options
    var options = new air.NativeWindowInitOptions();
    options.transparent = false;
    options.systemChrome = air.NativeWindowSystemChrome.STANDARD;
    options.type = air.NativeWindowType.NORMAL;

    //create the window
    var newWindow = new air.NativeWindow(options);
    newWindow.title = "A title";
    newWindow.width = 600;
    newWindow.height = 400;

    //activate and show the new window
    newWindow.activate();
}
```

Managing windows

Adobe AIR 1.0 and later

You use the properties and methods of the `NativeWindow` class to manage the appearance, behavior, and life cycle of desktop windows.

Note: When using the Flex framework, it is generally better to manage window behavior using the framework classes. Most of the `NativeWindow` properties and methods can be accessed through the `mx:WindowedApplication` and `mx:Window` classes.

Getting a NativeWindow instance

Adobe AIR 1.0 and later

To manipulate a window, you must first get the window instance. You can get a window instance from one of the following places:

- The native window constructor used to create the window:

```
var nativeWin = new air.NativeWindow(initOptions);
```

- The `stage` property of a display object in the window:

```
var nativeWin = window.htmlLoader.stage.nativeWindow;
```

- The target property of a native window event dispatched by the window:

```
function onNativeWindowEvent(event)
{
    var nativeWin = event.target;
}
```

- The nativeWindow property of an HTML page displayed in the window:

```
var nativeWin = window.nativeWindow;
```

- The activeWindow and openedWindows properties of the NativeApplication object:

```
var win = NativeApplication.nativeApplication.activeWindow;
var firstWindow = NativeApplication.nativeApplication.openedWindows[0];
```

NativeApplication.nativeApplication.activeWindow references the active window of an application (but returns null if the active window is not a window of this AIR application). The NativeApplication.nativeApplication.openedWindows array contains all of the windows in an AIR application that have not been closed.

Because the Flex mx:WindowedApplication, and mx:Window objects are display objects, you can easily reference the application window in an MXML file using the stage property, as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
applicationComplete="init();">
    <mx:Script>
        <![CDATA[
            import flash.display.NativeWindow;

            public function init():void{
                var appWindow:NativeWindow = this.stage.nativeWindow;
                //set window properties
                appWindow.visible = true;
            }
        ]]>
    </mx:Script>
</WindowedApplication>
```

Note: Until the WindowedApplication or Window component is added to the window stage by the Flex framework, the component's stage property is null. This behavior is consistent with that of the Flex Application component, but does mean that it is not possible to access the stage or the NativeWindow instance in listeners for events that occur earlier in the initialization cycle of the WindowedApplication and Window components, such as creationComplete. It is safe to access the stage and NativeWindow instance when the applicationComplete event is dispatched.

Activating, showing, and hiding windows

Adobe AIR 1.0 and later

To activate a window, call the NativeWindow activate() method. Activating a window brings the window to the front, gives it keyboard and mouse focus, and, if necessary, makes it visible by restoring the window or setting the visible property to true. Activating a window does not change the ordering of other windows in the application. Calling the activate() method causes the window to dispatch an activate event.

To show a hidden window without activating it, set the visible property to true. This brings the window to the front, but will not assign the focus to the window.

To hide a window from view, set its `visible` property to `false`. Hiding a window suppresses the display of both the window, any related taskbar icons, and, on Mac OS X, the entry in the Windows menu.

When you change the visibility of a window, the visibility of any windows that window owns is also changed. For example, if you hide a window, all of its owned windows are also hidden.

***Note:** On Mac OS X, it is not possible to completely hide a minimized window that has an icon in the window portion of the dock. If the `visible` property is set to `false` on a minimized window, the dock icon for the window is still displayed. If the user clicks the icon, the window is restored to a visible state and displayed.*

Changing the window display order

Adobe AIR 1.0 and later

AIR provides several methods for directly changing the display order of windows. You can move a window to the front of the display order or to the back; you can move a window above another window or behind it. At the same time, the user can reorder windows by activating them.

You can keep a window in front of other windows by setting its `alwaysInFront` property to `true`. If more than one window has this setting, then the display order of these windows is sorted among each other, but they are always sorted above windows which have `alwaysInFront` set to `false`.

Windows in the top-most group are also displayed above windows in other applications, even when the AIR application is not active. Because this behavior can be disruptive to a user, setting `alwaysInFront` to `true` should only be done when necessary and appropriate. Examples of justified uses include:

- Temporary pop-up windows for controls such as tool tips, pop-up lists, custom menus, or combo boxes. Because these windows should close when they lose focus, the annoyance of blocking a user from viewing another window can be avoided.
- Extremely urgent error messages and alerts. When an irrevocable change may occur if the user does not respond in a timely manner, it may be justified to push an alert window to the forefront. However, most errors and alerts can be handled in the normal window display order.
- Short-lived toast-style windows.

***Note:** AIR does not enforce proper use of the `alwaysInFront` property. However, if your application disrupts a user's workflow, it is likely to be consigned to that same user's trash can.*

If a window owns other windows, those windows are always ordered in front of it. If you call `orderToFront()` or set `alwaysInFront` to `true` on a window that owns other windows, then the owned windows are re-ordered along with the owner window in front of other windows, but the owned windows still display in front of the owner.

Calling the window ordering methods on owned windows works normally among windows owned by the same window, but can also change the ordering of the entire group of owned windows compared to windows outside that group. For example, if you call `orderToFront()` on an owned window, then both that window, its owner, and any other windows owned by the same owner are moved to the front of the window display order.

The `NativeWindow` class provides the following properties and methods for setting the display order of a window relative to other windows:

Member	Description
<code>alwaysInFront</code> property	Specifies whether the window is displayed in the top-most group of windows. In almost all cases, <code>false</code> is the best setting. Changing the value from <code>false</code> to <code>true</code> brings the window to the front of all windows (but does not activate it). Changing the value from <code>true</code> to <code>false</code> orders the window behind windows remaining in the top-most group, but still in front of other windows. Setting the property to its current value for a window does not change the window display order. The <code>alwaysInFront</code> setting has no affect on windows owned by another window.
<code>orderToFront()</code>	Brings the window to the front.
<code>orderInFrontOf()</code>	Brings the window directly in front of a particular window.
<code>orderToBack()</code>	Sends the window behind other windows.
<code>orderBehind()</code>	Sends the window directly behind a particular window.
<code>activate()</code>	Brings the window to the front (along with making the window visible and assigning focus).

Note: If a window is hidden (`visible` is `false`) or minimized, then calling the display order methods has no effect.

On the Linux operating system, different window managers enforce different rules regarding the window display order:

- On some window managers, utility windows are always displayed in front of normal windows.
- On some window managers, a full screen window with `alwaysInFront` set to `true` is always displayed in front of other windows that also have `alwaysInFront` set to `true`.

Closing a window

Adobe AIR 1.0 and later

To close a window, use the `NativeWindow.close()` method.

Closing a window unloads the contents of the window, but if other objects have references to this content, the content objects will not be destroyed. The `NativeWindow.close()` method executes asynchronously, the application that is contained in the window continues to run during the closing process. The close method dispatches a close event when the close operation is complete. The `NativeWindow` object is still technically valid, but accessing most properties and methods on a closed window generates an `IllegalOperationError`. You cannot reopen a closed window. Check the `closed` property of a window to test whether a window has been closed. To simply hide a window from view, set the `NativeWindow.visible` property to `false`.

If the `NativeApplication.autoExit` property is `true`, which is the default, then the application exits when its last window closes.

Any windows that have an owner are closed when the owner is closed. The owned windows do not dispatch a closing event and hence cannot prevent closure. A close event is dispatched.

Allowing cancellation of window operations

Adobe AIR 1.0 and later

When a window uses system chrome, user interaction with the window can be canceled by listening for, and canceling the default behavior of the appropriate events. For example, when a user clicks the system chrome close button, the `closing` event is dispatched. If any registered listener calls the `preventDefault()` method of the event, then the window does not close.

When a window does not use system chrome, notification events for intended changes are not automatically dispatched before the change is made. Hence, if you call the methods for closing a window, changing the window state, or set any of the window bounds properties, the change cannot be canceled. To notify components in your application before a window change is made, your application logic can dispatch the relevant notification event using the `dispatchEvent()` method of the window.

For example, the following logic implements a cancelable event handler for a window close button:

```
function onCloseCommand(event) {
    var closingEvent = new air.Event(air.Event.CLOSING, true, true);
    dispatchEvent(closingEvent);
    if (!closingEvent.isDefaultPrevented()) {
        win.close();
    }
}
```

The `dispatchEvent()` method returns `false` if the event `preventDefault()` method is called by a listener. However, it can also return `false` for other reasons, so it is better to explicitly use the `isDefaultPrevented()` method to test whether the change should be canceled.

Maximizing, minimizing, and restoring a window

Adobe AIR 1.0 and later

To maximize the window, use the `NativeWindow maximize()` method.

```
window.nativeWindow.maximize();
```

To minimize the window, use the `NativeWindow minimize()` method.

```
window.nativeWindow.minimize();
```

To restore the window (that is, return it to the size that it was before it was either minimized or maximized), use the `NativeWindow restore()` method.

```
window.nativeWindow.restore();
```

A window that has an owner is minimized and restored when the owning window is minimized or restored. No events are dispatched by the owned window when it is minimized because its owner is minimized.

Note: The behavior that results from maximizing an AIR window is different from the Mac OS X standard behavior. Rather than toggling between an application-defined “standard” size and the last size set by the user, AIR windows toggle between the size last set by the application or user and the full usable area of the screen.

On the Linux operating system, different window managers enforce different rules regarding setting the window display state:

- On some window managers, utility windows cannot be maximized.
- If a maximum size is set for the window, then some windows do not allow a window to be maximized. Some other window managers set the display state to maximized, but do not resize the window. In either of these cases, no display state change event is dispatched.
- Some window managers do not honor the window `maximizable` or `minimizable` settings.

Note: On Linux, window properties are changed asynchronously. If you change the display state in one line of your program, and read the value in the next, the value read will still reflect the old setting. On all platforms, the `NativeWindow` object dispatches the `displayStateChange` event when the display state changes. If you need to take some action based on the new state of the window, always do so in a `displayStateChange` event handler. See “[Listening for window events](#)” on page 105.

Example: Minimizing, maximizing, restoring and closing a window

Adobe AIR 1.0 and later

The following short HTML page demonstrates the `NativeWindow` `maximize()`, `minimize()`, `restore()`, and `close()` methods:

```
<html>
<head>
<title>Change Window Display State</title>
<script src="AIRAliases.js"/>
<script type="text/javascript">
    function onMaximize() {
        window.nativeWindow.maximize();
    }

    function onMinimize() {
        window.nativeWindow.minimize();
    }

    function onRestore() {
        window.nativeWindow.restore();
    }

    function onClose() {
        window.nativeWindow.close();
    }
</script>
</head>

<body>
    <h1>AIR window display state commands</h1>
    <button onClick="onMaximize()">Maximize</button>
    <button onClick="onMinimize()">Minimize</button>
    <button onClick="onRestore()">Restore</button>
    <button onClick="onClose()">Close</button>
</body>
</html>
```

Resizing and moving a window

Adobe AIR 1.0 and later

When a window uses system chrome, the chrome provides drag controls for resizing the window and moving around the desktop. If a window does not use system chrome you must add your own controls to allow the user to resize and move the window.

Note: To resize or move a window, you must first obtain a reference to the `NativeWindow` instance. For information about how to obtain a window reference, see [“Getting a NativeWindow instance”](#) on page 97.

Resizing a window

To allow a user to resize a window interactively, use the `NativeWindow.startResize()` method. When this method is called from a `mouseDown` event, the resizing operation is driven by the mouse and completes when the operating system receives a `mouseUp` event. When calling `startResize()`, you pass in an argument that specifies the edge or corner from which to resize the window.

To set the window size programmatically, set the `width`, `height`, or `bounds` properties of the window to the desired dimensions. When you set the bounds, the window size and position can all be changed at the same time. However, the order that the changes occur is not guaranteed. Some Linux window managers do not allow windows to extend outside the bounds of the desktop screen. In these cases, the final window size may be limited because of the order in which the properties are set, even though the net affect of the changes would otherwise have resulted in a legal window. For example, if you change both the height and y position of a window near the bottom of the screen, then the full height change might not occur when the height change is applied before the y position change.

Note: On Linux, window properties are changed asynchronously. If you resize a window in one line of your program, and read the dimensions in the next, they will still reflect the old settings. On all platforms, the `NativeWindow` object dispatches the `resize` event when the window resizes. If you need to take some action, such as laying out controls in the window, based on the new size or state of the window, always do so in a `resize` event handler. See [“Listening for window events”](#) on page 105.

Moving a window

To move a window without resizing it, use the `NativeWindow.startMove()` method. Like the `startResize()` method, when the `startMove()` method is called from a `mouseDown` event, the move process is mouse-driven and completes when the operating system receives a `mouseUp` event.

For more information about the `startResize()` and `startMove()` methods, see the [Adobe AIR API Reference for HTML Developers](#).

To move a window programmatically, set the `x`, `y`, or `bounds` properties of the window to the desired position. When you set the bounds, the window size and position can both be changed at the same time.

Note: On Linux, window properties are changed asynchronously. If you move a window in one line of your program, and read the position in the next, the value read will still reflect the old setting. On all platforms, the `NativeWindow` object dispatches the `move` event when the position changes. If you need to take some action based on the new position of the window, always do so in a `move` event handler. See [“Listening for window events”](#) on page 105.

Example: Resizing and moving windows

Adobe AIR 1.0 and later

The following example shows how to initiate resizing and moving operations on a window:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<script src="AIRAliases.js"/>
<script type="text/javascript">
    function onResize(type){
        nativeWindow.startResize(type);
    }

    function onNativeMove(){
        nativeWindow.startMove();
    }
</script>
<style type="text/css" media="screen">

.drag {
    width:200px;
    height:200px;
    margin:0px auto;
    padding:15px;
    border:1px dashed #333;
    background-color:#eee;
}

.resize {
    background-color:#FF0000;
    padding:10px;
}

.left {
    float:left;
}

.right {
    float:right;
}

</style>
<title>Move and Resize the Window</title>
</head>

<body>
<div class="resize left" onmousedown="onResize(air.NativeWindowResize.TOP_LEFT)">Drag to
resize</div>
<div class="resize right" onmousedown="onResize(air.NativeWindowResize.TOP_RIGHT)">Drag to
resize</div>
<div class="drag" onmousedown="onNativeMove()">Drag to move</div>
<div class="resize left" onmousedown="onResize(air.NativeWindowResize.BOTTOM_LEFT)">Drag to
resize</div>
<div class="resize right" onmousedown="onResize(air.NativeWindowResize.BOTTOM_RIGHT)">Drag to
resize</div>
</body>
</html>
```

Listening for window events

Adobe AIR 1.0 and later

To listen for the events dispatched by a window, register a listener with the window instance. For example, to listen for the closing event, register a listener with the window as follows:

```
window.nativeWindow.addEventListener(air.Event.CLOSING, onClosingEvent);
```

When an event is dispatched, the `target` property references the window sending the event.

Most window events have two related messages. The first message signals that a window change is imminent (and can be canceled), while the second message signals that the change has occurred. For example, when a user clicks the close button of a window, the closing event message is dispatched. If no listeners cancel the event, the window closes and the close event is dispatched to any listeners.

Typically, the warning events, such as `closing`, are only dispatched when system chrome has been used to trigger an event. Calling the window `close()` method, for example, does not automatically dispatch the `closing` event—only the `close` event is dispatched. You can, however, construct a closing event object and dispatch it using the window `dispatchEvent()` method.

The window events that dispatch an Event object are:

Event	Description
activate	Dispatched when the window receives focus.
deactivate	Dispatched when the window loses focus
closing	Dispatched when the window is about to close. This only occurs automatically when the system chrome close button is pressed or, on Mac OS X, when the Quit command is invoked.
close	Dispatched when the window has closed.

The window events that dispatch an `NativeWindowBoundsEvent` object are:

Event	Description
moving	Dispatched immediately before the top-left corner of the window changes position, either as a result of moving, resizing or changing the window display state.
move	Dispatched after the top-left corner has changed position.
resizing	Dispatched immediately before the window width or height changes either as a result of resizing or a display state change.
resize	Dispatched after the window has changed size.

For `NativeWindowBoundsEvent` events, you can use the `beforeBounds` and `afterBounds` properties to determine the window bounds before and after the impending or completed change.

The window events that dispatch an `NativeWindowDisplayStateEvent` object are:

Event	Description
displayStateChanging	Dispatched immediately before the window display state changes.
displayStateChange	Dispatched after the window display state has changed.

For `NativeWindowDisplayStateEvent` events, you can use the `beforeDisplayState` and `afterDisplayState` properties to determine the window display state before and after the impending or completed change.

On some Linux window managers, a display state change event is not dispatched when a window with a maximum size setting is maximized. (The window is set to the maximized display state, but is not resized.)

Displaying full-screen windows

Adobe AIR 1.0 and later

Setting the `displayState` property of the Stage to `StageDisplayState.FULL_SCREEN_INTERACTIVE` places the window in full-screen mode, and keyboard input *is* permitted in this mode. (In SWF content running in a browser, keyboard input is not permitted). To exit full-screen mode, the user presses the Escape key.

Note: Some Linux window managers will not change the window dimensions to fill the screen if a maximum size is set for the window (but do remove the window system chrome).

For example, the following Flex code defines a simple AIR application that sets up a simple full-screen terminal:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical"
    applicationComplete="init()" backgroundColor="0x003030" focusRect="false">
  <mx:Script>
    <![CDATA[
      private function init():void
      {
        stage.displayState = StageDisplayState.FULL_SCREEN_INTERACTIVE;
        focusManager.setFocus(terminal);
        terminal.text = "Welcome to the dumb terminal app. Press the ESC key to
exit..\n";

        terminal.selectionBeginIndex = terminal.text.length;
        terminal.selectionEndIndex = terminal.text.length;
      }
    ]]>
  </mx:Script>
  <mx:TextArea
    id="terminal"
    height="100%" width="100%"
    scroll="false"
    backgroundColor="0x003030"
    color="0xCCFF00"
    fontFamily="Lucida Console"
    fontSize="44"/>
</mx:WindowedApplication>
```

The following ActionScript example for Flash simulates a simple full-screen text terminal:


```
import flash.display.Sprite;
import flash.display.StageDisplayState;
import flash.text.TextField;
import flash.text.TextFormat;

public class FullScreenTerminalExample extends Sprite
{
    public function FullScreenTerminalExample():void
    {
        var terminal:TextField = new TextField();
        terminal.multiline = true;
        terminal.wordWrap = true;
        terminal.selectable = true;
        terminal.background = true;
        terminal.backgroundColor = 0x00333333;

        this.stage.displayState = StageDisplayState.FULL_SCREEN_INTERACTIVE;

        addChild(terminal);
        terminal.width = 550;
        terminal.height = 400;

        terminal.text = "Welcome to the dumb terminal application. Press the ESC key to
exit.\n_";

        var tf:TextFormat = new TextFormat();
        tf.font = "Courier New";
        tf.color = 0x00CCFF00;
        tf.size = 12;
        terminal.setTextFormat(tf);

        terminal.setSelection(terminal.text.length - 1, terminal.text.length);
    }
}
```

The following HTML page simulates a full screen text terminal:

```
<html>
<head>
<title>Fullscreen Mode</title>
<script language="JavaScript" type="text/javascript">
function setDisplayState() {
    window.nativeWindow.stage.displayState =
        runtime.flash.display.StageDisplayState.FULL_SCREEN_INTERACTIVE;
}
</script>
<style type="text/css">
body, .mono {
    font-family: Courier New, Courier, monospace;
    font-size: x-large;
    color:#CCFF00;
    background-color:#003030;
}
</style>
</head>
<body onload="setDisplayState();">
    <p class="mono">Welcome to the dumb terminal app. Press the ESC key to exit...</p>
    <textarea name="dumb" class="mono" cols="100" rows="40">%</textarea>
</body>
</html>
```

Chapter 8: Display screens in AIR

Adobe AIR 1.0 and later

Use the Adobe® AIR® Screen class to access information about the display screens attached to a computer or device.

More Help topics

[flash.display.Screen](#)

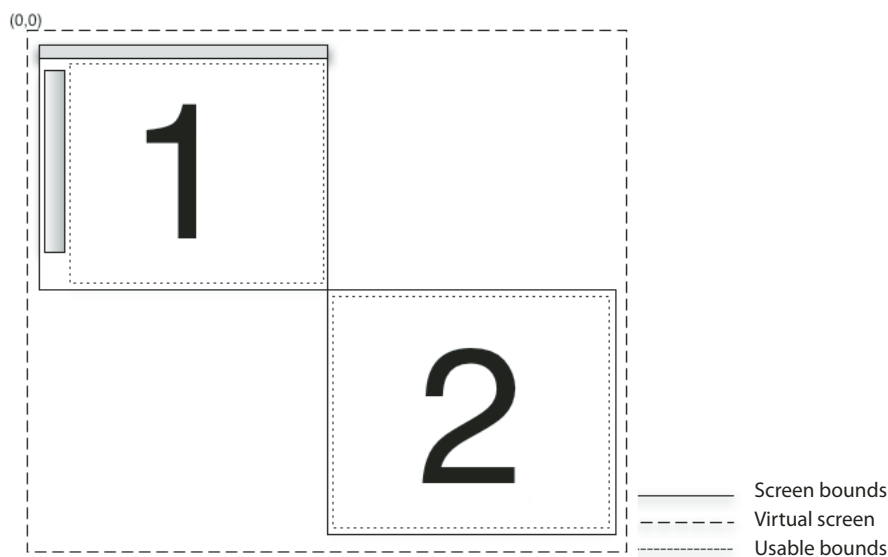
Basics of display screens in AIR

Adobe AIR 1.0 and later

The screen API contains a single class, Screen, which provides static members for getting system screen information, and instance members for describing a particular screen.

A computer system can have several monitors or displays attached, which can correspond to several desktop screens arranged in a virtual space. The AIR Screen class provides information about the screens, their relative arrangement, and their usable space. If more than one monitor maps to the same screen, only one screen exists. If the size of a screen is larger than the display area of the monitor, there is no way to determine which portion of the screen is currently visible.

A screen represents an independent desktop display area. Screens are described as rectangles within the virtual desktop. The upper-left corner of screen designated as the primary display is the origin of the virtual desktop coordinate system. All values used to describe a screen are provided in pixels.



In this screen arrangement, two screens exist on the virtual desktop. The coordinates of the upper-left corner of the main screen (#1) are always (0,0). If the screen arrangement is changed to designate screen #2 as the main screen, then the coordinates of screen #1 become negative. Menubars, taskbars, and docks are excluded when reporting the usable bounds for a screen.

For detailed information about the screen API class, methods, properties, and events, see the [Adobe AIR API Reference for HTML Developers](#).

Enumerating the screens

Adobe AIR 1.0 and later

You can enumerate the screens of the virtual desktop with the following screen methods and properties:

Method or Property	Description
Screen.screens	Provides an array of Screen objects describing the available screens. The order of the array is not significant.
Screen.mainScreen	Provides a Screen object for the main screen. On Mac OS X, the main screen is the screen displaying the menu bar. On Windows, the main screen is the system-designated primary screen.
Screen.getScreensForRectangle()	Provides an array of Screen objects describing the screens intersected by the given rectangle. The rectangle passed to this method is in pixel coordinates on the virtual desktop. If no screens intersect the rectangle, then the array is empty. You can use this method to find out on which screens a window is displayed.

Do not save the values returned by the Screen class methods and properties. The user or operating system can change the available screens and their arrangement at any time.

The following example uses the screen API to move a window between multiple screens in response to pressing the arrow keys. To move the window to the next screen, the example gets the `screens` array and sorts it either vertically or horizontally (depending on the arrow key pressed). The code then walks through the sorted array, comparing each screen to the coordinates of the current screen. To identify the current screen of the window, the example calls `Screen.getScreensForRectangle()`, passing in the window bounds.

```
<html>
  <head>
    <script src="AIRAliases.js" type="text/javascript"></script>
    <script type="text/javascript">
      function onKey(event) {
        if (air.Screen.screens.length > 1) {
          switch (event.keyCode) {
            case air.Keyboard.LEFT :
              moveLeft ();
              break;
            case air.Keyboard.RIGHT :
              moveRight ();
              break;
            case air.Keyboard.UP :
              moveUp ();
              break;
            case air.Keyboard.DOWN :
              moveDown ();
              break;
          }
        }
      }
    </script>
  </head>
</html>
```

```
function moveLeft(){
    var currentScreen = getCurrentScreen();
    var left = air.Screen.screens;
    left.sort(sortHorizontal);
    for(var i = 0; i < left.length - 1; i++){
        if(left[i].bounds.left < window.nativeWindow.bounds.left){
            window.nativeWindow.x += left[i].bounds.left - currentScreen.bounds.left;
            window.nativeWindow.y += left[i].bounds.top - currentScreen.bounds.top;
        }
    }
}

function moveRight(){
    var currentScreen = getCurrentScreen();
    var left = air.Screen.screens;
    left.sort(sortHorizontal);
    for(var i = left.length - 1; i > 0; i--){
        if(left[i].bounds.left > window.nativeWindow.bounds.left){
            window.nativeWindow.x += left[i].bounds.left - currentScreen.bounds.left;
            window.nativeWindow.y += left[i].bounds.top - currentScreen.bounds.top;
        }
    }
}

function moveUp(){
    var currentScreen = getCurrentScreen();
    var top = air.Screen.screens;
    top.sort(sortVertical);
    for(var i = 0; i < top.length - 1; i++){
        if(top[i].bounds.top < window.nativeWindow.bounds.top){
            window.nativeWindow.x += top[i].bounds.left - currentScreen.bounds.left;
            window.nativeWindow.y += top[i].bounds.top - currentScreen.bounds.top;
            break;
        }
    }
}

function moveDown(){
    var currentScreen = getCurrentScreen();

    var top = air.Screen.screens;
    top.sort(sortVertical);
    for(var i = top.length - 1; i > 0; i--){
        if(top[i].bounds.top > window.nativeWindow.bounds.top){
            window.nativeWindow.x += top[i].bounds.left - currentScreen.bounds.left;
            window.nativeWindow.y += top[i].bounds.top - currentScreen.bounds.top;
            break;
        }
    }
}

function sortHorizontal(a,b){
    if (a.bounds.left > b.bounds.left){
        return 1;
    } else if (a.bounds.left < b.bounds.left){
        return -1;
    } else {return 0;}
}
```

```
    }

    function sortVertical(a,b){
        if (a.bounds.top > b.bounds.top){
            return 1;
        } else if (a.bounds.top < b.bounds.top){
            return -1;
        } else {return 0;}
    }

    function getCurrentScreen(){
        var current;
        var screens = air.Screen.getScreensForRectangle(window.nativeWindow.bounds);
        (screens.length > 0) ? current = screens[0] : current = air.Screen.mainScreen;
        return current;
    }

    function init(){
        window.nativeWindow.stage.addEventListener("keyDown",onKey);
    }
</script>
<title>Screen Hopper</title>
</head>
<body onload="init()">
    <p>Use the arrow keys to move the window between monitors.</p>
</body>
</html>
```

Chapter 9: Working with menus

Flash Player 9 and later, Adobe AIR 1.0 and later

Use the classes in the native menu API to define application, window, context, and pop-up menus in Adobe® AIR®.

Menu basics

Flash Player 9 and later, Adobe AIR 1.0 and later

For a quick explanation and code examples of creating native menus in AIR applications, see the following quick start articles on the Adobe Developer Connection:

- [Adding native menus to an AIR application](#)

The native menu classes allow you to access the native menu features of the operating system on which your application is running. NativeMenu objects can be used for application menus (available on Mac OS X), window menus (available on Windows and Linux), context menus, and pop-up menus.

Outside of AIR, you can use the context menu classes to modify the context menu that Flash Player automatically displays when a user right-clicks or cmd-clicks on an object in your application. (An automatic context menu is not displayed for AIR applications.)

Menu classes

Flash Player 9 and later, Adobe AIR 1.0 and later

The menu classes include:

Package	Classes
flash.display	<ul style="list-style-type: none"> • NativeMenu • NativeMenuItem
flash.events	<ul style="list-style-type: none"> • Event

Menu varieties

Flash Player 9 and later, Adobe AIR 1.0 and later

AIR supports the following types of menus:

Context menus Context menus open in response to a right-click or command-click on an interactive object in SWF content or a document element in HTML content.

In the Flash Player runtime, a context menu is automatically displayed. You can use the ContextMenu and ContextMenuItem classes to add your own commands to the menu. You can also remove some, but not all, of the built-in commands.

Working with menus

In the AIR runtime, you can create a context menu using either the `NativeMenu` or the `ContextMenu` class. In HTML content in AIR, you can use the Webkit HTML and JavaScript APIs to add context menus to HTML elements.

Application menus (AIR only) An application menu is a global menu that applies to the entire application. Application menus are supported on Mac OS X, but not on Windows or Linux. On Mac OS X, the operating system automatically creates an application menu. You can use the AIR menu API to add items and submenus to the standard menus. You can add listeners for handling the existing menu commands. Or you can remove existing items.

Window menus (AIR only) A window menu is associated with a single window and is displayed below the title bar. Menus can be added to a window by creating a `NativeMenu` object and assigning it to the `menu` property of the `NativeWindow` object. Window menus are supported on the Windows and Linux operating systems, but not on Mac OS X. Native window menus can only be used with windows that have system chrome.

Dock and system tray icon menus (AIR only) These icon menus are similar to context menus and are assigned to an application icon in the Mac OS X dock or the Windows and Linux notification areas on the taskbar. Dock and system tray icon menus use the `NativeMenu` class. On Mac OS X, the items in the menu are added above the standard operating system items. On Windows or Linux, there is no standard menu.

Pop-up menus (AIR only) An AIR pop-up menu is like a context menu, but is not necessarily associated with a particular application object or component. Pop-up menus can be displayed anywhere in a window by calling the `display()` method of any `NativeMenu` object.

Custom menus Native menus are drawn entirely by the operating system and, as such, exist outside the Flash and HTML rendering models. Instead of using native menus, you can always create your own custom, non-native menus using MXML, ActionScript, or JavaScript (in AIR). Such menus must be fully rendered inside application content.

Default menus (AIR only)

The following default menus are provided by the operating system or a built-in AIR class:

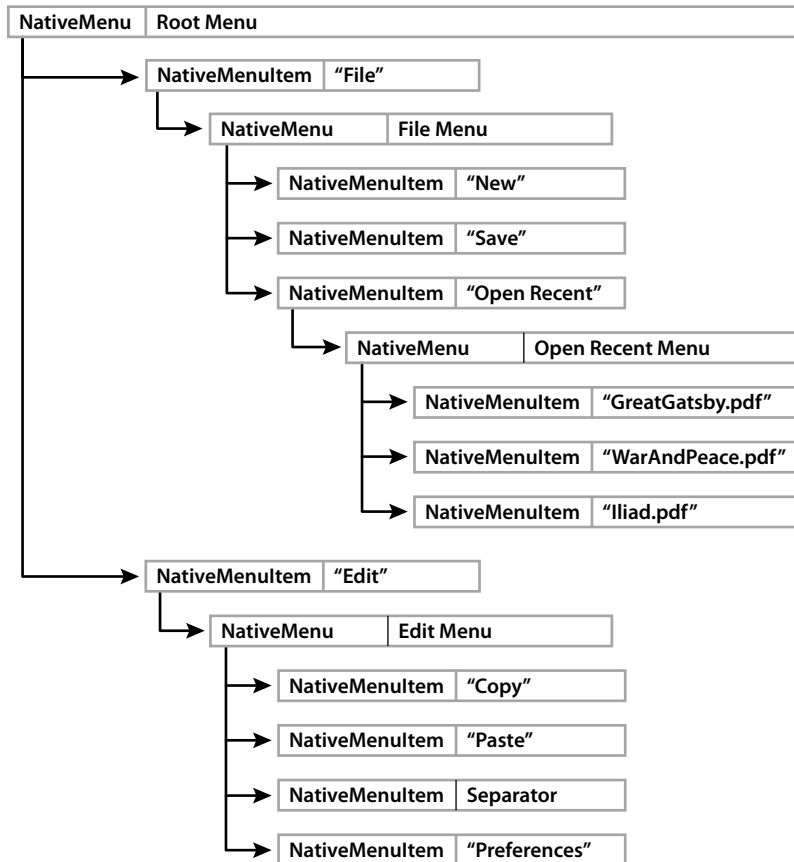
- Application menu on Mac OS X
- Dock icon menu on Mac OS X
- Context menu for selected text and images in HTML content
- Context menu for selected text in a `TextField` object (or an object that extends `TextField`)

Native menu structure (AIR)**Adobe AIR 1.0 and later**

Native menus are hierarchical in nature. `NativeMenu` objects contain child `NativeMenuItem` objects.

`NativeMenuItem` objects that represent submenus, in turn, can contain `NativeMenu` objects. The top- or root-level menu object in the structure represents the menu bar for application and window menus. (Context, icon, and pop-up menus don't have a menu bar).

The following diagram illustrates the structure of a typical menu. The root menu represents the menu bar and contains two menu items referencing a *File* submenu and an *Edit* submenu. The File submenu in this structure contains two command items and an item that references an *Open Recent Menu* submenu, which, itself, contains three items. The Edit submenu contains three commands and a separator.



Defining a submenu requires both a `NativeMenu` and a `NativeMenuItem` object. The `NativeMenuItem` object defines the label displayed in the parent menu and allows the user to open the submenu. The `NativeMenu` object serves as a container for items in the submenu. The `NativeMenuItem` object references the `NativeMenu` object through the `NativeMenuItem submenu` property.

To view a code example that creates this menu see [“Native menu example: Window and application menu \(AIR\)”](#) on page 123.

Menu events

Adobe AIR 1.0 and later

`NativeMenu` and `NativeMenuItem` objects both dispatch `preparing`, `displaying`, and `select` events:

Preparing: Whenever the object is about to begin a user interaction, the menu and its menu items dispatch a `preparing` event to any registered listeners. Interaction includes opening the menu or selecting an item with a keyboard shortcut.

Note: The `preparing` event is available only for Adobe AIR 2.6 and later.

Displaying: Immediately before a menu is displayed, the menu and its menu items dispatch a `displaying` event to any registered listeners.

The `preparing` and `displaying` events give you an opportunity to update the menu contents or item appearance before it is shown to the user. For example, in the listener for the `displaying` event of an “Open Recent” menu, you could change the menu items to reflect the current list of recently viewed documents.

If you remove the menu item whose keyboard shortcut triggered a `preparing` event, the menu interaction is effectively canceled and a `select` event is not dispatched.

The `target` and `currentTarget` properties of the event are both the object on which the listener is registered: either the menu itself, or one of its items.

The `preparing` event is dispatched before the `displaying` event. You typically listen for one event or the other, not both.

Select: When a command item is chosen by the user, the item dispatches a `select` event to any registered listeners. Submenu and separator items cannot be selected and so never dispatch a `select` event.

A `select` event bubbles up from a menu item to its containing menu, on up to the root menu. You can listen for `select` events directly on an item and you can listen higher up in the menu structure. When you listen for the `select` event on a menu, you can identify the selected item using the event `target` property. As the event bubbles up through the menu hierarchy, the `currentTarget` property of the event object identifies the current menu object.

Note: `ContextMenu` and `ContextMenuItem` objects dispatch `menuItemSelect` and `menuSelect` events as well as `select`, `preparing`, and `displaying` events.

Key equivalents for native menu commands (AIR)

Adobe AIR 1.0 and later

You can assign a key equivalent (sometimes called an accelerator) to a menu command. The menu item dispatches a `select` event to any registered listeners when the key, or key combination is pressed. The menu containing the item must be part of the menu of the application or the active window for the command to be invoked.

Key equivalents have two parts, a string representing the primary key and an array of modifier keys that must also be pressed. To assign the primary key, set the menu item `keyEquivalent` property to the single character string for that key. If you use an uppercase letter, the shift key is added to the modifier array automatically.

On Mac OS X, the default modifier is the command key (`Keyboard.COMMAND`). On Windows and Linux, it is the control key (`Keyboard.CONTROL`). These default keys are automatically added to the modifier array. To assign different modifier keys, assign a new array containing the desired key codes to the `keyEquivalentModifiers` property. The default array is overwritten. Whether you use the default modifiers or assign your own modifier array, the shift key is added if the string you assign to the `keyEquivalent` property is an uppercase letter. Constants for the key codes to use for the modifier keys are defined in the `Keyboard` class.

The assigned key equivalent string is automatically displayed beside the menu item name. The format depends on the user's operating system and system preferences.

Note: If you assign the `Keyboard.COMMAND` value to a key modifier array on the Windows operating system, no key equivalent is displayed in the menu. However, the control key must be used to activate the menu command.

The following example assigns `Ctrl+Shift+G` as the key equivalent for a menu item:

```
var item = new air.NativeMenuItem("Ungroup");  
item.keyEquivalent = "G";
```

This example assigns `Ctrl+Shift+G` as the key equivalent by setting the modifier array directly:

```
var item = new air.NativeMenuItem("Ungroup");
item.keyEquivalent = "G";
item.keyEquivalentModifiers = [air.Keyboard.CONTROL];
```

Note: Key equivalents are only triggered for application and window menus. If you add a key equivalent to a context or pop-up menu, the key equivalent is displayed in the menu label, but the associated menu command is never invoked.

Mnemonics (AIR)

Adobe AIR 1.0 and later

Mnemonics are part of the operating system keyboard interface to menus. Linux, Mac OS X, and Windows allow users to open menus and select commands with the keyboard, but there are subtle differences.

On Mac OS X, the user types the first letter or two of the menu or command and then presses the return key. The `mnemonicIndex` property is ignored.

On Windows, only a single letter is significant. By default, the significant letter is the first character in the label, but if you assign a mnemonic to the menu item, then the significant character becomes the designated letter. If two items in a menu have the same significant character (whether or not a mnemonic has been assigned), then the user's keyboard interaction with the menu changes slightly. Instead of pressing a single letter to select the menu or command, the user must press the letter as many times as necessary to highlight the desired item and then press the enter key to complete the selection. To maintain a consistent behavior, you should assign a unique mnemonic to each item in a menu for window menus.

On Linux, no default mnemonic is provided. You must specify a value for the `mnemonicIndex` property of a menu item to provide a mnemonic.

Specify the mnemonic character as an index into the label string. The index of the first character in a label is 0. Thus, to use "r" as the mnemonic for a menu item labeled, "Format," you would set the `mnemonicIndex` property equal to 2.

```
var item = new air.NativeMenuItem("Format");
item.mnemonicIndex = 2;
```

Menu item state

Adobe AIR 1.0 and later

Menu items have the two state properties, `checked` and `enabled`:

checked Set to `true` to display a check mark next to the item label.

```
var item = new air.NativeMenuItem("Format");
item.checked = true;
```

enabled Toggle the value between `true` and `false` to control whether the command is enabled. Disabled items are visually "grayed-out" and do not dispatch `select` events.

```
var item = new air.NativeMenuItem("Format");
item.enabled = false;
```

Attaching an object to a menu item

Adobe AIR 1.0 and later

The `data` property of the `NativeMenuItem` class allows you to reference an arbitrary object in each item. For example, in an “Open Recent” menu, you could assign the `File` object for each document to each menu item.

```
var file = air.File.applicationStorageDirectory.resolvePath("GreatGatsby.pdf")
var menuItem = docMenu.addItem(new air.NativeMenuItem(file.name));
menuItem.data = file;
```

Creating native menus (AIR)

Adobe AIR 1.0 and later

This topic describes how to create the various types of native menu supported by AIR.

Creating a root menu object

Adobe AIR 1.0 and later

To create a `NativeMenu` object to serve as the root of the menu, use the `NativeMenu` constructor:

```
var root = new air.NativeMenu();
```

For application and window menus, the root menu represents the menu bar and should only contain items that open submenus. Context menu and pop-up menus do not have a menu bar, so the root menu can contain commands and separator lines as well as submenus.

After the menu is created, you can add menu items. Items appear in the menu in the order in which they are added, unless you add the items at a specific index using the `addItemAt()` method of a menu object.

Assign the menu as an application, window, or icon menu, or display it as a pop-up menu, as shown in the following sections:

Setting the application menu or window menu

It's important that your code accommodate both application menus (supported on Mac OS) and window menus (supported on other operating systems)

```
var root = new air.NativeMenu();
if (air.NativeApplication.supportsMenu)
{
    air.NativeApplication.nativeApplication.menu = root;
}
else if (NativeWindow.supportsMenu)
{
    nativeWindow.menu = root;
}
```

Note: Mac OS defines a menu containing standard items for every application. Assigning a new `NativeMenu` object to the `menu` property of the `NativeApplication` object replaces the standard menu. You can also use the standard menu instead of replacing it.

The Adobe Flex provides a `FlexNativeMenu` class for easily creating menus that work across platforms. If you are using the Flex Framework, use the `FlexNativeMenu` classes instead of the `NativeMenu` class.

Setting a dock icon menu or system tray icon menu

```
air.NativeApplication.nativeApplication.icon.menu = root;
```

Note: Mac OS X defines a standard menu for the application dock icon. When you assign a new `NativeMenu` to the `menu` property of the `DockIcon` object, the items in that menu are displayed above the standard items. You cannot remove, access, or modify the standard menu items.

Displaying a menu as a pop-up

```
root.display(window.nativeWindow.stage, x, y);
```

More Help topics

[Developing cross-platform AIR applications](#)

Creating a submenu

Adobe AIR 1.0 and later

To create a submenu, you add a `NativeMenuItem` object to the parent menu and then assign the `NativeMenu` object defining the submenu to the item's `submenu` property. AIR provides two ways to create submenu items and their associated menu object:

You can create a menu item and its related menu object in one step with the `addSubMenu()` method:

```
var editMenuItem = root.addSubMenu(new air.NativeMenu(), "Edit");
```

You can also create the menu item and assign the menu object to its `submenu` property separately:

```
var editMenuItem = root.addItem("Edit", false);  
editMenuItem.submenu = new air.NativeMenu();
```

Creating a menu command

Adobe AIR 1.0 and later

To create a menu command, add a `NativeMenuItem` object to a menu and add an event listener referencing the function implementing the menu command:

```
var copy = new air.NativeMenuItem("Copy", false);  
copy.addEventListener(air.Event.SELECT, onCopyCommand);  
editMenu.addItem(copy);
```

You can listen for the `select` event on the command item itself (as shown in the example), or you can listen for the `select` event on a parent menu object.

Note: Menu items that represent submenus and separator lines do not dispatch `select` events and so cannot be used as commands.

Creating a menu separator line

Adobe AIR 1.0 and later

To create a separator line, create a `NativeMenuItem`, setting the `isSeparator` parameter to `true` in the constructor. Then add the separator item to the menu in the correct location:

```
var separatorA = new air.NativeMenuItem("A", true);  
editMenu.addItem(separatorA);
```

The label specified for the separator, if any, is not displayed.

About context menus in HTML (AIR)

Adobe AIR 1.0 and later

In HTML content displayed using the `HTMLLoader` object, the `contextmenu` event can be used to display a context menu. By default, a context menu is displayed automatically when the user invokes the context menu event on selected text (by right-clicking or command-clicking the text). To prevent the default menu from opening, listen for the `contextmenu` event and call the event object's `preventDefault()` method:

```
function showContextMenu(event) {  
    event.preventDefault();  
}
```

You can then display a custom context menu using DHTML techniques or by displaying an AIR native context menu. The following example displays a native context menu by calling the `menu.display()` method in response to the HTML `contextmenu` event:

```
<html>
<head>
<script src="AIRAliases.js" language="JavaScript" type="text/javascript"></script>
<script language="javascript" type="text/javascript">

function showContextMenu(event) {
    event.preventDefault();
    contextMenu.display(window.nativeWindow.stage, event.clientX, event.clientY);
}

function createContextMenu() {
    var menu = new air.NativeMenu();
    var command = menu.addItem(new air.NativeMenuItem("Custom command"));
    command.addEventListener(air.Event.SELECT, onCommand);
    return menu;
}

function onCommand() {
    air.trace("Context command invoked.");
}

var contextMenu = createContextMenu();
</script>
</head>
<body>
<p oncontextmenu="showContextMenu(event)" style="-khtml-user-select:auto;">Custom context
menu.</p>
</body>
</html>
```

Displaying pop-up native menus (AIR)

Adobe AIR 1.0 and later

You can display any `NativeMenu` object at an arbitrary time and location above a window, by calling the menu `display()` method. The method requires a reference to the stage; thus, only content in the application sandbox can display a menu as a pop-up.

The following method displays the menu defined by a `NativeMenu` object named `popupMenu` in response to a mouse click:

```
function onMouseClick(event) {
    popupMenu.display(window.nativeWindow.stage, event.clientX, event.clientY);
}
```

Note: The menu does not need to be displayed in direct response to an event. Any method can call the `display()` function.

Handling menu events

Flash Player 9 and later, Adobe AIR 1.0 and later

A menu dispatches events when the user selects the menu or when the user selects a menu item.

Events summary for menu classes

Flash Player 9 and later, Adobe AIR 1.0 and later

Add event listeners to menus or individual items to handle menu events.

Object	Events dispatched
NativeMenu (AIR)	Event.PREPARING (Adobe AIR 2.6 and later) Event.DISPLAYING Event.SELECT (propagated from child items and submenus)
NativeMenuItem (AIR)	Event.PREPARING (Adobe AIR 2.6 and later) Event.SELECT Event.DISPLAYING (propagated from parent menu)

Select menu events

Adobe AIR 1.0 and later

To handle a click on a menu item, add an event listener for the `select` event to the `NativeMenuItem` object:

```
var menuCommandX = new NativeMenuItem("Command X");  
menuCommand.addEventListener(air.Event.SELECT, doCommandX)
```

Because `select` events bubble up to the containing menus, you can also listen for `select` events on a parent menu. When listening at the menu level, you can use the event object `target` property to determine which menu command was selected. The following example traces the label of the selected command:


```
var colorMenuItem = new air.NativeMenuItem("Choose a color");
var colorMenu = new air.NativeMenu();
colorMenuItem.submenu = colorMenu;

var red = new air.NativeMenuItem("Red");
var green = new air.NativeMenuItem("Green");
var blue = new air.NativeMenuItem("Blue");
colorMenu.addItem(red);
colorMenu.addItem(green);
colorMenu.addItem(blue);

if (air.NativeApplication.supportsMenu) {
    air.NativeApplication.nativeApplication.menu.addItem(colorMenuItem);
    air.NativeApplication.nativeApplication.menu.addEventListener(air.Event.SELECT,
                                                                    colorChoice);
} else if (air.NativeWindow.supportsMenu) {
    var windowMenu = new air.NativeMenu();
    window.nativeWindow.menu = windowMenu;
    windowMenu.addItem(colorMenuItem);
    windowMenu.addEventListener(air.Event.SELECT, colorChoice);
}

function colorChoice(event) {
    var menuItem = event.target;
    air.trace(menuItem.label + " has been selected");
}
```

If you are using the `ContextMenuItem` class, you can listen for either the `select` event or the `menuItemSelect` event. The `menuItemSelect` event gives you additional information about the object owning the context menu, but does not bubble up to the containing menus.

Displaying menu events

Adobe AIR 1.0 and later

To handle the opening of a menu, you can add a listener for the `displaying` event, which is dispatched before a menu is displayed. You can use the `displaying` event to update the menu, for example by adding or removing items, or by updating the enabled or checked states of individual items. You can also listen for the `menuSelect` event from a `ContextMenu` object.

In AIR 2.6 and later, you can use the `preparing` event to update a menu in response to either displaying a menu or selecting an item with a keyboard shortcut.

Native menu example: Window and application menu (AIR)

Adobe AIR 1.0 and later

The following example creates the menu shown in “[Native menu structure \(AIR\)](#)” on page 114.

The menu is designed to work both on Windows, for which only window menus are supported, and on Mac OS X, for which only application menus are supported. To make the distinction, the `MenuExample` class constructor checks the static `supportsMenu` properties of the `NativeWindow` and `NativeApplication` classes. If `NativeWindow.supportsMenu` is `true`, then the constructor creates a `NativeMenu` object for the window and then creates and adds the File and Edit submenus. If `NativeApplication.supportsMenu` is `true`, then the constructor creates and adds the File and Edit menus to the existing menu provided by the Mac OS X operating system.

The example also illustrates menu event handling. The `select` event is handled at the item level and also at the menu level. Each menu in the chain from the menu containing the selected item to the root menu responds to the `select` event. The `displaying` event is used with the "Open Recent" menu. Just before the menu is opened, the items in the menu are refreshed from the recent Documents array (which doesn't actually change in this example). Although not shown in this example, you can also listen for `displaying` events on individual items.

```
<html>
<head>
<script src="AIRAliases.js" type="text/javascript"></script>
<script type="text/javascript">
var application = air.NativeApplication.nativeApplication;
var recentDocuments =
    new Array(new air.File("app-storage:/GreatGatsby.pdf"),
              new air.File("app-storage:/WarAndPeace.pdf"),
              new air.File("app-storage:/Iliad.pdf"));

function MenuExample() {
    var fileMenu;
    var editMenu;

    if (air.NativeWindow.supportsMenu &&
        nativeWindow.systemChrome != air.NativeWindowSystemChrome.NONE) {
        nativeWindow.menu = new air.NativeMenu();
        nativeWindow.menu.addEventListener(air.Event.SELECT, selectCommandMenu);
        fileMenu = nativeWindow.menu.addItem(new air.NativeMenuItem("File"));
        fileMenu.submenu = createFileMenu();

        editMenu = nativeWindow.menu.addItem(new air.NativeMenuItem("Edit"));
        editMenu.submenu = createEditMenu();
    }

    if (air.NativeApplication.supportsMenu) {
        application.menu.addEventListener(air.Event.SELECT, selectCommandMenu);
        fileMenu = application.menu.addItem(new air.NativeMenuItem("File"));
        fileMenu.submenu = createFileMenu();
        editMenu = application.menu.addItem(new air.NativeMenuItem("Edit"));
        editMenu.submenu = createEditMenu();
    }
}

function createFileMenu() {
    var fileMenu = new air.NativeMenu();
    fileMenu.addEventListener(air.Event.SELECT, selectCommandMenu);

    var newCommand = fileMenu.addItem(new air.NativeMenuItem("New"));
    newCommand.addEventListener(air.Event.SELECT, selectCommand);
    var saveCommand = fileMenu.addItem(new air.NativeMenuItem("Save"));
    saveCommand.addEventListener(air.Event.SELECT, selectCommand);
    var openFile = fileMenu.addItem(new air.NativeMenuItem("Open Recent"));
}
```

```
    openFile.submenu = new air.NativeMenu();
    openFile.submenu.addEventListener(air.Event.DISPLAYING, updateRecentDocumentMenu);
    openFile.submenu.addEventListener(air.Event.SELECT, selectCommandMenu);

    return fileMenu;
}

function createEditMenu() {
    var editMenu = new air.NativeMenu();
    editMenu.addEventListener(air.Event.SELECT, selectCommandMenu);

    var copyCommand = editMenu.addItem(new air.NativeMenuItem("Copy"));
    copyCommand.addEventListener(air.Event.SELECT, selectCommand);
    copyCommand.keyEquivalent = "c";
    var pasteCommand = editMenu.addItem(new air.NativeMenuItem("Paste"));
    pasteCommand.addEventListener(air.Event.SELECT, selectCommand);
    pasteCommand.keyEquivalent = "v";
    editMenu.addItem(new air.NativeMenuItem("", true));
    var preferencesCommand = editMenu.addItem(new air.NativeMenuItem("Preferences"));
    preferencesCommand.addEventListener(air.Event.SELECT, selectCommand);

    return editMenu;
}

function updateRecentDocumentMenu(event) {
    air.trace("Updating recent document menu.");
    var docMenu = air.NativeMenu(event.target);

    for (var i = docMenu.numItems - 1; i >= 0; i--) {
        docMenu.removeItemAt(i);
    }

    for (var file in recentDocuments) {
        var menuItem =
            docMenu.addItem(new air.NativeMenuItem(recentDocuments[file].name));
        menuItem.data = recentDocuments[file];
        menuItem.addEventListener(air.Event.SELECT, selectRecentDocument);
    }
}

function selectRecentDocument(event) {
    air.trace("Selected recent document: " + event.target.data.name);
}

function selectCommand(event) {
    air.trace("Selected command: " + event.target.label);
}

function selectCommandMenu(event) {
    if (event.currentTarget.parent != null) {
        var menuItem = findItemForMenu(event.currentTarget);
        if (menuItem != null) {
            air.trace("Select event for \"" + event.target.label +
                "\" command handled by menu: " + menuItem.label);
        }
    } else {
        air.trace("Select event for \"" + event.target.label +
```

```
        "\" command handled by root menu.");
    }
}

function findItemForMenu(menu) {
    for (var item in menu.parent.items) {
        if (item != null) {
            if (item.submenu == menu) {
                return item;
            }
        }
    }
    return null;
}
</script>
<title>AIR menus</title>
</head>
<body onload="MenuExample()"></body>
</html>
```

Using the MenuBuilder framework

Adobe AIR 1.0 and later

In addition to the standard menu classes, Adobe AIR includes a menu builder JavaScript framework to make it easier for developers to create menus. The MenuBuilder framework allows you to define the structure of your menus declaratively in XML or JSON format. It also provides helper methods for creating any of the menu types available to an AIR application. For a complete list of the ways a native menu can be used in AIR, see “[Menu basics](#)” on page 113.

Creating a menu with the MenuBuilder framework

Adobe AIR 1.0 and later

The MenuBuilder framework allows you to define the structure of a menu using XML or JSON. The framework includes methods for loading and parsing the file containing the menu structure. Once a menu structure is loaded, additional methods allow you to designate how the menu is used in the application. The methods allow you to set the menu as the Mac OS X application menu, as a window menu, or as a context menu.

The MenuBuilder framework is not built in to the runtime. To use the framework, include the AIRMenuBuilder.js file (included with the Adobe AIR SDK) in your application code, as shown here:

```
<script type="text/javascript" src="AIRMenuBuilder.js"></script>
```

The MenuBuilder framework is designed to run in the application sandbox. The framework methods can't be called from the classic sandbox.

All the framework methods that are for developer use are defined as class methods on the `air.ui.Menu` class.

MenuBuilder basic workflow

Adobe AIR 1.0 and later

In general, regardless of the type of menu you want to create, you follow three steps to create a menu with the MenuBuilder framework:

- 1 Define the menu structure:** Create a file containing XML or JSON that defines the menu structure. For some menu types, the top-level menu items are menus (for example in a window or application menu). For other menu types, the top-level items are individual menu commands (such as in a context menu). For details on the format for defining menu structure, see [“Defining MenuBuilder menu structure”](#) on page 129.
- 2 Load the menu structure:** Call the appropriate Menu class method, either `Menu.createFromXML()` or `Menu.createFromJSON()`, to load the menu structure file and parse it into an actual menu object. Either method returns a `NativeMenu` object that can be passed to one of the framework's menu-setting methods.
- 3 Assign the menu:** Call the appropriate Menu class method according to how the menu is used. The options are:
 - `Menu.setAsMenu()` for a window or application menu
 - `Menu.setAsContextMenu()` to display the menu as a context menu for a DOM element
 - `Menu.setAsIconMenu()` to set the menu as the context menu for a system tray or dock icon

The timing of when the code executes can be important. In particular, a window menu must be assigned before the actual operating system window is created. Any `setAsMenu()` call that sets a menu as a window menu must execute directly in the HTML page rather than in the `onload` or other event handler. The code to create the menu must run before the operating system opens the window. At the same time, any `setAsContextMenu()` call that refers to a DOM element must occur after the DOM element is created. The safest approach is to place the `<script>` block containing the menu assignment code just inside the closing `</body>` tag at the end of the HTML page.

Loading menu structure

Adobe AIR 1.0 and later

Regardless of the intended use of your menu, you define the structure of the menu as a separate file containing an XML or JSON structure. Before you can assign a menu in your application, first use the framework to load and parse the menu structure file. To load and parse a menu structure file, use one of these two framework methods:

- `Menu.createFromXML()` to load and parse an XML-formatted menu structure file
- `Menu.createFromJSON()` to load and parse a JSON-formatted menu structure file

Both methods accept one argument: the file path of the menu structure file. Both methods load the file from that location. They parse the file contents and return a `NativeMenu` object with the menu structure defined in the file. For example, the following code loads a menu structure file named “windowMenu.xml” that’s in the same directory as the HTML file that’s loading it:

```
var windowMenu = air.ui.Menu.createFromXML("windowMenu.xml");
```

In the next example, the code loads a menu structure file named “contextMenu.js” from a directory named “menus”:

```
var contextMenu = air.ui.Menu.createFromJSON("menus/contextMenu.js");
```

Note: The generated `NativeMenu` object can only be used once as an application or window menu. However, a generated `NativeMenu` object can be used multiple times in an application as a context or icon menu. Using the MenuBuilder framework on Mac OS X, if the same `NativeMenu` is assigned as the application menu and also as another type of menu, it is only used as the application menu.

For details of the specific menu structure that the MenuBuilder framework accepts, see [“Defining MenuBuilder menu structure”](#) on page 129.

Creating an application or window menu

Adobe AIR 1.0 and later

When you create an application or window menu using the MenuBuilder framework, the top-level objects or nodes in the menu data structure correspond to the items that show up in the menu bar. Items nested inside one of those top-level items define the individual menu commands. Likewise, those menu items can contain other items. In that case the menu item is a submenu rather than a command. When the user selects the menu item it expands its own menu of items.

You use the `Menu.setAsMenu()` method to set a menu as the application menu or window menu for the window in which the call executes. The `setAsMenu()` method takes one parameter: the `NativeMenu` object to use. The following example loads an XML file and sets the generated menu as the application or window menu:

```
var windowMenu = air.ui.Menu.createFromXML("windowMenu.xml");
air.ui.Menu.setAsMenu(windowMenu);
```

On an operating system that supports window menus, the `setAsMenu()` call sets the menu as the window menu for the current window (the window that's represented as `window.nativeWindow`). On an operating system that supports an application menu, the menu is used as the application menu.

Mac OS X defines a set of standard menus as the default application menu, with the same set of menu items for every application. These menus include an application menu whose name matches the application name, an Edit menu, and a Window menu. When you assign a `NativeMenu` object as the application menu by calling the `Menu.setAsMenu()` method, the items in the `NativeMenu` are inserted into the standard menu structure between the Edit and Window menus. The standard menus are not modified or replaced.

You can replace the standard menus rather than supplement them if you prefer. To replace the existing menu, pass a second argument with the value `true` to the `setAsMenu()` call, as in this example:

```
air.ui.Menu.setAsMenu(windowMenu, true);
```

Creating a DOM element context menu

Adobe AIR 1.0 and later

Creating a context menu for a DOM element using the MenuBuilder framework involves two steps. First you create the `NativeMenu` instance that defines the menu structure using the `Menu.createFromXML()` or `Menu.createFromJSON()` method. You then assign that menu as the context menu for a DOM element by calling the `Menu.setAsContextMenu()` method. Because a context menu consists of a single menu, the top-level menu items in the menu data structure serve as the items in the single menu. Any menu item that contains child menu items defines a submenu. To assign a `NativeMenu` as the context menu for a DOM element, call the `Menu.setAsContextMenu()` method. This method requires two parameters: the `NativeMenu` to set as the context menu, and the id (a string) of the DOM element to which it is assigned:

```
var treeContextMenu = air.ui.Menu.createFromXML("treeContextMenu.xml");
air.ui.Menu.setAsContextMenu(treeContextMenu, "navTree");
```

If you omit the DOM element parameter, the method uses the HTML document from which the method is called as the default value. In other words, the menu is set as the context menu for the HTML document's entire window. This technique is convenient for removing the default context menu from an entire HTML window by passing `null` for the first parameter, as in this example:

```
air.ui.Menu.setAsContextMenu(null);
```

You can also remove an assigned context menu from any DOM element. Call the `setAsContextMenu()` method and pass `null` and the element id as the two arguments.

Creating an icon context menu

Adobe AIR 1.0 and later

In addition to context menus for DOM elements within an application window, an Adobe AIR application supports two other special context menus: dock icon menus for operating systems that support a dock, and system tray icon menus for operating systems that use a system tray. To set either of these menus, you first create a `NativeMenu` using the `Menu.createFromXML()` or `Menu.createFromJSON()` method. Then you assign the `NativeMenu` as the dock or system tray icon menu by calling the `Menu.setAsIconMenu()` method.

This method accepts two arguments. The first argument, which is required, is the `NativeMenu` to use as the icon menu. The second argument is an `Array` containing strings that are file paths to images to use as the icon, or `BitmapData` objects containing image data for the icon. This argument is required unless default icons are specified in the application.xml file. If default icons are specified in the application.xml file, those icons are used by default for the system tray icon.

The following example demonstrates loading menu data and assigning the menu as the dock or system tray icon context menu:

```
// Assumes that icons are specified in the application.xml file.  
// Otherwise the icons would need to be specified using a second  
// parameter to the setAsIconMenu() function.  
var iconMenu = air.ui.Menu.createFromXML("iconMenu.xml");  
air.ui.Menu.setAsIconMenu(iconMenu);
```

Note: Mac OS X defines a standard context menu for the application dock icon. When you assign a menu as the dock icon context menu, the items in the menu are displayed above the standard OS menu items. You cannot remove, access, or modify the standard menu items.

Defining MenuBuilder menu structure

Adobe AIR 1.0 and later

When you create a `NativeMenu` object using the `Menu.createFromXML()` or `Menu.createFromJSON()` method, the structure of XML elements or objects defines the structure of the resulting menu. Once the menu is created, you can change its structure or properties at run time. To change a menu item at run time you access the `NativeMenuItem` object by navigating through the `NativeMenu` object's hierarchy.

The `MenuBuilder` framework looks for certain XML attributes or object properties as it parses through the menu data source. The presence and value of those attributes or properties determines the structure of the menu that's created.

When you use XML for the menu structure, the XML file must contain a root node. The child nodes of the root node are used as the top-level menu item nodes. The XML nodes can have any name. The names of the XML nodes don't affect the menu structure. Only the hierarchical structure of the nodes and their attribute values are used to define the menu.

Menu item types

Adobe AIR 1.0 and later

Each entry in the menu data source (each XML element or JSON object) can specify an item type and type-specific information about the menu item it represents. Adobe AIR supports the following menu item types, which can be set as the values of the `type` attribute or property in the data source:

Menu item type	Description
normal	The default type. Selecting an item with the normal type triggers a <code>select</code> event and calls the function specified in the <code>onSelect</code> field of the data source. Alternatively, if the item has children, the menu item dispatches a <code>preparing</code> event, then a <code>displaying</code> event and then opens the submenu.
check	Selecting an item with the <code>check</code> type toggles the <code>NativeMenuItem</code> 's <code>checked</code> property between <code>true</code> and <code>false</code> values, triggers a <code>select</code> event, and calls the function specified in the <code>onSelect</code> field of the data source. When the menu item is in the <code>true</code> state, it displays a check mark in the menu next to the item's label.
separator	Items with the <code>separator</code> type provide a simple horizontal line that divides the items in the menu into different visual groups.

A normal menu item is treated as a submenu if it has children. With an XML data source, this means that the menu item element contains other XML elements. For a JSON data source, give the object representing the menu item a property named `items` containing an array of other objects.

Menu data source attributes or properties

Adobe AIR 1.0 and later

Items in the menu data source can specify several XML attributes or object properties that determine how the item is displayed and behaves. The following table lists the attributes you can specify, their data types, their purposes, and how the data source must represent them:

Attribute or property	Type	Description
<code>altKey</code>	Boolean	Specifies whether the Alt key is required as part of the key equivalent for the item.
<code>cmdKey</code>	Boolean	Specifies whether the Command key is required as part of the key equivalent for the item. The <code>defaultKeyEquivalentModifiers</code> field also affects this value.
<code>ctrlKey</code>	Boolean	Specifies whether the Control key is required as part of the key equivalent for the item. The <code>defaultKeyEquivalentModifiers</code> field also affects this value.
<code>defaultKeyEquivalentModifiers</code>	Boolean	Specifies whether the operating system default modifier key (Command for Mac OS X and Control for Windows) is required as part of the key equivalent for the item. If not specified, the <code>MenuBuilder</code> framework treats the item as if the value was <code>true</code> .

Attribute or property	Type	Description
<code>enabled</code>	Boolean	Specifies whether the user can select the menu item (true), or not (false). If not specified, the MenuBuilder framework treats the item as if the value was <code>true</code> .
<code>items</code>	Array	(JSON only) specifies that the menu item is itself a menu. The objects in the array are the child menu items contained in the menu.
<code>keyEquivalent</code>	String	Specifies a keyboard character which, when pressed, triggers an event as though the menu item was selected. If this value is an uppercase character, the shift key is required as part of the key equivalent of the item.
<code>label</code>	String	Specifies the text that appears in the control. This item is used for all menu item types except <code>separator</code> .
<code>mnemonicIndex</code>	Integer	Specifies the index position of the character in the label that is used as the mnemonic for the menu item. Alternatively, you can indicate that a character in the label is the menu item's mnemonic by including an underscore immediately to the left of that character.
<code>onSelect</code>	String or Function	Specifies the name of a function (a String) or a reference to the function (a Function object). The specified function is called as an event listener when the user selects the menu item. For more information see "Handling MenuBuilder menu events" on page 136.
<code>shiftKey</code>	String	Specifies whether the Shift key is required as part of the key equivalent for the item. Alternatively, the <code>keyEquivalent</code> value specifies this value as well. If the <code>keyEquivalent</code> value is an uppercase letter, the shift key is required as part of the key equivalent.
<code>toggled</code>	Boolean	Specifies whether a check item is selected. If not specified, the MenuBuilder framework treats the item as if the value was <code>false</code> and the item is not selected.
<code>type</code>	String	Specifies the type of menu item. Meaningful values are <code>separator</code> and <code>check</code> . The MenuBuilder framework treats all other values, or elements or objects with no <code>type</code> entry, as normal menu entries.

The MenuBuilder framework ignores all other object properties or XML attributes.

Example: An XML MenuBuilder data source

Adobe AIR 1.0 and later

The following example uses the MenuBuilder framework to define a context menu for a region of text. It shows how to define the menu structure using XML as the data source. For an application that specifies an identical menu structure using a JSON array, see [“Example: A JSON MenuBuilder data source”](#) on page 133.

The application consists of two files.

The first file is the menu data source, in a file named “textContextMenu.xml.” While this example uses menu item nodes named “menuitem,” the actual name of the XML nodes doesn’t matter. As described previously, only the structure of the XML and the attribute values affect the structure of the generated menu.

```
<?xml version="1.0" encoding="utf-8" ?>
<root>
  <menuitem label="MenuItem A" />
  <menuitem label="MenuItem B" type="check" toggled="true" />
  <menuitem label="MenuItem C" enabled="false" />
  <menuitem type="separator" />
  <menuitem label="MenuItem D">
    <menuitem label="SubMenuItem D-1" />
    <menuitem label="SubMenuItem D-2" />
    <menuitem label="SubMenuItem D-3" />
  </menuitem>
</root>
```

The second file is the source code for the application user interface (the HTML file specified as the initial window in the application.xml file:

```
<html>
  <head>
    <title>XML-based menu data source example</title>
    <script type="text/javascript" src="AIRAliases.js"></script>
    <script type="text/javascript" src="AIRMenuBuilder.js"></script>
    <style type="text/css">
      #contextEnabledText
      {
        margin-left: auto;
        margin-right: auto;
        margin-top: 100px;
        width: 50%
      }
    </style>
  </head>
  <body>
    <div id="contextEnabledText">This block of text is context menu enabled. Right click
or Command-click on the text to view the context menu.</div>
    <script type="text/javascript">
      // Create a NativeMenu from "textContextMenu.xml" and set it
      // as context menu for the "contextEnabledText" DOM element:
      var textMenu = air.ui.Menu.createFromXML("textContextMenu.xml");
      air.ui.Menu.setAsContextMenu(textMenu, "contextEnabledText");

      // Remove the default context menu from the page:
      air.ui.Menu.setAsContextMenu(null);
    </script>
  </body>
</html>
```

Example: A JSON MenuBuilder data source

Adobe AIR 1.0 and later

The following example uses the MenuBuilder framework to define a context menu for a region of text using a JSON array as the data source. For an application that specifies an identical menu structure in XML, see [“Example: An XML MenuBuilder data source”](#) on page 132.

The application consists of two files.

The first file is the menu data source, in a file named “textContextMenu.js.”

```
[
  {label: "MenuItem A"},
  {label: "MenuItem B", type: "check", toggled: "true"},
  {label: "MenuItem C", enabled: "false"},
  {type: "separator"},
  {label: "MenuItem D", items:
    [
      {label: "SubMenuItem D-1"},
      {label: "SubMenuItem D-2"},
      {label: "SubMenuItem D-3"}
    ]
  }
]
```

The second file is the source code for the application user interface (the HTML file specified as the initial window in the application.xml file):

```
<html>
  <head>
    <title>JSON-based menu data source example</title>
    <script type="text/javascript" src="AIRAliases.js"></script>
    <script type="text/javascript" src="AIRMenuBuilder.js"></script>
    <style type="text/css">
      #contextEnabledText
      {
        margin-left: auto;
        margin-right: auto;
        margin-top: 100px;
        width: 50%
      }
    </style>
  </head>
  <body>
    <div id="contextEnabledText">This block of text is context menu enabled. Right click
or Command-click on the text to view the context menu.</div>
    <script type="text/javascript">
      // Create a NativeMenu from "textContextMenu.js" and set it
      // as context menu for the "contextEnabledText" DOM element:
      var textMenu = air.ui.Menu.createFromJSON("textContextMenu.js");
      air.ui.Menu.setAsContextMenu(textMenu, "contextEnabledText");

      // Remove the default context menu from the page:
      air.ui.Menu.setAsContextMenu(null);
    </script>
  </body>
</html>
```

Adding menu keyboard features with MenuBuilder

Adobe AIR 1.0 and later

Operating system native menus support the use of keyboard shortcuts, and these shortcuts are also available in Adobe AIR. Two of the types of keyboard shortcuts that can be specified in a menu data source are keyboard equivalents for menu commands and mnemonics.

Specifying menu keyboard equivalents

Adobe AIR 1.0 and later

You can specify a key equivalent (sometimes called an accelerator) for a window or application menu command. When the key or key combination is pressed the NativeMenuItem dispatches a `select` event and any `onSelect` event handler specified in the data source is called. The behavior is the same as though the user had selected the menu item.

For complete details about menu keyboard equivalents, see “[Key equivalents for native menu commands \(AIR\)](#)” on page 116.

Using the MenuBuilder framework, you can specify a keyboard equivalent for a menu item in its corresponding node in the data source. If the data source has a `keyEquivalent` field, the MenuBuilder framework uses that value as the key equivalent character.

Working with menus

You can also specify modifier keys that are part of the key equivalent combination. To add a modifier, specify `true` for the `altKey`, `ctrlKey`, `cmdKey`, or `shiftKey` field. The specified key or keys become part of the key equivalent combination. By default the Control key is specified for Windows and the Command key is specified for Mac OS X. To override this default behavior, include a `defaultKeyEquivalentModifiers` field set to `false`.

The following example shows the data structure for an XML-based menu data source that includes keyboard equivalents, in a file named "keyEquivalentMenu.xml":

```
<?xml version="1.0" encoding="utf-8" ?>
<root>
  <menuitem label="File">
    <menuitem label="New" keyEquivalent="n"/>
    <menuitem label="Open" keyEquivalent="o"/>
    <menuitem label="Save" keyEquivalent="s"/>
    <menuitem label="Save As..." keyEquivalent="s" shiftKey="true"/>
    <menuitem label="Close" keyEquivalent="w"/>
  </menuitem>
  <menuitem label="Edit">
    <menuitem label="Cut" keyEquivalent="x"/>
    <menuitem label="Copy" keyEquivalent="c"/>
    <menuitem label="Paste" keyEquivalent="v"/>
  </menuitem>
</root>
```

The following example application loads the menu structure from "keyEquivalentMenu.xml" and uses it as the structure for the window or application menu for the application:

```
<html>
  <head>
    <title>XML-based menu with key equivalents example</title>
    <script type="text/javascript" src="AIRAliases.js"></script>
    <script type="text/javascript" src="AIRMenuBuilder.js"></script>
  </head>
  <body>
    <script type="text/javascript">
      // Create a NativeMenu from "keyEquivalentMenu.xml" and set it
      // as the application/window menu
      var keyEquivMenu = air.ui.Menu.createFromXML("keyEquivalentMenu.xml");
      air.ui.Menu.setAsMenu(keyEquivMenu);
    </script>
  </body>
</html>
```

Specifying menu item mnemonics**Adobe AIR 1.0 and later**

A menu item mnemonic is a key associated with a menu item. When the key is pressed while the menu is displayed, the menu item command is triggered. The behavior is the same as if the user had selected the menu item with the mouse. Typically the operating system indicates a menu item mnemonic by underlining that character in the name of the menu item.

For more information about mnemonics, see "[Mnemonics \(AIR\)](#)" on page 117.

Working with menus

With the MenuBuilder framework, the simplest way to specify a mnemonic for a menu item is to include an underscore character (“_”) in the menu item’s `label` field. Place the underscore immediately to the left of the letter that serves as the mnemonic for that menu item. For example, if the following XML node is used in a data source that’s loaded using the MenuBuilder framework, the mnemonic for the command is the first character of the second word (the letter “A”):

```
<menuitem label="Save _As"/>
```

When the `NativeMenu` object is created, the underscore is not included in the label. Instead, the character following the underscore becomes the mnemonic for the menu item. To include a literal underscore character in a menu item’s name, use two underscore characters (“__”). This sequence is converted to an underscore in the menu item label.

As an alternative to using an underscore character in the `label` field, you can provide an integer index position for the mnemonic character. Specify the index in the `mnemonicIndex` field in the menu item data source object or XML element.

Handling MenuBuilder menu events

Adobe AIR 1.0 and later

User interaction with a `NativeMenu` is event-driven. When the user selects a menu item or opens a menu or submenu, the `NativeMenuItem` object dispatches an event. With a `NativeMenu` object created using the MenuBuilder framework, you can register event listeners with individual `NativeMenuItem` objects or with the `NativeMenu`. You subscribe and respond to these events the same way as if you had created the `NativeMenu` and `NativeMenuItem` objects manually rather than using the MenuBuilder framework. For more information see “[Menu events](#)” on page 115.

The MenuBuilder framework supplements the standard event handling, providing a way to specify a `select` event handler function for a menu item within the menu data source. If you specify an `onSelect` field in the menu item data source, the specified function is called when the user selects the menu item. For example, suppose the following XML node is included in a data source that’s loaded using the MenuBuilder framework. When the menu item is selected the function named `doSave()` is called:

```
<menuitem label="Save" onSelect="doSave"/>
```

The `onSelect` field is a `String` when it’s used with an XML data source. With a JSON array, the field can be a `String` with the name of the function. In addition, for a JSON array only, the field can also be a variable reference to the function as an object. However, if the JSON array uses a `Function` variable reference the menu must be created before or during the `onload` event handler or a JavaScript security violation occurs. In all cases, the specified function must be defined in the global scope.

When the specified function is called, the runtime passes two arguments to it. The first argument is the event object dispatched by the `select` event. It is an instance of the `Event` class. The second argument that’s passed to the function is an anonymous object containing the data that was used to create the menu item. This object has the following properties. Each property’s value matches the value in the original data structure or `null` if the property is not set in the original data structure:

- `altKey`
- `cmdKey`
- `ctrlKey`
- `defaultKeyEquivalentModifiers`
- `enabled`
- `keyEquivalent`

- label
- mnemonicIndex
- onSelect
- shiftKey
- toggled
- type

The following example lets you experiment with NativeMenu events. The example includes two menus. The window and application menu is created using an XML data source. The context menu for the list of items represented by the and elements is created using a JSON array data source. A text area on the screen displays information about each event as the user selects menu items.

The following listing is the source code of the application:

```
<html>
  <head>
    <title>Menu event handling example</title>
    <script type="text/javascript" src="AIRAliases.js"></script>
    <script type="text/javascript" src="AIRMenuBuilder.js"></script>
    <script type="text/javascript" src="printObject.js"></script>
    <script type="text/javascript">
      function fileMenuCommand(event, data) {
        print("fileMenuCommand", event, data);
      }

      function editMenuCommand(event, data) {
        print("editMenuCommand", event, data);
      }

      function moveItemUp(event, data) {
        print("moveItemUp", event, data);
      }

      function moveItemDown(event, data) {
        print("moveItemDown", event, data);
      }

      function print(command, event, data) {
        var result = "";
        result += "<h1>Command: " + command + '</h1>';
        result += "<p>" + printObject(event) + "</p>";
        result += "<p>Data:</p>";
        result += "<ul>";
        for (var s in data) {
          result += "<li>" + s + ": " + printObject(data[s]) + "</li>";
        }
        result += "</ul>";

        var o = document.getElementById("output");
        o.innerHTML = result;
      }
    </script>
    <style type="text/css">
      #contextList {
```

```
        position: absolute; left: 0; top: 25px; bottom: 0; width: 100px;
        background: #eaeaea;
    }
    #output {
        position: absolute; left: 125px; top: 25px; right: 0; bottom: 0;
    }
</style>
</head>
<body>
    <div id="contextList">
        <ul>
            <li>List item 1</li>
            <li>List item 2</li>
            <li>List item 3</li>
        </ul>
    </div>
    <div id="output">
        Choose menu commands. Information about the events displays here.
    </div>
    <script type="text/javascript">
        var mainMenu = air.ui.Menu.createFromXML("mainMenu.xml");
        air.ui.Menu.setAsMenu(mainMenu);

        var listContextMenu = air.ui.Menu.createFromJSON("listContextMenu.js");
        air.ui.Menu.setAsContextMenu(listContextMenu, "contextList")

        // clear the default context menu
        air.ui.Menu.setAsContextMenu(null);
    </script>
</body>
</html>
```

The following listing is the data source for the main menu ("mainMenu.xml"):

```
<?xml version="1.0" encoding="utf-8" ?>
<root>
    <menuitem label="File">
        <menuitem label="New" keyEquivalent="n" onSelect="fileMenuCommand"/>
        <menuitem label="Open" keyEquivalent="o" onSelect="fileMenuCommand"/>
        <menuitem label="Save" keyEquivalent="s" onSelect="fileMenuCommand"/>
        <menuitem label="Save As..." keyEquivalent="S" onSelect="fileMenuCommand"/>
        <menuitem label="Close" keyEquivalent="w" onSelect="fileMenuCommand"/>
    </menuitem>
    <menuitem label="Edit">
        <menuitem label="Cut" keyEquivalent="x" onSelect="editMenuCommand"/>
        <menuitem label="Copy" keyEquivalent="c" onSelect="editMenuCommand"/>
        <menuitem label="Paste" keyEquivalent="v" onSelect="editMenuCommand"/>
    </menuitem>
</root>
```

The following listing is the data source for the context menu ("listContextMenu.js");

```
[
    {label: "Move Item Up", onSelect: "moveItemUp"},
    {label: "Move Item Down", onSelect: "moveItemDown"}
]
```


The following listing contains the code from the `printObject.js` file. The file includes the `printObject()` function, which the application uses but which doesn't affect the operation of the menus in the example.

```
function printObject(obj) {
    if (!obj) {
        if (typeof obj == "undefined") { return "[undefined]"; };
        if (typeof obj == "object") { return "[null]"; };
        return "[false]";
    } else {
        if (typeof obj == "boolean") { return "[true]"; };
        if (typeof obj == "object") {
            if (typeof obj.length == "number") {
                var ret = [];
                for (var i=0; i<obj.length; i++) {
                    ret.push(printObject(obj[i]));
                }
                return "[" + ret.join(", ") + "]" + ".join(" ");
            } else {
                var ret = [];
                var hadChildren = false;
                for (var k in obj) {
                    hadChildren = true;
                    ret.push ([k, " => ", printObject(obj[k])]);
                }
                if (hadChildren) {
                    return ["{\n", ret.join(",\n"), "\n"}].join("");
                }
            }
        }
        if (typeof obj == "function") { return "[Function]"; }
        return String(obj);
    }
}
```

Chapter 10: Taskbar icons in AIR

Adobe AIR 1.0 and later

Many operating systems provide a taskbar, such as the Mac OS X dock, that can contain an icon to represent an application. Adobe® AIR® provides an interface for interacting with the application task bar icon through the `NativeApplication.nativeApplication.icon` property.

More Help topics

[flash.desktop.NativeApplication](#)

[flash.desktop.DockIcon](#)

[flash.desktop.SystemTrayIcon](#)

About taskbar icons

Adobe AIR 1.0 and later

AIR creates the `NativeApplication.nativeApplication.icon` object automatically. The object type is either `DockIcon` or `SystemTrayIcon`, depending on the operating system. You can determine which of these `InteractiveIcon` subclasses that AIR supports on the current operating system using the `NativeApplication.supportsDockIcon` and `NativeApplication.supportsSystemTrayIcon` properties. The `InteractiveIcon` base class provides the properties `width`, `height`, and `bitmaps`, which you can use to change the image used for the icon. However, accessing properties specific to `DockIcon` or `SystemTrayIcon` on the wrong operating system generates a runtime error.

To set or change the image used for an icon, create an array containing one or more images and assign it to the `NativeApplication.nativeApplication.icon.bitmaps` property. The size of taskbar icons can be different on different operating systems. To avoid image degradation due to scaling, you can add multiple sizes of images to the `bitmaps` array. If you provide more than one image, AIR selects the size closest to the current display size of the taskbar icon, scaling it only if necessary. The following example sets the image for a taskbar icon using two images:

```
air.NativeApplication.nativeApplication.icon.bitmaps =  
    [bmp16x16.bitmapData, bmp128x128.bitmapData];
```

To change the icon image, assign an array containing the new image or images to the `bitmaps` property. You can animate the icon by changing the image in response to an `enterFrame` or `timer` event.

To remove the icon from the notification area on Windows and Linux, or to restore the default icon appearance on Mac OS X, set `bitmaps` to an empty array:

```
air.NativeApplication.nativeApplication.icon.bitmaps = [];
```

Dock icons

Adobe AIR 1.0 and later

AIR supports dock icons when `NativeApplication.supportsDockIcon` is `true`. The `NativeApplication.nativeApplication.icon` property represents the application icon on the dock (not a window dock icon).

Note: AIR does not support changing window icons on the dock under Mac OS X. Also, changes to the application dock icon only apply while an application is running — the icon reverts to its normal appearance when the application terminates.

Dock icon menus

Adobe AIR 1.0 and later

You can add commands to the standard dock menu by creating a `NativeMenu` object containing the commands and assigning it to the `NativeApplication.nativeApplication.icon.menu` property. The items in the menu are displayed above the standard dock icon menu items.

Bouncing the dock

Adobe AIR 1.0 and later

You can bounce the dock icon by calling the `NativeApplication.nativeApplication.icon.bounce()` method. If you set the `bounce()` `priority` parameter to `informational`, then the icon bounces once. If you set it to `critical`, then the icon bounces until the user activates the application. Constants for the `priority` parameter are defined in the `NotificationType` class.

Note: The icon does not bounce if the application is already active.

Dock icon events

Adobe AIR 1.0 and later

When the dock icon is clicked, the `NativeApplication` object dispatches an `invoke` event. If the application is not running, the system launches it. Otherwise, the `invoke` event is delivered to the running application instance.

System Tray icons

Adobe AIR 1.0 and later

AIR supports system tray icons when `NativeApplication.supportsSystemTrayIcon` is `true`, which is currently the case only on Windows and most Linux distributions. On Windows and Linux, system tray icons are displayed in the notification area of the taskbar. No icon is displayed by default. To show an icon, assign an array containing `BitmapData` objects to the `icon.bitmaps` property. To change the icon image, assign an array containing the new images to `bitmaps`. To remove the icon, set `bitmaps` to `null`.

System tray icon menus

Adobe AIR 1.0 and later

You can add a menu to the system tray icon by creating a `NativeMenu` object and assigning it to the `NativeApplication.nativeApplication.icon.menu` property (no default menu is provided by the operating system). Access the system tray icon menu by right-clicking the icon.

System tray icon tooltips

Adobe AIR 1.0 and later

Add a tooltip to an icon by setting the tooltip property:

```
air.NativeApplication.nativeApplication.icon.tooltip = "Application name";
```

System tray icon events

Adobe AIR 1.0 and later

The `SystemTrayIcon` object referenced by the `NativeApplication.nativeApplication.icon` property dispatches a `ScreenMouseEvent` for `click`, `mouseDown`, `mouseUp`, `rightClick`, `rightMouseDown`, and `rightMouseUp` events. You can use these events, along with an icon menu, to allow users to interact with your application when it has no visible windows.

Example: Creating an application with no windows

Adobe AIR 1.0 and later

The following example creates an AIR application which has a system tray icon, but no visible windows. (The `visible` property of the application must not be set to `true` in the application descriptor, or the window will be visible when the application starts up.)

Note: When using the `Flex WindowedApplication` component, you must set the `visible` attribute of the `WindowedApplication` tag to `false`. This attribute supercedes the setting in the application descriptor.

```
<html>
<head>
<script src="AIRAliases.js" language="JavaScript" type="text/javascript"></script>
<script language="JavaScript" type="text/javascript">
    var iconLoadComplete = function(event)
    {
        air.NativeApplication.nativeApplication.icon.bitmaps =
[event.target.content.bitmapData];
    }

    air.NativeApplication.nativeApplication.autoExit = false;
    var iconLoad = new air.Loader();
    var iconMenu = new air.NativeMenu();
    var exitCommand = iconMenu.addItem(new air.NativeMenuItem("Exit"));
    exitCommand.addEventListener(air.Event.SELECT,function(event) {
        air.NativeApplication.nativeApplication.icon.bitmaps = [];
        air.NativeApplication.nativeApplication.exit();
    });

    if (air.NativeApplication.supportsSystemTrayIcon) {
        air.NativeApplication.nativeApplication.autoExit = false;
        iconLoad.contentLoaderInfo.addEventListener(air.Event.COMPLETE,iconLoadComplete);
        iconLoad.load(new air.URLRequest("icons/AIRApp_16.png"));
        air.NativeApplication.nativeApplication.icon.tooltip = "AIR application";
        air.NativeApplication.nativeApplication.icon.menu = iconMenu;
    }

    if (air.NativeApplication.supportsDockIcon) {
        iconLoad.contentLoaderInfo.addEventListener(air.Event.COMPLETE,iconLoadComplete);
        iconLoad.load(new air.URLRequest("icons/AIRApp_128.png"));
        air.NativeApplication.nativeApplication.icon.menu = iconMenu;
    }

</script>
</head>
<body>
</body>
</html>
```

Note: The example assumes that there are image files named `AIRApp_16.png` and `AIRApp_128.png` in an `icons` subdirectory of the application. (Sample icon files, which you can copy to your project folder, are included in the AIR SDK.)

Window taskbar icons and buttons

Adobe AIR 1.0 and later

Iconified representations of windows are typically displayed in the window area of a taskbar or dock to allow users to easily access background or minimized windows. The Mac OS X dock displays an icon for your application as well as an icon for each minimized window. The Microsoft Windows and Linux taskbars display a button containing the program icon and title for each normal-type window in your application.

Highlighting the taskbar window button

Adobe AIR 1.0 and later

When a window is in the background, you can notify the user that an event of interest related to the window has occurred. On Mac OS X, you can notify the user by bouncing the application dock icon (as described in “[Bouncing the dock](#)” on page 141). On Windows and Linux, you can highlight the window taskbar button by calling the `notifyUser()` method of the `NativeWindow` instance. The `type` parameter passed to the method determines the urgency of the notification:

- `NotificationType.CRITICAL`: the window icon flashes until the user brings the window to the foreground.
- `NotificationType.INFORMATIONAL`: the window icon highlights by changing color.

Note: On Linux, only the informational type of notification is supported. Passing either type value to the `notifyUser()` function will create the same effect.

The following statement highlights the taskbar button of a window:

```
window.nativeWindow.notifyUser(air.NotificationType.INFORMATIONAL);
```

Calling the `NativeWindow.notifyUser()` method on an operating system that does not support window-level notification has no effect. Use the `NativeWindow.supportsNotification` property to determine if window notification is supported.

Creating windows without taskbar buttons or icons

Adobe AIR 1.0 and later

On the Windows operating system, windows created with the types *utility* or *lightweight* do not appear on the taskbar. Invisible windows do not appear on the taskbar, either.

Because the initial window is necessarily of type, *normal*, in order to create an application without any windows appearing in the taskbar, you must either close the initial window or leave it invisible. To close all windows in your application without terminating the application, set the `autoExit` property of the `NativeApplication` object to `false` before closing the last window. To simply prevent the initial window from ever becoming visible, add `<visible>false</visible>` to the `<initialWindow>` element of the application descriptor file (and do not set the `visible` property to `true` or call the `activate()` method of the window).

In new windows opened by the application, set the `type` property of the `NativeWindowInitOption` object passed to the window constructor to `NativeWindowType.UTILITY` or `NativeWindowType.LIGHTWEIGHT`.

On Mac OS X, windows that are minimized are displayed on the dock taskbar. You can prevent the minimized icon from being displayed by hiding the window instead of minimizing it. The following example listens for a `nativeWindowDisplayState` change event and cancels it if the window is being minimized. Instead the handler sets the window `visible` property to `false`:

```
function preventMinimize(event) {
    if(event.afterDisplayState == air.NativeWindowDisplayState.MINIMIZED) {
        event.preventDefault();
        event.target.visible = false;
    }
}
```

If a window is minimized on the Mac OS X dock when you set the `visible` property to `false`, the dock icon is not removed. A user can still click the icon to make the window reappear.

Chapter 11: Working with the file system

Flash Player 9 and later, Adobe AIR 1.0 and later

The Adobe® AIR® file system API provides complete access to the file system of the host computer. Using these classes, you can access and manage directories and files, create directories and files, write data to files, and so on.

More Help topics

[flash.filesystem.File](#)

[flash.filesystem.FileStream](#)

Using the AIR file system API

Adobe AIR 1.0 and later

The Adobe AIR file system API includes the following classes:

- File
- FileMode
- FileStream

The file system API lets you do the following (and more):

- Copy, create, delete, and move files and directories
- Get information about files and directories
- Read and write files

AIR file basics

Adobe AIR 1.0 and later

For a quick explanation and code examples of working with the file system in AIR, see the following quick start articles on the Adobe Developer Connection:

- [Building a text-file editor](#)
- [Building a directory search application](#)
- [Reading and writing from an XML preferences file](#)

Adobe AIR provides classes that you can use to access, create, and manage both files and folders. These classes, contained in the `flash.filesystem` package, are used as follows:

Adobe AIR provides classes that you can use to access, create, and manage both files and folders. These classes, contained in the `runtime.flash.filesystem` package, are used as follows:

File classes	Description
File	File object represents a path to a file or directory. You use a file object to create a pointer to a file or folder, initiating interaction with the file or folder.
FileMode	The <code>FileMode</code> class defines string constants used in the <code>fileMode</code> parameter of the <code>open()</code> and <code>openAsync()</code> methods of the <code>FileStream</code> class. The <code>fileMode</code> parameter of these methods determines the capabilities available to the <code>FileStream</code> object once the file is opened, which include writing, reading, appending, and updating.
FileStream	<code>FileStream</code> object is used to open files for reading and writing. Once you've created a <code>File</code> object that points to a new or existing file, you pass that pointer to the <code>FileStream</code> object so that you can open it and read or write data.

Some methods in the `File` class have both synchronous and asynchronous versions:

- `File.copyTo()` and `File.copyToAsync()`
- `File.deleteDirectory()` and `File.deleteDirectoryAsync()`
- `File.deleteFile()` and `File.deleteFileAsync()`
- `File.getDirectoryListing()` and `File.getDirectoryListingAsync()`
- `File.moveTo()` and `File.moveToAsync()`
- `File.moveToTrash()` and `File.moveToTrashAsync()`

Also, `FileStream` operations work synchronously or asynchronously depending on how the `FileStream` object opens the file: by calling the `open()` method or by calling the `openAsync()` method.

The asynchronous versions let you initiate processes that run in the background and dispatch events when complete (or when error events occur). Other code can execute while these asynchronous background processes are taking place. With asynchronous versions of the operations, you must set up event listener functions, using the `addEventListener()` method of the `File` or `FileStream` object that calls the function.

The synchronous versions let you write simpler code that does not rely on setting up event listeners. However, since other code cannot execute while a synchronous method is executing, important processes such as display object rendering and animation can be delayed.

Working with File objects in AIR

Adobe AIR 1.0 and later

A `File` object is a pointer to a file or directory in the file system.

The `File` class extends the `FileReference` class. The `FileReference` class, which is available in Adobe® Flash® Player as well as AIR, represents a pointer to a file. The `File` class adds properties and methods that are not exposed in Flash Player (in a SWF file running in a browser), due to security considerations.

About the File class

Adobe AIR 1.0 and later

You can use the `File` class for the following:

- Getting the path to special directories, including the user directory, the user's documents directory, the directory from which the application was launched, and the application directory
- Copying files and directories

- Moving files and directories
- Deleting files and directories (or moving them to the trash)
- Listing files and directories contained in a directory
- Creating temporary files and folders

Once a File object points to a file path, you can use it to read and write file data, using the `FileStream` class.

A File object can point to the path of a file or directory that does not yet exist. You can use such a File object in creating a file or directory.

Paths of File objects

Adobe AIR 1.0 and later

Each File object has two properties that each define its path:

Property	Description
<code>nativePath</code>	Specifies the platform-specific path to a file. For example, on Windows a path might be "c:\Sample directory\test.txt" whereas on Mac OS it could be "/Sample directory/test.txt". A <code>nativePath</code> property uses the backslash (\) character as the directory separator character on Windows, and it uses the forward slash (/) character on Mac OS and Linux.
<code>url</code>	This may use the file URL scheme to point to a file. For example, on Windows a path might be "file:///c:/Sample%20directory/test.txt" whereas on Mac OS it could be "file:///Sample%20directory/test.txt". The runtime includes other special URL schemes besides <code>file</code> and are described in " Supported AIR URL schemes " on page 153

The File class includes static properties for pointing to standard directories on Mac OS, Windows, and Linux. These properties include:

- `File.applicationStorageDirectory`—a storage directory unique to each installed AIR application. This directory is an appropriate place to store dynamic application assets and user preferences. Consider storing large amounts of data elsewhere.
- `File.applicationDirectory`—the directory where the application is installed (along with any installed assets). On some operating systems, the application is stored in a single package file rather than a physical directory. In this case, the contents may not be accessible using the native path. The application directory is read-only.
- `File.desktopDirectory`—the user's desktop directory. If a platform does not define a desktop directory, another location on the file system is used.
- `File.documentsDirectory`—the user's documents directory. If a platform does not define a documents directory, another location on the file system is used.
- `File.userDirectory`—the user directory. If a platform does not define a user directory, another location on the file system is used.

Note: When a platform does not define standard locations for desktop, documents, or user directories, `File.documentsDirectory`, `File.desktopDirectory`, and `File.userDirectory` can reference the same directory.

These properties have different values on different operating systems. For example, Mac and Windows each have a different native path to the user's desktop directory. However, the `File.desktopDirectory` property points to an appropriate directory path on every platform. To write applications that work well across platforms, use these properties as the basis for referencing other directories and files used by the application. Then use the `resolvePath()` method to refine the path. For example, this code points to the `preferences.xml` file in the application storage directory:

```
var prefsFile:File = air.File.applicationStorageDirectory;
prefsFile = prefsFile.resolvePath("preferences.xml");
```

Although the File class lets you point to a specific file path, doing so can lead to applications that do not work across platforms. For example, the path C:\Documents and Settings\joe\ only works on Windows. For these reasons, it is best to use the static properties of the File class, such as File.documentsDirectory.

Common directory locations

Platform	Directory type	Typical file system location
Linux	Application	/opt/filename/share
	Application-storage	/home/userName/.appdata/applicationID/Local Store
	Desktop	/home/userName/Desktop
	Documents	/home/userName/Documents
	Temporary	/tmp/FlashTmp.randomString
	User	/home/userName
Mac	Application	/Applications/filename.app/Contents/Resources
	Application-storage	/Users/userName/Library/Preferences/applicationID/Local Store
	Desktop	/Users/userName/Desktop
	Documents	/Users/userName/Documents
	Temporary	/private/var/folders/JY/randomString/TemporaryItems/FlashTmp
	User	/Users/userName
Windows	Application	C:\Program Files\filename
	Application-storage	C:\Documents and settings\userName\ApplicationData\applicationID\Local Store
	Desktop	C:\Documents and settings\userName\Desktop
	Documents	C:\Documents and settings\userName\My Documents
	Temporary	C:\Documents and settings\userName\Local Settings\Temp\randomString.tmp
	User	C:\Documents and settings\userName

The actual native paths for these directories vary based on the operating system and computer configuration. The paths shown in this table are typical examples. You should always use the appropriate static File class properties to refer to these directories so that your application works correctly on any platform. In an actual AIR application, the values for applicationID and filename shown in the table are taken from the application descriptor. If you specify a publisher ID in the application descriptor, then the publisher ID is appended to the application ID in these paths. The value for userName is the account name of the installing user.

Pointing a File object to a directory

Adobe AIR 1.0 and later

There are different ways to set a File object to point to a directory.

Pointing to the user's home directory

Adobe AIR 1.0 and later

You can point a File object to the user's home directory. The following code sets a File object to point to an AIR Test subdirectory of the home directory:

```
var file = air.File.userDirectory.resolvePath("AIR Test");
```

Pointing to the user's documents directory

Adobe AIR 1.0 and later

You can point a File object to the user's documents directory. The following code sets a File object to point to an AIR Test subdirectory of the documents directory:

```
var file = air.File.documentsDirectory.resolvePath("AIR Test");
```

Pointing to the desktop directory

Adobe AIR 1.0 and later

You can point a File object to the desktop. The following code sets a File object to point to an AIR Test subdirectory of the desktop:

```
var file = air.File.desktopDirectory.resolvePath("AIR Test");
```

Pointing to the application storage directory

Adobe AIR 1.0 and later

You can point a File object to the application storage directory. For every AIR application, there is a unique associated path that defines the application storage directory. This directory is unique to each application and user. You can use this directory to store user-specific, application-specific data (such as user data or preferences files). For example, the following code points a File object to a preferences file, `prefs.xml`, contained in the application storage directory:

```
var file = air.File.applicationStorageDirectory;  
file = file.resolvePath("prefs.xml");
```

The application storage directory location is typically based on the user name and the application ID. The following file system locations are given here to help you debug your application. You should always use the `File.applicationStorage` property or `app-storage:` URI scheme to resolve files in this directory:

- On Mac OS—In:

```
/Users/user name/Library/Preferences/applicationID/Local Store/
```

For example:

```
/Users/babbage/Library/Preferences/com.example.TestApp/Local Store
```

- On Windows—In the documents and Settings directory, in:

```
C:\Documents and Settings\user name\Application Data\applicationID\Local Store\
```

For example:

```
C:\Documents and Settings\babbage\Application Data\com.example.TestApp\Local Store
```

- On Linux—In:

```
/home/user name/.appdata/applicationID/Local Store/
```

For example:

```
/home/babbage/.appdata/com.example.TestApp/Local Store
```

Note: If an application has a publisher ID, then the publisher ID is also used as part of the path to the application storage directory.

The URL (and `url` property) for a File object created with `File.applicationStorageDirectory` uses the `app-storage` URL scheme (see “[Supported AIR URL schemes](#)” on page 153), as in the following:

```
var dir = air.File.applicationStorageDirectory;
dir = dir.resolvePath("prefs.xml");
air.trace(dir.url); // app-storage:/preferences
```

Pointing to the application directory

Adobe AIR 1.0 and later

You can point a File object to the directory in which the application was installed, known as the application directory. You can reference this directory using the `File.applicationDirectory` property. You can use this directory to examine the application descriptor file or other resources installed with the application. For example, the following code points a File object to a directory named *images* in the application directory:

```
var dir = air.File.applicationDirectory;
dir = dir.resolvePath("images");
```

The URL (and `url` property) for a File object created with `File.applicationDirectory` uses the `app` URL scheme (see “[Supported AIR URL schemes](#)” on page 153), as in the following:

```
var dir = air.File.applicationDirectory;
dir = dir.resolvePath("images");
air.trace(dir.url); // app:/images
```

Pointing to the file system root

Adobe AIR 1.0 and later

The `File.getRootDirectories()` method lists all root volumes, such as C: and mounted volumes, on a Windows computer. On Mac OS and Linux, this method always returns the unique root directory for the machine (the `/` directory). The `StorageVolumeInfo.getStorageVolumes()` method provides more detailed information on mounted storage volumes (see “[Working with storage volumes](#)” on page 163).

Pointing to an explicit directory

Adobe AIR 1.0 and later

You can point the File object to an explicit directory by setting the `nativePath` property of the File object, as in the following example (on Windows):

```
var file = new air.File();
file.nativePath = "C:\\AIR Test";
```

Important: Pointing to an explicit path this way can lead to code that does not work across platforms. For example, the previous example only works on Windows. You can use the static properties of the File object, such as `File.applicationStorageDirectory`, to locate a directory that works cross-platform. Then use the `resolvePath()` method (see the next section) to navigate to a relative path.

Navigating to relative paths

Adobe AIR 1.0 and later

You can use the `resolvePath()` method to obtain a path relative to another given path. For example, the following code sets a `File` object to point to an "AIR Test" subdirectory of the user's home directory:

```
var file = air.File.userDirectory;
file = file.resolvePath("AIR Test");
```

You can also use the `url` property of a `File` object to point it to a directory based on a URL string, as in the following:

```
var urlStr = "file:///C:/AIR Test/";
var file = new air.File()
file.url = urlStr;
```

For more information, see [“Modifying File paths”](#) on page 153.

Letting the user browse to select a directory

Adobe AIR 1.0 and later

The `File` class includes the `browseForDirectory()` method, which presents a system dialog box in which the user can select a directory to assign to the object. The `browseForDirectory()` method is asynchronous. The `File` object dispatches a `select` event if the user selects a directory and clicks the Open button, or it dispatches a `cancel` event if the user clicks the Cancel button.

For example, the following code lets the user select a directory and outputs the directory path upon selection:

```
var file = new air.File();
file.addEventListener(air.Event.SELECT, dirSelected);
file.browseForDirectory("Select a directory");
function dirSelected(event) {
    alert(file.nativePath);
}
```

Pointing to the directory from which the application was invoked

Adobe AIR 1.0 and later

You can get the directory location from which an application is invoked, by checking the `currentDirectory` property of the `InvokeEvent` object dispatched when the application is invoked. For details, see [“Capturing command line arguments”](#) on page 297.

Pointing a File object to a file

Adobe AIR 1.0 and later

There are different ways to set the file to which a `File` object points.

Pointing to an explicit file path

Adobe AIR 1.0 and later

Important: Pointing to an explicit path can lead to code that does not work across platforms. For example, the path `C:/foo.txt` only works on Windows. You can use the static properties of the `File` object, such as `File.applicationStorageDirectory`, to locate a directory that works cross-platform. Then use the `resolvePath()` method (see [“Modifying File paths”](#) on page 153) to navigate to a relative path.

You can use the `url` property of a `File` object to point it to a file or directory based on a URL string, as in the following:

```
var urlStr = "file:///C:/AIR Test/test.txt";
var file = new air.File()
file.url = urlStr;
```

You can also pass the URL to the `File()` constructor function, as in the following:

```
var urlStr = "file:///C:/AIR Test/test.txt";
var file = new air.File(urlStr);
```

The `url` property always returns the URI-encoded version of the URL (for example, blank spaces are replaced with "%20"):

```
file.url = "file:///c:/AIR Test";
alert(file.url); // file:///c:/AIR%20Test
```

You can also use the `nativePath` property of a `File` object to set an explicit path. For example, the following code, when run on a Windows computer, sets a `File` object to the `test.txt` file in the `AIR Test` subdirectory of the `C:` drive:

```
var file = new air.File();
file.nativePath = "C:/AIR Test/test.txt";
```

You can also pass this path to the `File()` constructor function, as in the following:

```
var file = new air.File("C:/AIR Test/test.txt");
```

Use the forward slash (/) character as the path delimiter for the `nativePath` property. On Windows, you can also use the backslash (\) character, but doing so leads to applications that do not work across platforms.

For more information, see “[Modifying File paths](#)” on page 153.

Enumerating files in a directory

Adobe AIR 1.0 and later

You can use the `getDirectoryListing()` method of a `File` object to get an array of `File` objects pointing to files and subdirectories at the root level of a directory. For more information, see “[Enumerating directories](#)” on page 159.

Letting the user browse to select a file

Adobe AIR 1.0 and later

The `File` class includes the following methods that present a system dialog box in which the user can select a file to assign to the object:

- `browseForOpen()`
- `browseForSave()`
- `browseForOpenMultiple()`

These methods are each asynchronous. The `browseForOpen()` and `browseForSave()` methods dispatch the `select` event when the user selects a file (or a target path, in the case of `browseForSave()`). With the `browseForOpen()` and `browseForSave()` methods, upon selection the target `File` object points to the selected files. The `browseForOpenMultiple()` method dispatches a `selectMultiple` event when the user selects files. The `selectMultiple` event is of type `FileListEvent`, which has a `files` property that is an array of `File` objects (pointing to the selected files).

For example, the following code presents the user with an “Open” dialog box in which the user can select a file:

```
var fileToOpen = air.File.documentsDirectory;
selectTextFile(fileToOpen);

function selectTextFile(root)
{
    var txtFilter = new air.FileFilter("Text", "*.as;*.css;*.html;*.txt;*.xml");
    root.browseForOpen("Open", new window.runtime.Array(txtFilter));
    root.addEventListener(air.Event.SELECT, fileSelected);
}

function fileSelected(event)
{
    trace(fileToOpen.nativePath);
}
```

If the application has another browser dialog box open when you call a browse method, the runtime throws an Error exception.

Modifying File paths

Adobe AIR 1.0 and later

You can also modify the path of an existing File object by calling the `resolvePath()` method or by modifying the `nativePath` or `url` property of the object, as in the following examples (on Windows):

```
file1 = air.File.documentsDirectory;
file1 = file1.resolvePath("AIR Test");
alert(file1.nativePath); // C:\Documents and Settings\userName\My Documents\AIR Test
var file2 = air.File.documentsDirectory;
file2 = file2.resolvePath("..");
alert(file2.nativePath); // C:\Documents and Settings\userName
var file3 = air.File.documentsDirectory;
file3.nativePath += "/subdirectory";
alert(file3.nativePath); // C:\Documents and Settings\userName\My Documents\subdirectory
var file4 = new air.File();
file4.url = "file:///c:/AIR Test/test.txt";
alert(file4.nativePath); // C:\AIR Test\test.txt
```

When using the `nativePath` property, use the forward slash (/) character as the directory separator character. On Windows, you can use the backslash (\) character as well, but you should not do so, as it leads to code that does not work cross-platform.

Supported AIR URL schemes

Adobe AIR 1.0 and later

In AIR, you can use any of the following URL schemes in defining the `url` property of a File object:

URL scheme	Description
file	Use to specify a path relative to the root of the file system. For example: <code>file:///c:/AIR Test/test.txt</code> The URL standard specifies that a file URL takes the form <code>file://<host>/<path></code> . As a special case, <code><host></code> can be the empty string, which is interpreted as "the machine from which the URL is being interpreted." For this reason, file URLs often have three slashes (<code>///</code>).
app	Use to specify a path relative to the root directory of the installed application (the directory that contains the <code>application.xml</code> file for the installed application). For example, the following path points to an images subdirectory of the directory of the installed application: <code>app:/images</code>
app-storage	Use to specify a path relative to the application store directory. For each installed application, AIR defines a unique application store directory, which is a useful place to store data specific to that application. For example, the following path points to a <code>prefs.xml</code> file in a settings subdirectory of the application store directory: <code>app-storage:/settings/prefs.xml</code>

Finding the relative path between two files

Adobe AIR 1.0 and later

You can use the `getRelativePath()` method to find the relative path between two files:

```
var file1 = air.File.documentsDirectory;
file1 = file1.resolvePath("AIR Test");
var file2 = air.File.documentsDirectory;
file2 = file2.resolvePath("AIR Test/bob/test.txt");

alert(file1.getRelativePath(file2)); // bob/test.txt
```

The second parameter of the `getRelativePath()` method, the `useDotDot` parameter, allows for `..` syntax to be returned in results, to indicate parent directories:

```
var file1 = air.File.documentsDirectory;
file1 = file1.resolvePath("AIR Test");
var file2 = air.File.documentsDirectory;
file2 = file2.resolvePath("AIR Test/bob/test.txt");
var file3 = air.File.documentsDirectory;
file3 = file3.resolvePath("AIR Test/susan/test.txt");

alert(file2.getRelativePath(file1, true)); // ../../
alert(file3.getRelativePath(file2, true)); // ../../bob/test.txt
```

Obtaining canonical versions of file names

Adobe AIR 1.0 and later

File and path names are not case sensitive on Windows and Mac OS. In the following, two File objects point to the same file:

```
File.documentsDirectory.resolvePath("test.txt");
File.documentsDirectory.resolvePath("TeSt.TxT");
```

However, documents and directory names do include capitalization. For example, the following assumes that there is a folder named AIR Test in the documents directory, as in the following examples:


```
var file = air.File.documentsDirectory;
file = file.resolvePath("AIR test");
trace(file.nativePath); // ... AIR test
file.canonicalize();
alert(file.nativePath); // ... AIR Test
```

The `canonicalize()` method converts the `nativePath` object to use the correct capitalization for the file or directory name. On case sensitive file systems (such as Linux), when multiple files exist with names differing only in case, the `canonicalize()` method adjusts the path to match the first file found (in an order determined by the file system).

You can also use the `canonicalize()` method to convert short file names ("8.3" names) to long file names on Windows, as in the following examples:

```
var path = new air.File();
path.nativePath = "C:\\AIR~1";
path.canonicalize();
alert(path.nativePath); // C:\AIR Test
```

Working with packages and symbolic links

Adobe AIR 1.0 and later

Various operating systems support package files and symbolic link files:

Packages—On Mac OS, directories can be designated as packages and show up in the Mac OS Finder as a single file rather than as a directory.

Symbolic links—Mac OS, Linux, and Windows Vista support symbolic links. Symbolic links allow a file to point to another file or directory on disk. Although similar, symbolic links are not the same as aliases. An alias is always reported as a file (rather than a directory), and reading or writing to an alias or shortcut never affects the original file or directory that it points to. On the other hand, a symbolic link behaves exactly like the file or directory it points to. It can be reported as a file or a directory, and reading or writing to a symbolic link affects the file or directory that it points to, not the symbolic link itself. Additionally, on Windows the `isSymbolicLink` property for a File object referencing a junction point (used in the NTFS file system) is set to `true`.

The File class includes the `isPackage` and `isSymbolicLink` properties for checking if a File object references a package or symbolic link.

The following code iterates through the user's desktop directory, listing subdirectories that are *not* packages:

```
var desktopNodes = air.File.desktopDirectory.getDirectoryListing();
for (i = 0; i < desktopNodes.length; i++)
{
    if (desktopNodes[i].isDirectory && !desktopNodes[i].isPackage)
    {
        air.trace(desktopNodes[i].name);
    }
}
```

The following code iterates through the user's desktop directory, listing files and directories that are *not* symbolic links:

```
var desktopNodes = air.File.desktopDirectory.getDirectoryListing();
for (i = 0; i < desktopNodes.length; i++)
{
    if (!desktopNodes[i].isSymbolicLink)
    {
        air.trace(desktopNodes[i].name);
    }
}
```

The `canonicalize()` method changes the path of a symbolic link to point to the file or directory to which the link refers. The following code iterates through the user's desktop directory, and reports the paths referenced by files that are symbolic links:

```
var desktopNodes = air.File.desktopDirectory.getDirectoryListing();
for (i = 0; i < desktopNodes.length; i++)
{
    if (desktopNodes[i].isSymbolicLink)
    {
        var linkNode = desktopNodes[i];
        linkNode.canonicalize();
        air.trace(desktopNodes[i].name);
    }
}
```

Determining space available on a volume

Adobe AIR 1.0 and later

The `spaceAvailable` property of a File object is the space available for use at the File location, in bytes. For example, the following code checks the space available in the application storage directory:

```
air.trace(air.File.applicationStorageDirectory.spaceAvailable);
```

If the File object references a directory, the `spaceAvailable` property indicates the space in the directory that files can use. If the File object references a file, the `spaceAvailable` property indicates the space into which the file could grow. If the file location does not exist, the `spaceAvailable` property is set to 0. If the File object references a symbolic link, the `spaceAvailable` property is set to space available at the location the symbolic link points to.

Typically the space available for a directory or file is the same as the space available on the volume containing the directory or file. However, space available can take into account quotas and per-directory limits.

Adding a file or directory to a volume generally requires more space than the actual size of the file or the size of the contents of the directory. For example, the operating system may require more space to store index information. Or the disk sectors required may use additional space. Also, available space changes dynamically. So, you cannot expect to allocate all of the reported space for file storage. For information on writing to the file system, see [“Reading and writing files”](#) on page 164.

The `StorageVolumeInfo.getStorageVolumes()` method provides more detailed information on mounted storage volumes (see [“Working with storage volumes”](#) on page 163).

Opening files with the default system application

Adobe AIR 2 and later

In AIR 2, you can open a file using the application registered by the operating system to open it. For example, an AIR application can open a DOC file with the application registered to open it. Use the `openWithDefaultApplication()` method of a `File` object to open the file. For example, the following code opens a file named `test.doc` on the user's desktop and opens it with the default application for DOC files:

```
var file = air.File.desktopDirectory;
file = file.resolvePath("test.doc");
file.openWithDefaultApplication();
```

Note: On Linux, the file's MIME type, not the filename extension, determines the default application for a file.

The following code lets the user navigate to an mp3 file and open it in the default application for playing mp3 files:

```
var file = air.File.documentsDirectory;
var mp3Filter = new air.FileFilter("MP3 Files", "*.mp3");
file.browseForOpen("Open", [mp3Filter]);
file.addEventListener(Event.SELECT, fileSelected);
function fileSelected(event)
{
    file.openWithDefaultApplication();
}
```

You cannot use the `openWithDefaultApplication()` method with files located in the application directory.

AIR prevents you from using the `openWithDefaultApplication()` method to open certain files. On Windows, AIR prevents you from opening files that have certain filetypes, such as EXE or BAT. On Mac OS and Linux, AIR prevents you from opening files that will launch in certain application. (These include Terminal and AppletLauncher on Mac OS; and `csh`, `bash`, or `ruby` on Linux.) Attempting to open one of these files using the `openWithDefaultApplication()` method results in an exception. For a complete list of prevented filetypes, see the language reference entry for the `File.openWithDefaultApplication()` method.

Note: This limitation does not exist for an AIR application installed using a native installer (an extended desktop application).

Getting file system information

Adobe AIR 1.0 and later

The `File` class includes the following static properties that provide some useful information about the file system:

Property	Description
<code>File.lineEnding</code>	The line-ending character sequence used by the host operating system. On Mac OS and Linux, this is the line-feed character. On Windows, this is the carriage return character followed by the line-feed character.
<code>File.separator</code>	The host operating system's path component separator character. On Mac OS and Linux, this is the forward slash (/) character. On Windows, it is the backslash (\) character.
<code>File.systemCharset</code>	The default encoding used for files by the host operating system. This pertains to the character set used by the operating system, corresponding to its language.

The `Capabilities` class also includes useful system information that can be useful when working with files:

Property	Description
Capabilities.hasIME	Specifies whether the player is running on a system that does (<code>true</code>) or does not (<code>false</code>) have an input method editor (IME) installed.
Capabilities.language	Specifies the language code of the system on which the player is running.
Capabilities.os	Specifies the current operating system.

Note: Be careful when using `Capabilities.os` to determine system characteristics. If a more specific property exists to determine a system characteristic, use it. Otherwise, you run the risk of writing code that does not work correctly on all platforms. For example, consider the following code:

```
var separator:String;
if (Capabilities.os.indexOf("Mac") > -1)
{
    separator = "/";
}
else
{
    separator = "\\";
}
```

This code leads to problems on Linux. It is better to simply use the `File.separator` property.

Working with directories

Adobe AIR 1.0 and later

The runtime provides you with capabilities to work with directories on the local file system.

For details on creating `File` objects that point to directories, see “[Pointing a File object to a directory](#)” on page 148.

Creating directories

Adobe AIR 1.0 and later

The `File.createDirectory()` method lets you create a directory. For example, the following code creates a directory named AIR Test as a subdirectory of the user's home directory:

```
var dir = air.File.userDirectory.resolvePath("AIR Test");
dir.createDirectory();
```

If the directory exists, the `createDirectory()` method does nothing.

Also, in some modes, a `FileStream` object creates directories when opening files. Missing directories are created when you instantiate a `FileStream` instance with the `fileMode` parameter of the `FileStream()` constructor set to `FileMode.APPEND` or `FileMode.WRITE`. For more information, see “[Workflow for reading and writing files](#)” on page 164.

Creating a temporary directory

Adobe AIR 1.0 and later

The `File` class includes a `createTempDirectory()` method, which creates a directory in the temporary directory folder for the System, as in the following example:

```
var temp = air.File.createTempDirectory();
```

The `createTempDirectory()` method automatically creates a unique temporary directory (saving you the work of determining a new unique location).

You can use a temporary directory to temporarily store temporary files used for a session of the application. Note that there is a `createTempFile()` method for creating new, unique temporary files in the System temporary directory.

You may want to delete the temporary directory before closing the application, as it is *not* automatically deleted on all devices.

Enumerating directories

Adobe AIR 1.0 and later

You can use the `getDirectoryListing()` method or the `getDirectoryListingAsync()` method of a `File` object to get an array of `File` objects pointing to files and subfolders in a directory.

For example, the following code lists the contents of the user's documents directory (without examining subdirectories):

```
var directory = air.File.documentsDirectory;
var contents = directory.getDirectoryListing();
for (i = 0; i < contents.length; i++)
{
    alert(contents[i].name, contents[i].size);
}
```

When using the asynchronous version of the method, the `directoryListing` event object has a `files` property that is the array of `File` objects pertaining to the directories:

```
var directory = air.File.documentsDirectory;
directory.getDirectoryListingAsync();
directory.addEventListener(air.FileListEvent.DIRECTORY_LISTING, dirListHandler);

function dirListHandler(event)
{
    var contents = event.files;
    for (i = 0; i < contents.length; i++)
    {
        alert(contents[i].name, contents[i].size);
    }
}
```

Copying and moving directories

Adobe AIR 1.0 and later

You can copy or move a directory, using the same methods as you would to copy or move a file. For example, the following code copies a directory synchronously:

```
var sourceDir = air.File.documentsDirectory.resolvePath("AIR Test");
var resultDir = air.File.documentsDirectory.resolvePath("AIR Test Copy");
sourceDir.copyTo(resultDir);
```

When you specify `true` for the `overwrite` parameter of the `copyTo()` method, all files and folders in an existing target directory are deleted and replaced with the files and folders in the source directory (even if the target file does not exist in the source directory).

The directory that you specify as the `newLocation` parameter of the `copyTo()` method specifies the path to the resulting directory; it does *not* specify the *parent* directory that will contain the resulting directory.

For details, see “[Copying and moving files](#)” on page 161.

Deleting directory contents

Adobe AIR 1.0 and later

The `File` class includes a `deleteDirectory()` method and a `deleteDirectoryAsync()` method. These methods delete directories, the first working synchronously, the second working asynchronously (see “[AIR file basics](#)” on page 145). Both methods include a `deleteDirectoryContents` parameter (which takes a Boolean value); when this parameter is set to `true` (the default value is `false`) the call to the method deletes non-empty directories; otherwise, only empty directories are deleted.

For example, the following code synchronously deletes the AIR Test subdirectory of the user's documents directory:

```
var directory = air.File.documentsDirectory.resolvePath("AIR Test");
directory.deleteDirectory(true);
```

The following code asynchronously deletes the AIR Test subdirectory of the user's documents directory:

```
var directory = air.File.documentsDirectory.resolvePath("AIR Test");
directory.addEventListener(air.Event.COMPLETE, completeHandler)
directory.deleteDirectoryAsync(true);
```

```
function completeHandler(event) {
    alert("Deleted.")
}
```

Also included are the `moveToTrash()` and `moveToTrashAsync()` methods, which you can use to move a directory to the System trash. For details, see “[Moving a file to the trash](#)” on page 162.

Working with files

Adobe AIR 1.0 and later

Using the AIR file API, you can add basic file interaction capabilities to your applications. For example, you can read and write files, copy and delete files, and so on. Since your applications can access the local file system, refer to “[AIR security](#)” on page 69, if you haven't already done so.

Note: You can associate a file type with an AIR application (so that double-clicking it opens the application). For details, see “[Managing file associations](#)” on page 304.

Getting file information

Adobe AIR 1.0 and later

The `File` class includes the following properties that provide information about a file or directory to which a `File` object points:

File property	Description
creationDate	The creation date of the file on the local disk.
creator	Obsolete—use the <code>extension</code> property. (This property reports the Macintosh creator type of the file, which is only used in Mac OS versions prior to Mac OS X.)
downloaded	(AIR 2 and later) Indicates whether the referenced file or directory was downloaded (from the internet) or not. property is only meaningful on operating systems in which files can be flagged as downloaded: <ul style="list-style-type: none"> Windows XP service pack 2 and later, and on Windows Vista Mac OS 10.5 and later
exists	Whether the referenced file or directory exists.
extension	The file extension, which is the part of the name following (and not including) the final dot ("."). If there is no dot in the filename, the extension is <code>null</code> .
icon	An <code>Icon</code> object containing the icons defined for the file.
isDirectory	Whether the File object reference is to a directory.
modificationDate	The date that the file or directory on the local disk was last modified.
name	The name of the file or directory (including the file extension, if there is one) on the local disk.
nativePath	The full path in the host operating system representation. See “ Paths of File objects ” on page 147.
parent	The folder that contains the folder or file represented by the File object. This property is <code>null</code> if the File object references a file or directory in the root of the file system.
size	The size of the file on the local disk in bytes.
type	Obsolete—use the <code>extension</code> property. (On the Macintosh, this property is the four-character file type, which is only used in Mac OS versions prior to Mac OS X.)
url	The URL for the file or directory. See “ Paths of File objects ” on page 147.

For details on these properties, see the File class entry in the [Adobe AIR API Reference for HTML Developers](#).

Copying and moving files

Adobe AIR 1.0 and later

The File class includes two methods for copying files or directories: `copyTo()` and `copyToAsync()`. The File class includes two methods for moving files or directories: `moveTo()` and `moveToAsync()`. The `copyTo()` and `moveTo()` methods work synchronously, and the `copyToAsync()` and `moveToAsync()` methods work asynchronously (see “[AIR file basics](#)” on page 145).

To copy or move a file, you set up two File objects. One points to the file to copy or move, and it is the object that calls the copy or move method; the other points to the destination (result) path.

The following copies a test.txt file from the AIR Test subdirectory of the user's documents directory to a file named copy.txt in the same directory:

```
var original = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var newFile = air.File.documentsDirectory.resolvePath("AIR Test/copy.txt");
original.copyTo(newFile, true);
```

In this example, the value of `overwrite` parameter of the `copyTo()` method (the second parameter) is set to `true`. By setting `overwrite` to `true`, an existing target file is overwritten. This parameter is optional. If you set it to `false` (the default value), the operation dispatches an `IOErrorEvent` event if the target file exists (and the file is not copied).

The “Async” versions of the copy and move methods work asynchronously. Use the `addEventListener()` method to monitor completion of the task or error conditions, as in the following code:

```
var original = air.File.documentsDirectory;
original = original.resolvePath("AIR Test/test.txt");

var destination = air.File.documentsDirectory;
destination = destination.resolvePath("AIR Test 2/copy.txt");

original.addEventListener(air.Event.COMPLETE, fileMoveCompleteHandler);
original.addEventListener(air.IOErrorEvent.IO_ERROR, fileMoveIOErrorEventHandler);
original.moveToAsync(destination);

function fileMoveCompleteHandler(event) {
    alert(event.target); // [object File]
}
function fileMoveIOErrorEventHandler(event) {
    alert("I/O Error.");
}
```

The File class also includes the `File.moveToTrash()` and `File.moveToTrashAsync()` methods, which move a file or directory to the system trash.

Deleting a file

Adobe AIR 1.0 and later

The File class includes a `deleteFile()` method and a `deleteFileAsync()` method. These methods delete files, the first working synchronously, the second working asynchronously (see “[AIR file basics](#)” on page 145).

For example, the following code synchronously deletes the test.txt file in the user's documents directory:

```
var file = air.File.documentsDirectory.resolvePath("test.txt");
file.deleteFile();
```

The following code asynchronously deletes the test.txt file of the user's documents directory:

```
var file = air.File.documentsDirectory.resolvePath("test.txt");
file.addEventListener(air.Event.COMPLETE, completeHandler)
file.deleteFileAsync();

function completeHandler(event) {
    alert("Deleted.")
}
```

Also included are the `moveToTrash()` and `moveToTrashAsync()` methods, which you can use to move a file or directory to the System trash. For details, see “[Moving a file to the trash](#)” on page 162.

Moving a file to the trash

Adobe AIR 1.0 and later

The File class includes a `moveToTrash()` method and a `moveToTrashAsync()` method. These methods send a file or directory to the System trash, the first working synchronously, the second working asynchronously (see “[AIR file basics](#)” on page 145).

For example, the following code synchronously moves the test.txt file in the user's documents directory to the System trash:


```
var file = air.File.documentsDirectory.resolvePath("test.txt");  
file.moveToTrash();
```

Note: On operating systems that do not support the concept of a recoverable trash folder, the files are removed immediately.

Creating a temporary file

Adobe AIR 1.0 and later

The File class includes a `createTempFile()` method, which creates a file in the temporary directory folder for the System, as in the following example:

```
var temp = air.File.createTempFile();
```

The `createTempFile()` method automatically creates a unique temporary file (saving you the work of determining a new unique location).

You can use a temporary file to temporarily store information used in a session of the application. Note that there is also a `createTempDirectory()` method, for creating a unique temporary directory in the System temporary directory.

You may want to delete the temporary file before closing the application, as it is *not* automatically deleted on all devices.

Working with storage volumes

Adobe AIR 2 and later

In AIR 2, you can detect when mass storage volumes are mounted or unmounted. The `StorageVolumeInfo` class defines a singleton `storageVolumeInfo` object. The `StorageVolumeInfo.storageVolumeInfo` object dispatches a `storageVolumeMount` event when a storage volume is mounted. And it dispatches a `storageVolumeUnmount` event when a volume is unmounted. The `StorageVolumeChangeEvent` class defines these events.

Note: On modern Linux distributions, the `StorageVolumeInfo` object only dispatches `storageVolumeMount` and `storageVolumeUnmount` events for physical devices and network drives mounted at particular locations.

The `storageVolume` property of the `StorageVolumeChangeEvent` class is a `StorageVolume` object. The `StorageVolume` class defines basic properties of the storage volume:

- `drive`—The volume drive letter on Windows (`null` on other operating systems)
- `fileSystemType`—The type of file system on the storage volume (such as "FAT", "NTFS", "HFS", or "UFS")
- `isRemoveable`—Whether a volume is removable (`true`) or not (`false`)
- `isWritable`—Whether a volume is writable (`true`) or not (`false`)
- `name`—The name of the volume
- `rootDirectory`—A File object corresponding to the root directory of the volume

The `StorageVolumeChangeEvent` class also includes a `rootDirectory` property. The `rootDirectory` property is a File object referencing the root directory of the storage volume that has been mounted or unmounted.

The `storageVolume` property of the `StorageVolumeChangeEvent` object is undefined (`null`) for an unmounted volume. However you can access the `rootDirectory` property of the event.

The following code outputs the name and file path of a storage volume when it is mounted:

```
air.StorageVolumeInfo.storageVolumeInfo.addEventListener(air.StorageVolumeChangeEvent.STORAGE_VOLUME_MOUNT, onVolumeMount);
function onVolumeMount(event)
{
    air.trace(event.storageVolume.name, event.rootDirectory.nativePath);
}
```

The following code outputs the file path of a storage volume when it is unmounted:

```
air.StorageVolumeInfo.storageVolumeInfo.addEventListener(air.StorageVolumeChangeEvent.STORAGE_VOLUME_UNMOUNT, onVolumeUnmount);
function onVolumeUnmount(event)
{
    air.trace(event.rootDirectory.nativePath);
}
```

The `StorageVolumeInfo.storageVolumeInfo` object includes a `getStorageVolumes()` method. This method returns a vector of `StorageVolume` objects corresponding to the currently mounted storage volumes. The following code shows how to list the names and root directories of all mounted storage volumes:

```
var volumes = air.StorageVolumeInfo.storageVolumeInfo.getStorageVolumes();
for (i = 0; i < volumes.length; i++)
{
    air.trace(volumes[i].name, volumes[i].rootDirectory.nativePath);
}
```

Note: On modern Linux distributions, the `getStorageVolumes()` method returns objects corresponding to physical devices and network drives mounted at particular locations.

The `File.getRootDirectories()` method lists the root directories (see “[Pointing to the file system root](#)” on page 150). However, the `StorageVolume` objects (enumerated by the `StorageVolumeInfo.getStorageVolumes()` method) provides more information about the storage volumes.

You can use the `spaceAvailable` property of the `rootDirectory` property of a `StorageVolume` object to get the space available on a storage volume. (See “[Determining space available on a volume](#)” on page 156.)

More Help topics

[StorageVolume](#)

[StorageVolumeInfo](#)

Reading and writing files

Adobe AIR 1.0 and later

The [FileStream](#) class lets AIR applications read and write to the file system.

Workflow for reading and writing files

Adobe AIR 1.0 and later

The workflow for reading and writing files is as follows.

Initialize a File object that points to the path.

The `File` object represents the path of the file that you want to work with (or a file that you will later create).

```
var file = air.File.documentsDirectory;  
file = file.resolvePath("AIR Test/testFile.txt");
```

This example uses the `File.documentsDirectory` property and the `resolvePath()` method of a `File` object to initialize the `File` object. However, there are many other ways to point a `File` object to a file. For more information, see [“Pointing a File object to a file”](#) on page 151.

Initialize a `FileStream` object.

Call the `open()` method or the `openAsync()` method of the `FileStream` object.

The method you call depends on whether you want to open the file for synchronous or asynchronous operations. Use the `File` object as the `file` parameter of the `open` method. For the `fileMode` parameter, specify a constant from the `FileMode` class that specifies the way in which you will use the file.

For example, the following code initializes a `FileStream` object that is used to create a file and overwrite any existing data:

```
var fileStream = new air.FileStream();  
fileStream.open(file, air.FileMode.WRITE);
```

For more information, see [“Initializing a FileStream object, and opening and closing files”](#) on page 166 and [“FileStream open modes”](#) on page 166.

If you opened the file asynchronously (using the `openAsync()` method), add and set up event listeners for the `FileStream` object.

These event listener methods respond to events dispatched by the `FileStream` object in various situations. These situations include when data is read in from the file, when I/O errors are encountered, or when the complete amount of data to be written has been written.

For details, see [“Asynchronous programming and the events generated by a FileStream object opened asynchronously”](#) on page 170.

Include code for reading and writing data, as needed.

There are many methods of the `FileStream` class related to reading and writing. (They each begin with “read” or “write”.) The method you choose to use to read or write data depends on the format of the data in the target file.

For example, if the data in the target file is UTF-encoded text, you may use the `readUTFBytes()` and `writeUTFBytes()` methods. If you want to deal with the data as byte arrays, you may use the `readByte()`, `readBytes()`, `writeByte()`, and `writeBytes()` methods. For details, see [“Data formats, and choosing the read and write methods to use”](#) on page 171.

If you opened the file asynchronously, then be sure that enough data is available before calling a read method. For details, see [“The read buffer and the bytesAvailable property of a FileStream object”](#) on page 169.

Before writing to a file, if you want to check the amount of disk space available, you can check the `spaceAvailable` property of the `File` object. For more information, see [“Determining space available on a volume”](#) on page 156.

Call the `close()` method of the `FileStream` object when you are done working with the file.

Calling the `close()` method makes the file available to other applications.

For details, see [“Initializing a FileStream object, and opening and closing files”](#) on page 166.

To see a sample application that uses the `FileStream` class to read and write files, see the following articles at the Adobe AIR Developer Center:

- [Building a text-file editor](#)

- [Building a text-file editor](#)
- [Building a text-file editor](#)
- [Reading and writing from an XML preferences file](#)
- [Reading and writing from an XML preferences file](#)

Working with FileStream objects

Adobe AIR 1.0 and later

The `FileStream` class defines methods for opening, reading, and writing files.

FileStream open modes

Adobe AIR 1.0 and later

The `open()` and `openAsync()` methods of a `FileStream` object each include a `fileMode` parameter, which defines some properties for a file stream, including the following:

- The ability to read from the file
- The ability to write to the file
- Whether data will always be appended past the end of the file (when writing)
- What to do when the file does not exist (and when its parent directories do not exist)

The following are the various file modes (which you can specify as the `fileMode` parameter of the `open()` and `openAsync()` methods):

File mode	Description
<code>FileMode.READ</code>	Specifies that the file is open for reading only.
<code>FileMode.WRITE</code>	Specifies that the file is open for writing. If the file does not exist, it is created when the <code>FileStream</code> object is opened. If the file does exist, any existing data is deleted.
<code>FileMode.APPEND</code>	Specifies that the file is open for appending. The file is created if it does not exist. If the file exists, existing data is not overwritten, and all writing begins at the end of the file.
<code>FileMode.UPDATE</code>	Specifies that the file is open for reading and writing. If the file does not exist, it is created. Specify this mode for random read/write access to the file. You can read from any position in the file. When writing to the file, only the bytes written overwrite existing bytes (all other bytes remain unchanged).

Initializing a FileStream object, and opening and closing files

Adobe AIR 1.0 and later

When you open a `FileStream` object, you make it available to read and write data to a file. You open a `FileStream` object by passing a `File` object to the `open()` or `openAsync()` method of the `FileStream` object:

```
var myFile = air.File.documentsDirectory;
myFile = myFile.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.open(myFile, air.FileMode.READ);
```

The `fileMode` parameter (the second parameter of the `open()` and `openAsync()` methods), specifies the mode in which to open the file: for read, write, append, or update. For details, see the previous section, “[FileStream open modes](#)” on page 166.

If you use the `openAsync()` method to open the file for asynchronous file operations, set up event listeners to handle the asynchronous events:

```
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.addEventListener(air.Event.COMPLETE, completeHandler);
myFileStream.addEventListener(air.ProgressEvent.PROGRESS, progressHandler);
myFileStream.addEventListener(air.IOErrorEvent.IOError, errorHandler);
myFileStream.open(myFile, air.FileMode.READ);

function completeHandler(event) {
    // ...
}

function progressHandler(event) {
    // ...
}

function errorHandler(event) {
    // ...
}
```

The file is opened for synchronous or asynchronous operations, depending upon whether you use the `open()` or `openAsync()` method. For details, see [“AIR file basics”](#) on page 145.

If you set the `fileMode` parameter to `FileMode.READ` or `FileMode.UPDATE` in the `open` method of the `FileStream` object, data is read into the read buffer as soon as you open the `FileStream` object. For details, see [“The read buffer and the bytesAvailable property of a FileStream object”](#) on page 169.

You can call the `close()` method of a `FileStream` object to close the associated file, making it available for use by other applications.

The position property of a FileStream object

Adobe AIR 1.0 and later

The `position` property of a `FileStream` object determines where data is read or written on the next read or write method.

Before a read or write operation, set the `position` property to any valid position in the file.

For example, the following code writes the string "hello" (in UTF encoding) at position 8 in the file:

```
var myFile = air.File.documentsDirectory;
myFile = myFile.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.open(myFile, air.FileMode.UPDATE);
myFileStream.position = 8;
myFileStream.writeUTFBytes("hello");
```

When you first open a `FileStream` object, the `position` property is set to 0.

Before a read operation, the value of `position` must be at least 0 and less than the number of bytes in the file (which are existing positions in the file).

The value of the `position` property is modified only in the following conditions:

- When you explicitly set the `position` property.
- When you call a read method.

- When you call a write method.

When you call a read or write method of a `FileStream` object, the `position` property is immediately incremented by the number of bytes that you read or write. Depending on the read method you use, the `position` property is either incremented by the number of bytes you specify to read or by the number of bytes available. When you call a read or write method subsequently, it reads or writes starting at the new position.

```
var myFile = air.File.documentsDirectory;
myFile = myFile.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.open(myFile, air.FileMode.UPDATE);
myFileStream.position = 4000;
alert(myFileStream.position); // 4000
myFileStream.writeBytes(myByteArray, 0, 200);
alert(myFileStream.position); // 4200
```

There is, however, one exception: for a `FileStream` opened in append mode, the `position` property is not changed after a call to a write method. (In append mode, data is always written to the end of the file, independent of the value of the `position` property.)

For a file opened for asynchronous operations, the write operation does not complete before the next line of code is executed. However, you can call multiple asynchronous methods sequentially, and the runtime executes them in order:

```
var myFile = air.File.documentsDirectory;
myFile = myFile.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.openAsync(myFile, air.FileMode.WRITE);
myFileStream.writeUTFBytes("hello");
myFileStream.writeUTFBytes("world");
myFileStream.addEventListener(air.Event.CLOSE, closeHandler);
myFileStream.close();
air.trace("started.");

closeHandler(event)
{
    air.trace("finished.");
}
```

The trace output for this code is the following:

```
started.
finished.
```

You *can* specify the `position` value immediately after you call a read or write method (or at any time), and the next read or write operation will take place starting at that position. For example, note that the following code sets the `position` property right after a call to the `writeBytes()` operation, and the `position` is set to that value (300) even after the write operation completes:

```
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.openAsync(myFile, air.FileMode.UPDATE);
myFileStream.position = 4000;
air.trace(myFileStream.position); // 4000
myFileStream.writeBytes(myByteArray, 0, 200);
myFileStream.position = 300;
air.trace(myFileStream.position); // 300
```

The read buffer and the bytesAvailable property of a FileStream object

Adobe AIR 1.0 and later

When a `FileStream` object with read capabilities (one in which the `fileMode` parameter of the `open()` or `openAsync()` method was set to `READ` or `UPDATE`) is opened, the runtime stores the data in an internal buffer. The `FileStream` object begins reading data into the buffer as soon as you open the file (by calling the `open()` or `openAsync()` method of the `FileStream` object).

For a file opened for synchronous operations (using the `open()` method), you can always set the `position` pointer to any valid position (within the bounds of the file) and begin reading any amount of data (within the bounds of the file), as shown in the following code (which assumes that the file contains at least 100 bytes):

```
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.open(myFile, air.FileMode.READ);
myFileStream.position = 10;
myFileStream.readBytes(myByteArray, 0, 20);
myFileStream.position = 89;
myFileStream.readBytes(myByteArray, 0, 10);
```

Whether a file is opened for synchronous or asynchronous operations, the read methods always read from the "available" bytes, represented by the `bytesAvailable` property. When reading synchronously, all of the bytes of the file are available all of the time. When reading asynchronously, the bytes become available starting at the position specified by the `position` property, in a series of asynchronous buffer fills signaled by `progress` events.

For files opened for *synchronous* operations, the `bytesAvailable` property is always set to represent the number of bytes from the `position` property to the end of the file (all bytes in the file are always available for reading).

For files opened for *asynchronous* operations, you need to ensure that the read buffer has consumed enough data before calling a read method. For a file opened asynchronously, as the read operation progresses, the data from the file, starting at the `position` specified when the read operation started, is added to the buffer, and the `bytesAvailable` property increments with each byte read. The `bytesAvailable` property indicates the number of bytes available starting with the byte at the position specified by the `position` property to the end of the buffer. Periodically, the `FileStream` object sends a `progress` event.

For a file opened asynchronously, as data becomes available in the read buffer, the `FileStream` object periodically dispatches the `progress` event. For example, the following code reads data into a `ByteArray` object, `bytes`, as it is read into the buffer:

```
var bytes = new air.ByteArray();
var myFile = new air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.addEventListener(air.ProgressEvent.PROGRESS, progressHandler);
myFileStream.openAsync(myFile, air.FileMode.READ);

function progressHandler(event)
{
    myFileStream.readBytes(bytes, myFileStream.position, myFileStream.bytesAvailable);
}
```

For a file opened asynchronously, only the data in the read buffer can be read. Furthermore, as you read the data, it is removed from the read buffer. For read operations, you need to ensure that the data exists in the read buffer before calling the read operation. For example, the following code reads 8000 bytes of data starting from position 4000 in the file:

```
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.addEventListener(air.ProgressEvent.PROGRESS, progressHandler);
myFileStream.addEventListener(air.Event.COMPLETE, completed);
myFileStream.openAsync(myFile, air.FileMode.READ);
myFileStream.position = 4000;

var str = "";

function progressHandler(event)
{
    if (myFileStream.bytesAvailable > 8000 )
    {
        str += myFileStream.readMultiByte(8000, "iso-8859-1");
    }
}
```

During a write operation, the `FileStream` object does not read data into the read buffer. When a write operation completes (all data in the write buffer is written to the file), the `FileStream` object starts a new read buffer (assuming that the associated `FileStream` object was opened with read capabilities), and starts reading data into the read buffer, starting from the position specified by the `position` property. The `position` property may be the position of the last byte written, or it may be a different position, if the user specifies a different value for the `position` object after the write operation.

Asynchronous programming and the events generated by a `FileStream` object opened asynchronously **Adobe AIR 1.0 and later**

When a file is opened asynchronously (using the `openAsync()` method), reading and writing files are done asynchronously. As data is read into the read buffer and as output data is being written, other ActionScript code can execute.

This means that you need to register for events generated by the `FileStream` object opened asynchronously.

By registering for the `progress` event, you can be notified as new data becomes available for reading, as in the following code:

```
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.addEventListener(air.ProgressEvent.PROGRESS, progressHandler);
myFileStream.openAsync(myFile, air.FileMode.READ);
var str = "";

function progressHandler(event)
{
    str += myFileStream.readMultiByte(myFileStream.bytesAvailable, "iso-8859-1");
}
```

You can read the entire data by registering for the `complete` event, as in the following code:


```
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.addEventListener(air.Event.COMPLETE, completed);
myFileStream.openAsync(myFile, air.FileMode.READ);
var str = "";
function completeHandler(event)
{
    str = myFileStream.readMultiByte(myFileStream.bytesAvailable, "iso-8859-1");
}
```

In much the same way that input data is buffered to enable asynchronous reading, data that you write on an asynchronous stream is buffered and written to the file asynchronously. As data is written to a file, the `FileStream` object periodically dispatches an `OutputProgressEvent` object. An `OutputProgressEvent` object includes a `bytesPending` property that is set to the number of bytes remaining to be written. You can register for the `outputProgress` event to be notified as this buffer is actually written to the file, perhaps in order to display a progress dialog. However, in general, it is not necessary to do so. In particular, you may call the `close()` method without concern for the unwritten bytes. The `FileStream` object will continue writing data and the `close` event will be delivered after the final byte is written to the file and the underlying file is closed.

Data formats, and choosing the read and write methods to use Adobe AIR 1.0 and later

Every file is a set of bytes on a disk. In ActionScript, the data from a file can always be represented as a `ByteArray`. For example, the following code reads the data from a file into a `ByteArray` object named `bytes`:

```
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.addEventListener(air.Event.COMPLETE, completeHandler);
myFileStream.openAsync(myFile, air.FileMode.READ);
var bytes = new air.ByteArray();

function completeHandler(event)
{
    myFileStream.readBytes(bytes, 0, myFileStream.bytesAvailable);
}
```

Similarly, the following code writes data from a `ByteArray` named `bytes` to a file:

```
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.open(myFile, air.FileMode.WRITE);
myFileStream.writeBytes(bytes, 0, bytes.length);
```

However, often you do not want to store the data in an ActionScript `ByteArray` object. And often the data file is in a specified file format.

For example, the data in the file may be in a text file format, and you may want to represent such data in a `String` object.

For this reason, the `FileStream` class includes read and write methods for reading and writing data to and from types other than `ByteArray` objects. For example, the `readMultiByte()` method lets you read data from a file and store it to a string, as in the following code:

```
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.addEventListener(air.Event.COMPLETE, completed);
myFileStream.openAsync(myFile, air.FileMode.READ);
var str = "";

function completeHandler(event)
{
    str = myFileStream.readMultiByte(myFileStream.bytesAvailable, "iso-8859-1");
}
```

The second parameter of the `readMultiByte()` method specifies the text format that ActionScript uses to interpret the data ("iso-8859-1" in the example). Adobe AIR supports common character set encodings (see Supported character sets).

The `FileStream` class also includes the `readUTFBytes()` method, which reads data from the read buffer into a string using the UTF-8 character set. Since characters in the UTF-8 character set are of variable length, do not use `readUTFBytes()` in a method that responds to the `progress` event, since the data at the end of the read buffer may represent an incomplete character. (This is also true when using the `readMultiByte()` method with a variable-length character encoding.) For this reason, read the entire set of data when the `FileStream` object dispatches the `complete` event.

There are also similar write methods, `writeMultiByte()` and `writeUTFBytes()`, for working with `String` objects and text files.

The `readUTF()` and the `writeUTF()` methods (not to be confused with `readUTFBytes()` and `writeUTFBytes()`) also read and write the text data to a file, but they assume that the text data is preceded by data specifying the length of the text data, which is not a common practice in standard text files.

Some UTF-encoded text files begin with a "UTF-BOM" (byte order mark) character that defines the endianness as well as the encoding format (such as UTF-16 or UTF-32).

For an example of reading and writing to a text file, see "[Example: Reading an XML file into an XML object](#)" on page 173.

The `readObject()` and `writeObject()` are convenient ways to store and retrieve data for complex ActionScript objects. The data is encoded in AMF (ActionScript Message Format). Adobe AIR, Flash Player, Flash Media Server, and Flex Data Services include APIs for working with data in this format.

There are some other read and write methods (such as `readDouble()` and `writeDouble()`). However, if you use these, make sure that the file format matches the formats of the data defined by these methods.

File formats are often more complex than simple text formats. For example, an MP3 file includes compressed data that can only be interpreted with the decompression and decoding algorithms specific to MP3 files. MP3 files also may include ID3 tags that contain meta tag information about the file (such as the title and artist for a song). There are multiple versions of the ID3 format, but the simplest (ID3 version 1) is discussed in the "[Example: Reading and writing data with random access](#)" on page 175 section.

Other files formats (for images, databases, application documents, and so on) have different structures, and to work with their data in ActionScript, you must understand how the data is structured.

Using the load() and save() methods

Flash Player 10 and later, Adobe AIR 1.5 and later

Flash Player 10 added the `load()` and `save()` methods to the `FileReference` class. These methods are also in AIR 1.5, and the `File` class inherits the methods from the `FileReference` class. These methods were designed to provide a secure means for users to load and save file data in Flash Player. However, AIR applications can also use these methods as an easy way to load and save files asynchronously.

For example, the following code saves a string to a text file:

```
var file = air.File.applicationStorageDirectory.resolvePath("test.txt");
var str = "Hello.";
file.addEventListener(air.Event.COMPLETE, fileSaved);
file.save(str);
function fileSaved(event)
{
    air.trace("Done.");
}
```

The `data` parameter of the `save()` method can take a `String` or `ByteArray` value. When the argument is a `String` value, the method saves the file as a UTF-8–encoded text file.

When this code sample executes, the application displays a dialog box in which the user selects the saved file destination.

The following code loads a string from a UTF-8–encoded text file:

```
var file = air.File.applicationStorageDirectory.resolvePath("test.txt");
file.addEventListener(air.Event.COMPLETE, loaded);
file.load();
var str;
function loaded(event)
{
    var bytes = file.data;
    str = bytes.readUTFBytes(bytes.length);
    air.trace(str);
}
```

The `FileStream` class provides more functionality than that provided by the `load()` and `save()` methods:

- Using the `FileStream` class, you can read and write data both synchronously and asynchronously.
- Using the `FileStream` class lets you write incrementally to a file.
- Using the `FileStream` class lets you open a file for random access (both reading from and writing to any section of the file).
- The `FileStream` class lets you specify the type of file access you have to the file, by setting the `fileMode` parameter of the `open()` or `openAsync()` method.
- The `FileStream` class lets you save data to files without presenting the user with an Open or Save dialog box.
- You can directly use types other than byte arrays when reading data with the `FileStream` class.

Example: Reading an XML file into an XML object

Adobe AIR 1.0 and later

The following examples demonstrate how to read and write to a text file that contains XML data.

To read from the file, initialize the File and FileStream objects, call the `readUTFBytes()` method of the FileStream and convert the string to an XML object:

```
var file = air.File.documentsDirectory.resolvePath("AIR Test/preferences.xml");
var fileStream = new air.FileStream();
fileStream.open(file, air.FileMode.READ);
var prefsXML = fileStream.readUTFBytes(fileStream.bytesAvailable);
fileStream.close();
```

Similarly, writing the data to the file is as easy as setting up appropriate File and FileStream objects, and then calling a write method of the FileStream object. Pass the string version of the XML data to the write method as in the following code:

```
var file = air.File.documentsDirectory.resolvePath("AIR Test/preferences.xml");
fileStream = new air.FileStream();
fileStream.open(file, air.FileMode.WRITE);

var outputString = '<?xml version="1.0" encoding="utf-8"?>\n';
outputString += '<prefs><autoSave>true</autoSave></prefs>';

fileStream.writeUTFBytes(outputString);
fileStream.close();
```

These examples use the `readUTFBytes()` and `writeUTFBytes()` methods, because they assume that the files are in UTF-8 format. If not, you may need to use a different method (see [“Data formats, and choosing the read and write methods to use”](#) on page 171).

The previous examples use FileStream objects opened for synchronous operation. You can also open files for asynchronous operations (which rely on event listener functions to respond to events). For example, the following code shows how to read an XML file asynchronously:

```
var file = air.File.documentsDirectory.resolvePath("AIR Test/preferences.xml");
var fileStream= new air.FileStream();
fileStream.addEventListener(air.Event.COMPLETE, processXMLData);
fileStream.openAsync(file, air.FileMode.READ);
var prefsXML;

function processXMLData(event)
{
    var xmlString = fileStream.readUTFBytes(fileStream.bytesAvailable);
    prefsXML = domParser.parseFromString(xmlString, "text/xml");
    fileStream.close();
}
```

The `processXMLData()` method is invoked when the entire file is read into the read buffer (when the FileStream object dispatches the `complete` event). It calls the `readUTFBytes()` method to get a string version of the read data, and it creates an XML object, `prefsXML`, based on that string.

To see a sample application that shows these capabilities, see [Reading and writing from an XML preferences file](#).

To see a sample application that shows these capabilities, see [Reading and writing from an XML Preferences File](#).

Example: Reading and writing data with random access

Adobe AIR 1.0 and later

MP3 files can include ID3 tags, which are sections at the beginning or end of the file that contain meta data identifying the recording. The ID3 tag format itself has different revisions. This example describes how to read and write from an MP3 file that contains the simplest ID3 format (ID3 version 1.0) using "random access to file data", which means that it reads from and writes to arbitrary locations in the file.

An MP3 file that contains an ID3 version 1 tag includes the ID3 data at the end of the file, in the final 128 bytes.

When accessing a file for random read/write access, it is important to specify `FileMode.UPDATE` as the `fileMode` parameter for the `open()` or `openAsync()` method:

```
var file = air.File.documentsDirectory.resolvePath("My Music/Sample ID3 v1.mp3");
var fileStr = new air.FileStream();
fileStr.open(file, air.FileMode.UPDATE);
```

This lets you both read and write to the file.

Upon opening the file, you can set the `position` pointer to the position 128 bytes before the end of the file:

```
fileStr.position = file.size - 128;
```

This code sets the `position` property to this location in the file because the ID3 v1.0 format specifies that the ID3 tag data is stored in the last 128 bytes of the file. The specification also says the following:

- The first 3 bytes of the tag contain the string "TAG".
- The next 30 characters contain the title for the MP3 track, as a string.
- The next 30 characters contain the name of the artist, as a string.
- The next 30 characters contain the name of the album, as a string.
- The next 4 characters contain the year, as a string.
- The next 30 characters contain the comment, as a string.
- The next byte contains a code indicating the track's genre.
- All text data is in ISO 8859-1 format.

The `id3TagRead()` method checks the data after it is read in (upon the `complete` event):

```
function id3TagRead()
{
    if (fileStr.readMultiByte(3, "iso-8859-1").match(/tag/i))
    {
        var id3Title = fileStr.readMultiByte(30, "iso-8859-1");
        var id3Artist = fileStr.readMultiByte(30, "iso-8859-1");
        var id3Album = fileStr.readMultiByte(30, "iso-8859-1");
        var id3Year = fileStr.readMultiByte(4, "iso-8859-1");
        var id3Comment = fileStr.readMultiByte(30, "iso-8859-1");
        var id3GenreCode = fileStr.readByte().toString(10);
    }
}
```

You can also perform a random-access write to the file. For example, you could parse the `id3Title` variable to ensure that it is correctly capitalized (using methods of the `String` class), and then write a modified string, called `newTitle`, to the file, as in the following:

```
fileStr.position = file.length - 125;    // 128 - 3
fileStr.writeMultiByte(newTitle, "iso-8859-1");
```

To conform with the ID3 version 1 standard, the length of the `newTitle` string should be 30 characters, padded at the end with the character code 0 (`String.fromCharCode(0)`).

Chapter 12: Drag and drop in AIR

Adobe AIR 1.0 and later

Use the classes in the Adobe® AIR™ drag-and-drop API to support user-interface drag-and-drop gestures. A *gesture* in this sense is an action by the user, mediated by both the operating system and your application, expressing an intent to copy, move, or link information. A *drag-out* gesture occurs when the user drags an object out of a component or application. A *drag-in* gesture occurs when the user drags in an object from outside a component or application.

With the drag-and-drop API, you can allow a user to drag data between applications and between components within an application. Supported transfer formats include:

- Bitmaps
- Files
- HTML-formatted text
- Text
- URLs

Drag and drop in HTML

Adobe AIR 1.0 and later

To drag data into and out of an HTML-based application (or into and out of the HTML displayed in an HTMLLoader), you can use HTML drag and drop events. The HTML drag-and-drop API allows you to drag to and from DOM elements in the HTML content.

Note: You can also use the AIR `NativeDragEvent` and `NativeDragManager` APIs by listening for events on the `HTMLLoader` object containing the HTML content. However, the HTML API is better integrated with the HTML DOM and gives you control of the default behavior. The `NativeDragEvent` and `NativeDragManager` APIs are not commonly used in HTML-based applications and so are not covered in the [Adobe AIR API Reference for HTML Developers](#). For more information about using these classes, refer to the [Adobe ActionScript 3.0 Developer's Guide](#) and the [ActionScript 3.0 Reference for the Adobe Flash Platform](#).

Default drag-and-drop behavior

Adobe AIR 1.0 and later

The HTML environment provides default behavior for drag-and-drop gestures involving text, images, and URLs. Using the default behavior, you can always drag these types of data out of an element. However, you can only drag text into an element and only to elements in an editable region of a page. When you drag text between or within editable regions of a page, the default behavior performs a move action. When you drag text to an editable region from a non-editable region or from outside the application, then the default behavior performs a copy action.

You can override the default behavior by handling the drag-and-drop events yourself. To cancel the default behavior, you must call the `preventDefault()` methods of the objects dispatched for the drag-and-drop events. You can then insert data into the drop target and remove data from the drag source as necessary to perform the chosen action.

By default, the user can select and drag any text, and drag images and links. You can use the WebKit CSS property, `-webkit-user-select` to control how any HTML element can be selected. For example, if you set `-webkit-user-select` to `none`, then the element contents are not selectable and so cannot be dragged. You can also use the `-webkit-user-drag` CSS property to control whether an element as a whole can be dragged. However, the contents of the element are treated separately. The user could still drag a selected portion of the text. For more information, see “[CSS in AIR](#)” on page 16.

Drag-and-drop events in HTML

Adobe AIR 1.0 and later

The events dispatched by the initiator element from which a drag originates, are:

Event	Description
dragstart	Dispatched when the user starts the drag gesture. The handler for this event can prevent the drag, if necessary, by calling the <code>preventDefault()</code> method of the event object. To control whether the dragged data can be copied, linked, or moved, set the <code>effectAllowed</code> property. Selected text, images, and links are put onto the clipboard by the default behavior, but you can set different data for the drag gesture using the <code>dataTransfer</code> property of the event object.
drag	Dispatched continuously during the drag gesture.
dragend	Dispatched when the user releases the mouse button to end the drag gesture.

The events dispatched by a drag target are:

Event	Description
dragover	Dispatched continuously while the drag gesture remains within the element boundaries. The handler for this event should set the <code>dataTransfer.dropEffect</code> property to indicate whether the drop will result in a copy, move, or link action if the user releases the mouse.
dragenter	Dispatched when the drag gesture enters the boundaries of the element. If you change any properties of a <code>dataTransfer</code> object in a <code>dragenter</code> event handler, those changes are quickly overridden by the next <code>dragover</code> event. On the other hand, there is a short delay between a <code>dragenter</code> and the first <code>dragover</code> event that can cause the cursor to flash if different properties are set. In many cases, you can use the same event handler for both events.
dragleave	Dispatched when the drag gesture leaves the element boundaries.
drop	Dispatched when the user drops the data onto the element. The data being dragged can only be accessed within the handler for this event.

The event object dispatched in response to these events is similar to a mouse event. You can use mouse event properties such as `(clientX, clientY)` and `(screenX, screenY)`, to determine the mouse position.

The most important property of a drag event object is `dataTransfer`, which contains the data being dragged. The `dataTransfer` object itself has the following properties and methods:

Property or Method	Description
effectAllowed	The effect allowed by the source of the drag. Typically, the handler for the dragstart event sets this value. See "Drag effects in HTML" on page 180.
dropEffect	The effect chosen by the target or the user. If you set the <code>dropEffect</code> in a <code>dragover</code> or <code>dragenter</code> event handler, then AIR updates the mouse cursor to indicate the effect that occurs if the user releases the mouse. If the <code>dropEffect</code> set does not match one of the allowed effects, no drop is allowed and the <i>unavailable</i> cursor is displayed. If you have not set a <code>dropEffect</code> in response to the latest <code>dragover</code> or <code>dragenter</code> event, then the user can choose from the allowed effects with the standard operating system modifier keys. The final effect is reported by the <code>dropEffect</code> property of the object dispatched for <code>dragend</code> . If the user abandons the drop by releasing the mouse outside an eligible target, then <code>dropEffect</code> is set to <code>none</code> .
types	An array containing the MIME type strings for each data format present in the <code>dataTransfer</code> object.
getData(mimeType)	Gets the data in the format specified by the <code>mimeType</code> parameter. The <code>getData()</code> method can only be called in response to the <code>drop</code> event.
setData(mimeType)	Adds data to the <code>dataTransfer</code> in the format specified by the <code>mimeType</code> parameter. You can add data in multiple formats by calling <code>setData()</code> for each MIME type. Any data placed in the <code>dataTransfer</code> object by the default drag behavior is cleared. The <code>setData()</code> method can only be called in response to the <code>dragstart</code> event.
clearData(mimeType)	Clears any data in the format specified by the <code>mimeType</code> parameter.
setDragImage(image, offsetX, offsetY)	Sets a custom drag image. The <code>setDragImage()</code> method can only be called in response to the <code>dragstart</code> event and only when an entire HTML element is dragged by setting its <code>-webkit-user-drag</code> CSS style to <code>element</code> . The <code>image</code> parameter can be a JavaScript Element or Image object.

MIME types for the HTML drag-and-drop

Adobe AIR 1.0 and later

The MIME types to use with the `dataTransfer` object of an HTML drag-and-drop event include:

Data format	MIME type
Text	"text/plain"
HTML	"text/html"
URL	"text/uri-list"
Bitmap	"image/x-vnd.adobe.air.bitmap"
File list	"application/x-vnd.adobe.air.file-list"

You can also use other MIME strings, including application-defined strings. However, other applications may not be able to recognize or use the transferred data. It is your responsibility to add data to the `dataTransfer` object in the expected format.

Important: Only code running in the application sandbox can access dropped files. Attempting to read or set any property of a `File` object within a non-application sandbox generates a security error. See ["Handling file drops in non-application HTML sandboxes"](#) on page 184 for more information.

Drag effects in HTML

Adobe AIR 1.0 and later

The initiator of the drag gesture can limit the allowed drag effects by setting the `dataTransfer.effectAllowed` property in the handler for the `dragstart` event. The following string values can be used:

String value	Description
"none"	No drag operations are allowed.
"copy"	The data will be copied to the destination, leaving the original in place.
"link"	The data will be shared with the drop destination using a link back to the original.
"move"	The data will be copied to the destination and removed from the original location.
"copyLink"	The data can be copied or linked.
"copyMove"	The data can be copied or moved.
"linkMove"	The data can be linked or moved.
"all"	The data can be copied, moved, or linked. <i>All</i> is the default effect when you prevent the default behavior.

The target of the drag gesture can set the `dataTransfer.dropEffect` property to indicate the action that is taken if the user completes the drop. If the drop effect is one of the allowed actions, then the system displays the appropriate copy, move, or link cursor. If not, then the system displays the *unavailable* cursor. If no drop effect is set by the target, the user can choose from the allowed actions with the modifier keys.

Set the `dropEffect` value in the handlers for both the `dragover` and `dragenter` events:

```
function doDragStart(event) {
    event.dataTransfer.setData("text/plain", "Text to drag");
    event.dataTransfer.effectAllowed = "copyMove";
}

function doDragOver(event) {
    event.dataTransfer.dropEffect = "copy";
}

function doDragEnter(event) {
    event.dataTransfer.dropEffect = "copy";
}
```

Note: Although you should always set the `dropEffect` property in the handler for `dragenter`, be aware that the next `dragover` event resets the property to its default value. Set `dropEffect` in response to both events.

Dragging data out of an HTML element

Adobe AIR 1.0 and later

The default behavior allows most content in an HTML page to be copied by dragging. You can control the content allowed to be dragged using CSS properties `-webkit-user-select` and `-webkit-user-drag`.

Override the default drag-out behavior in the handler for the `dragstart` event. Call the `setData()` method of the `dataTransfer` property of the event object to put your own data into the drag gesture.

To indicate which drag effects a source object supports when you are not relying on the default behavior, set the `dataTransfer.effectAllowed` property of the event object dispatched for the `dragstart` event. You can choose any combination of effects. For example, if a source element supports both *copy* and *link* effects, set the property to `"copyLink"`.

Setting the dragged data

Flash Player 9 and later, Adobe AIR 1.0 and later

Add the data for the drag gesture in the handler for the `dragstart` event with the `dataTransfer` property. Use the `dataTransfer.setData()` method to put data onto the clipboard, passing in the MIME type and the data to transfer.

For example, if you had an image element in your application, with the id *imageOfGeorge*, you could use the following `dragstart` event handler. This example adds representations of a picture of George in several data formats, which increases the likelihood that other applications can use the dragged data.

```
function dragStartHandler(event) {
    event.dataTransfer.effectAllowed = "copy";

    var dragImage = document.getElementById("imageOfGeorge");
    var dragFile = new air.File(dragImage.src);
    event.dataTransfer.setData("text/plain", "A picture of George");
    event.dataTransfer.setData("image/x-vnd.adobe.air.bitmap", dragImage);
    event.dataTransfer.setData("application/x-vnd.adobe.air.file-list",
        new Array(dragFile));
}
```

Note: When you call the `setData()` method of `dataTransfer` object, no data is added by the default drag-and-drop behavior.

Dragging data into an HTML element

Adobe AIR 1.0 and later

The default behavior only allows text to be dragged into editable regions of the page. You can specify that an element and its children can be made editable by including the `contenteditable` attribute in the opening tag of the element. You can also make an entire document editable by setting the document object `designMode` property to `"on"`.

You can support alternate drag-in behavior on a page by handling the `dragenter`, `dragover`, and `drop` events for any elements that can accept dragged data.

Enabling drag-in

Adobe AIR 1.0 and later

To handle the drag-in gesture, you must first cancel the default behavior. Listen for the `dragenter` and `dragover` events on any HTML elements you want to use as drop targets. In the handlers for these events, call the `preventDefault()` method of the dispatched event object. Canceling the default behavior allows non-editable regions to receive a drop.

Getting the dropped data

Adobe AIR 1.0 and later

You can access the dropped data in the handler for the `ondrop` event:

```
function doDrop(event){
    droppedText = event.dataTransfer.getData("text/plain");
}
```

Use the `dataTransfer.getData()` method to read the data onto the clipboard, passing in the MIME type of the data format to read. You can find out which data formats are available using the `types` property of the `dataTransfer` object. The `types` array contains the MIME type string of each available format.

When you cancel the default behavior in the `dragenter` or `dragover` events, you are responsible for inserting any dropped data into its proper place in the document. No API exists to convert a mouse position into an insertion point within an element. This limitation can make it difficult to implement insertion-type drag gestures.

Example: Overriding the default HTML drag-in behavior

Adobe AIR 1.0 and later

This example implements a drop target that displays a table showing each data format available in the dropped item.

The default behavior is used to allow text, links, and images to be dragged within the application. The example overrides the default drag-in behavior for the `div` element that serves as the drop target. The key step to enabling non-editable content to accept a drag-in gesture is to call the `preventDefault()` method of the event object dispatched for both the `dragenter` and `dragover` events. In response to a `drop` event, the handler converts the transferred data into an HTML row element and inserts the row into a table for display.

```
<html>
<head>
<title>Drag-and-drop</title>
<script language="javascript" type="text/javascript" src="AIRAliases.js"></script>
<script language="javascript">
    function init(){
        var target = document.getElementById('target');
        target.addEventListener("dragenter", dragEnterOverHandler);
        target.addEventListener("dragover", dragEnterOverHandler);
        target.addEventListener("drop", dropHandler);

        var source = document.getElementById('source');
        source.addEventListener("dragstart", dragStartHandler);
        source.addEventListener("dragend", dragEndHandler);

        emptyRow = document.getElementById("emptyTargetRow");
    }

    function dragStartHandler(event){
        event.dataTransfer.effectAllowed = "copy";
    }

    function dragEndHandler(event){
        air.trace(event.type + ": " + event.dataTransfer.dropEffect);
    }
</script>
</head>
<body>
    <div id="source">
        <input type="text" value="Text" />
    </div>
    <div id="target">
        <table border="1">
            <tr>
                <td><div id="emptyTargetRow"></div>
            </tr>
        </table>
    </div>
</body>
</html>
```

```
    }

    function dragEnterOverHandler(event){
        event.preventDefault();
    }

    var emptyRow;
    function dropHandler(event){
        for(var prop in event){
            air.trace(prop + " = " + event[prop]);
        }
        var row = document.createElement('tr');
        row.innerHTML = "<td>" + event.dataTransfer.getData("text/plain") + "</td>" +
            "<td>" + event.dataTransfer.getData("text/html") + "</td>" +
            "<td>" + event.dataTransfer.getData("text/uri-list") + "</td>" +
            "<td>" + event.dataTransfer.getData("application/x-vnd.adobe.air.file-list") +
            "</td>";

        var imageCell = document.createElement('td');
        if((event.dataTransfer.types.toString()).search("image/x-vnd.adobe.air.bitmap") > -
1){
            imageCell.appendChild(event.dataTransfer.getData("image/x-
vnd.adobe.air.bitmap"));
        }
        row.appendChild(imageCell);
        var parent = emptyRow.parentNode;
        parent.insertBefore(row, emptyRow);
    }
}
</script>
</head>
<body onLoad="init()" style="padding:5px">
<div>
    <h1>Source</h1>
    <p>Items to drag:</p>
    <ul id="source">
        <li>Plain text.</li>
        <li>HTML <b>formatted</b> text.</li>
        <li>A <a href="http://www.adobe.com">URL.</a></li>
        <li></li>
        <li style="-webkit-user-drag:none;">
            Uses "-webkit-user-drag:none" style.
        </li>
    </ul>
</div>
</body>
</html>
```

```
        <li style="-webkit-user-select:none;">
          Uses "-webkit-user-select:none" style.
        </li>

      </ul>
    </div>
    <div id="target" style="border-style:dashed;">
      <h1 >Target</h1>
      <p>Drag items from the source list (or elsewhere).</p>
      <table id="displayTable" border="1">
        <tr><th>Plain text</th><th>Html text</th><th>URL</th><th>File list</th><th>Bitmap
Data</th></tr>
        <tr
          id="emptyTargetRow"><td>&nbsp;</td><td>&nbsp;</td><td>&nbsp;</td><td>&nbsp;</td><td>&nbsp;</td><td>&nbsp;</td></tr>
        </table>
      </div>
    </div>
  </body>
</html>
```

Handling file drops in non-application HTML sandboxes

Adobe AIR 1.0 and later

Non-application content cannot access the File objects that result when files are dragged into an AIR application. Nor is it possible to pass one of these File objects to application content through a sandbox bridge. (The object properties must be accessed during serialization.) However, you can still drop files in your application by listening for the AIR nativeDragDrop events on the HTMLLoader object.

Normally, if a user drops a file into a frame that hosts non-application content, the drop event does not propagate from the child to the parent. However, since the events dispatched by the HTMLLoader (which is the container for all HTML content in an AIR application) are not part of the HTML event flow, you can still receive the drop event in application content.

To receive the event for a file drop, the parent document adds an event listener to the HTMLLoader object using the reference provided by `window.htmlLoader`:

```
window.htmlLoader.addEventListener("nativeDragDrop", function(event) {
    var filelist = event.clipboard.getData(air.ClipboardFormats.FILE_LIST_FORMAT);
    air.trace(filelist[0].url);
});
```

The NativeDragEvent objects behave like their HTML event counterparts, but the names and data types of some of the properties and methods are different. For example, the HTML event `dataTransfer` property serves the same purpose as the ActionScript event `clipboard` property. For more information about using these classes, refer to [Adobe ActionScript 3.0 Developer's Guide](#) and the [ActionScript 3.0 Reference for the Adobe Flash Platform](#).

The following example uses a parent document that loads a child page into a remote sandbox (`http://localhost/`). The parent listens for the `nativeDragDrop` event on the HTMLLoader object and traces out the file url.

```
<html>
<head>
<title>Drag-and-drop in a remote sandbox</title>
<script language="javascript" type="text/javascript" src="AIRAliases.js"></script>
<script language="javascript">
    window.htmlLoader.addEventListener("nativeDragDrop",function(event){
        var filelist = event.clipboard.getData(air.ClipboardFormats.FILE_LIST_FORMAT);
        air.trace(filelist[0].url);
    });
</script>
</head>
<body>
    <iframe src="child.html"
        sandboxRoot="http://localhost/"
        documentRoot="app:/"
        frameBorder="0" width="100%" height="100%">
    </iframe>
</body>
</html>
```

The child document must present a valid drop target by calling the Event object `preventDefault()` method in the HTML `dragenter` and `dragover` event handlers. Otherwise, the drop event can never occur.

```
<html>
<head>
    <title>Drag and drop target</title>
    <script language="javascript" type="text/javascript">
        function preventDefault(event){
            event.preventDefault();
        }
    </script>
</head>
<body ondragenter="preventDefault(event)" ondragover="preventDefault(event)">
<div>
<h1>Drop Files Here</h1>
</div>
</body>
</html>
```

Dropping file promises

Adobe AIR 2 and later

A file promise is a drag-and-drop clipboard format that allows a user to drag a file that does not yet exist out of an AIR application. For example, using file promises, your application could allow a user to drag a proxy icon to a desktop folder. The proxy icon represents a file or some data known to be available at a URL. After the user drops the icon, the runtime downloads the data and writes the file to the drop location.

You can use the `URLFilePromise` class in an AIR application to drag-and-drop files accessible at a URL. The `URLFilePromise` implementation is provided in the `aircore` library as part of the AIR 2 SDK. Use either the `aircore.swc` or `aircore.swf` file found in the SDK `frameworks/libs/air` directory.

Alternately, you can implement your own file promise logic using the `IFilePromise` interface (which is defined in the runtime `flash.desktop` package).

File promises are similar in concept to deferred rendering using a data handler function on the clipboard. Use file promises instead of deferred rendering when dragging and dropping files. The deferred rendering technique can lead to undesirable pauses in the drag gesture as the data is generated or downloaded. Use deferred rendering for copy and paste operations (for which file promises are not supported).

Limitations when using file promises

File promises have the following limitations compared to other data formats that you can put in a drag-and-drop clipboard:

- File promises can only be dragged out of an AIR application; they cannot be dropped into an AIR application.
- File promises are not supported on all operating systems. Use the `Clipboard.supportsFilePromise` property to test whether file promises are supported on the host system. On systems that do not support file promises, you should provide an alternative mechanism for downloading or generating the file data.
- File promises cannot be used with the copy-and-paste clipboard (`Clipboard.generalClipboard`).

More Help topics

[flash.desktop.IFilePromise](#)

[air.desktop.URLFilePromise](#)

Dropping remote files

Adobe AIR 2 and later

Use the `URLFilePromise` class to create file promise objects representing files or data available at a URL. Add one or more file promise objects to the clipboard using the `FILE_PROMISE_LIST` clipboard format. In the following example, a single file, available at `http://www.example.com/foo.txt`, is downloaded and saved to the drop location as `bar.txt`. (The remote and the local file names do not have to match.)


```
<html>
<head>
<script src="AIRAliases.js"></script>
<script src="aircore.swf" type="application/x-shockwave-flash"></script>
<script language="javascript">
    function init(){
        var source = document.getElementById('source');
        source.addEventListener("dragstart", dragStartHandler);
    }

    function dragStartHandler(event){
        event.preventDefault();
        startDrag();
    }
    function startDrag()
    {
        var filePromise = new air.URLFilePromise(); //defined in aircore.swf
        filePromise.request = new air.URLRequest("http://example.com/foo.txt");
        filePromise.relativePath = "bar.txt";
        var fileList = new Array( filePromise );
        var clipboard = new air.Clipboard();
        clipboard.setData( air.ClipboardFormats.FILE_PROMISE_LIST_FORMAT, fileList );
        air.NativeDragManager.doDrag( window.htmlLoader, clipboard );
    }
</script>
</head>
<body onLoad="init()">
    <p id="source" style="-webkit-user-drag:element; -webkit-user-select:none;">
        Drag to file system
    </p>
</body>
</html>
```

You can allow the user to drag more than one file at a time by adding more file promise objects to the array assigned to the clipboard. You can also specify subdirectories in the `relativePath` property so that some or all of the files included in the operation are placed in a subfolder relative to the drop location.

The following example illustrates how to initiate a drag operation that includes multiple file promises. In this example, an html page, *article.html*, is put on the clipboard as a file promise, along with its two linked image files. The images are copied into an *images* subfolder so that the relative links are maintained.

```
<html>
<head>
<script src="AIRAliases.js"></script>
<script src="aircore.swf" type="application/x-shockwave-flash"></script>
<script language="javascript">
    function init(){
        var source = document.getElementById('source');
        source.addEventListener("dragstart", dragStartHandler);
    }

    function dragStartHandler(event){
        event.preventDefault();
        startDrag();
    }
    function startDrag()
    {
        var filePromise = new air.URLFilePromise();
        filePromise.request = new air.URLRequest("http://example.com/article.html");
        filePromise.relativePath = "article.html";

        var image1Promise = new air.URLFilePromise();
        image1Promise.request = new air.URLRequest("http://example.com/images/img_1.jpg");
        image1Promise.relativePath = "images/img_1.html";
        var image2Promise = new air.URLFilePromise();
        image2Promise.request = new air.URLRequest("http://example.com/images/img_2.jpg");
        image2Promise.relativePath = "images/img_2.jpg";

        //Put the promise objects onto the clipboard inside an array
        var fileList = new Array( filePromise, image1Promise, image2Promise );
        var clipboard = new air.Clipboard();
        clipboard.setData( air.ClipboardFormats.FILE_PROMISE_LIST_FORMAT, fileList );
        air.NativeDragManager.doDrag( window.htmlLoader, clipboard );
    }
</script>
</head>
<body onLoad="init()">
    <p id="source" style="-webkit-user-drag:element; -webkit-user-select:none;">
        Drag to file system
    </p>
</body>
</html>
```

Implementing the IFilePromise interface

Adobe AIR 2 and later

To provide file promises for resources that cannot be accessed using a URLFilePromise object, you can implement the IFilePromise interface in a custom class. The IFilePromise interface defines the methods and properties used by the AIR runtime to access the data to be written to a file once the file promise is dropped.

Note: Since the Javascript language does not support the implementation of interfaces, you can only implement your own file promise logic using ActionScript. You can, of course, import a SWF file containing ActionScript classes into a HTML page using a `<script>` tag and access those classes in Javascript code.

An `IFilePromise` implementation passes another object to the AIR runtime that provides the data for the file promise. This object must implement the `IDataInput` interface, which the AIR runtime uses to read the data. For example, the `URLFilePromise` class, which implements `IFilePromise`, uses a `URLStream` object as the data provider.

AIR can read the data synchronously or asynchronously. The `IFilePromise` implementation reports which mode of access is supported by returning the appropriate value in the `isASync` property. If asynchronous data access is provided, the data provider object must implement the `IEventDispatcher` interface and dispatch the necessary events, such as `open`, `progress` and `complete`.

You can use a custom class, or one of the following built-in classes, as a data provider for a file promise:

- `ByteArray` (synchronous)
- `FileStream` (synchronous or asynchronous)
- `Socket` (asynchronous)
- `URLStream` (asynchronous)

To implement the `IFilePromise` interface, you must provide code for the following functions and properties:

- `open():IDataInput` — Returns the data provider object from which the data for the promised file is read. The object must implement the `IDataInput` interface. If the data is provided asynchronously, the object must also implement the `IEventDispatcher` interface and dispatch the necessary events (see [“Using an asynchronous data provider in a file promise”](#) on page 191).
- `getRelativePath():String` — Provides the path, including file name, for the created file. The path is resolved relative to the drop location chosen by the user in the drag-and-drop operation. To make sure that the path uses the proper separator character for the host operating system, use the `File.separator` constant when specifying paths containing directories. You can add a setter function or use a constructor parameter to allow the path to be set at runtime.
- `getIsAsync():Boolean` — Informs the AIR runtime whether the data provider object provides its data asynchronously or synchronously.
- `close():void` — Called by the runtime when the data is fully read (or an error prevents further reading). You can use this function to cleanup resources.
- `reportError(e:ErrorEvent):void` — Called by the runtime when an error reading the data occurs.

All of the `IFilePromise` methods are called by the runtime during a drag-and-drop operation involving the file promise. Typically, your application logic should not call any of these methods directly.

Using a synchronous data provider in a file promise

Adobe AIR 2 and later

The simplest way to implement the `IFilePromise` interface is to use a synchronous data provider object, such as a `ByteArray` or a synchronous `FileStream`. In the following example, a `ByteArray` object is created, filled with data, and returned when the `open()` method is called.

```
package
{
    import flash.desktop.IFilePromise;
    import flash.events.ErrorEvent;
    import flash.utils.ByteArray;
    import flash.utils.IDataInput;

    public class SynchronousFilePromise implements IFilePromise
    {
        private const fileSize:int = 5000; //size of file data
        private var filePath:String = "SynchronousFile.txt";

        public function get relativePath():String
        {
            return filePath;
        }

        public function get isAsync():Boolean
        {
            return false;
        }

        public function open():IDataInput
        {
            var fileContents:ByteArray = new ByteArray();

            //Create some arbitrary data for the file
            for( var i:int = 0; i < fileSize; i++ )
            {
                fileContents.writeUTFBytes( 'S' );
            }

            //Important: the ByteArray is read from the current position
            fileContents.position = 0;
            return fileContents;
        }

        public function close():void
        {
            //Nothing needs to be closed in this case.
        }

        public function reportError(e:ErrorEvent):void
        {
            trace("Something went wrong: " + e.errorID + " - " + e.type + ", " + e.text );
        }
    }
}
```

In practice, synchronous file promises have limited utility. If the amount of data is small, you could just as easily create a file in a temporary directory and add a normal file list array to the drag-and-drop clipboard. On the other hand, if the amount of data is large or generating the data is computationally expensive, a long synchronous process is necessary. Long synchronous processes can block UI updates for a noticeable amount of time and make your application seem unresponsive. To avoid this problem, you can create an asynchronous data provider driven by a timer.

Using an asynchronous data provider in a file promise

Adobe AIR 2 and later

When you use an asynchronous data provider object, the `IFilePromise` `isAsync` property must be `true` and the object returned by the `open()` method must implement the `IEventDispatcher` interface. The runtime listens for several alternative events so that different built-in objects can be used as a data provider. For example, `progress` events are dispatched by `FileStream` and `URLStream` objects, whereas `socketData` events are dispatched by `Socket` objects. The runtime listens for the appropriate events from all of these objects.

The following events drive the process of reading the data from the data provider object:

- `Event.OPEN` — Informs the runtime that the data source is ready.
- `ProgressEvent.PROGRESS` — Informs the runtime that data is available. The runtime will read the amount of available data from the data provider object.
- `ProgressEvent.SOCKET_DATA` — Informs the runtime that data is available. The `socketData` event is dispatched by socket-based objects. For other object types, you should dispatch a `progress` event. (The runtime listens for both events to detect when data can be read.)
- `Event.COMPLETE` — Informs the runtime that the data has all been read.
- `Event.CLOSE` — Informs the runtime that the data has all been read. (The runtime listens for both `close` and `complete` for this purpose.)
- `IOErrorEvent.IOERROR` — Informs the runtime that an error reading the data has occurred. The runtime aborts file creation and calls the `IFilePromise` `close()` method.
- `SecurityErrorEvent.SECURITY_ERROR` — Informs the runtime that a security error has occurred. The runtime aborts file creation and calls the `IFilePromise` `close()` method.
- `HTTPStatusEvent.HTTP_STATUS` — Used, along with `httpResponseStatus`, by the runtime to make sure that the data available represents the desired content, rather than an error message (such as a 404 page). Objects based on the HTTP protocol should dispatch this event.
- `HTTPStatusEvent.HTTP_RESPONSE_STATUS` — Used, along with `httpStatus`, by the runtime to make sure that the data available represents the desired content. Objects based on the HTTP protocol should dispatch this event.

The data provider should dispatch these events in the following sequence:

- 1 `open` event
- 2 `progress` or `socketData` events
- 3 `complete` or `close` event

Note: *The built-in objects, `FileStream`, `Socket`, and `URLStream`, dispatch the appropriate events automatically.*

The following example creates a file promise using a custom, asynchronous data provider. The data provider class extends `ByteArray` (for the `IDataInput` support) and implements the `IEventDispatcher` interface. At each timer event, the object generates a chunk of data and dispatches a `progress` event to inform the runtime that the data is available. When enough data has been produced, the object dispatches a `complete` event.

```
package
{
import flash.events.Event;
import flash.events.EventDispatcher;
import flash.events.IEventDispatcher;
import flash.events.ProgressEvent;
import flash.events.TimerEvent;
import flash.utils.ByteArray;
import flash.utils.Timer;

[Event(name="open", type="flash.events.Event.OPEN")]
[Event(name="complete", type="flash.events.Event.COMPLETE")]
[Event(name="progress", type="flash.events.ProgressEvent")]
[Event(name="ioError", type="flash.events.IOErrorEvent")]
[Event(name="securityError", type="flash.events.SecurityErrorEvent")]
public class AsyncDataProvider extends ByteArray implements IEventDispatcher
{
    private var dispatcher:EventDispatcher = new EventDispatcher();
    public var fileSize:int = 0; //The number of characters in the file
    private const chunkSize:int = 1000; //Amount of data written per event
    private var dispatchDataTimer:Timer = new Timer( 100 );
    private var opened:Boolean = false;

    public function AsyncDataProvider()
    {
        super();
        dispatchDataTimer.addEventListener( TimerEvent.TIMER, generateData );
    }

    public function begin():void{
        dispatchDataTimer.start();
    }

    public function end():void
    {
        dispatchDataTimer.stop();
    }
    private function generateData( event:Event ):void
    {
        if( !opened )
        {
            var open:Event = new Event( Event.OPEN );
            dispatchEvent( open );
            opened = true;
        }
        else if( position + chunkSize < fileSize )
        {
            for( var i:int = 0; i <= chunkSize; i++ )
            {
                writeUTFBytes( 'A' );
            }
            //Set position back to the start of the new data
            this.position -= chunkSize;
            var progress:ProgressEvent =
                new ProgressEvent( ProgressEvent.PROGRESS, false, false, bytesAvailable,
bytesAvailable + chunkSize);
            dispatchEvent( progress )
        }
    }
}
```

```
    }
    else
    {
        var complete:Event = new Event( Event.COMPLETE );
        dispatchEvent( complete );
    }
}
//IEventDispatcher implementation
public function addEventListener(type:String, listener:Function,
useCapture:Boolean=false, priority:int=0, useWeakReference:Boolean=false):void
{
    dispatcher.addEventListener( type, listener, useCapture, priority, useWeakReference );
}

public function removeEventListener(type:String, listener:Function,
useCapture:Boolean=false):void
{
    dispatcher.removeEventListener( type, listener, useCapture );
}

public function dispatchEvent(event:Event):Boolean
{
    return dispatcher.dispatchEvent( event );
}

public function hasEventListener(type:String):Boolean
{
    return dispatcher.hasEventListener( type );
}

public function willTrigger(type:String):Boolean
{
    return dispatcher.willTrigger( type );
}
}
}
```

Note: Because the `AsyncDataProvider` class in the example extends `ByteArray`, it cannot also extend `EventDispatcher`. To implement the `IEventDispatcher` interface, the class uses an internal `EventDispatcher` object and forwards the `IEventDispatcher` method calls to that internal object. You could also extend `EventDispatcher` and implement `IDataInput` (or implement both interfaces).

The asynchronous `IFilePromise` implementation is almost identical to the synchronous implementation. The main differences are that `isAsync` returns `true` and that the `open()` method returns an asynchronous data object:

```
package
{
    import flash.desktop.IFilePromise;
    import flash.events.ErrorEvent;
    import flash.events.EventDispatcher;
    import flash.utils.IDataInput;

    public class AsynchronousFilePromise extends EventDispatcher implements IFilePromise
    {
        private var fileGenerator:AsyncDataProvider;
        private const fileSize:int = 5000; //size of file data
        private var filePath:String = "AsynchronousFile.txt";

        public function get relativePath():String
        {
            return filePath;
        }

        public function get isAsync():Boolean
        {
            return true;
        }

        public function open():IDataInput
        {
            fileGenerator = new AsyncDataProvider();
            fileGenerator.fileSize = fileSize;
            fileGenerator.begin();
            return fileGenerator;
        }

        public function close():void
        {
            fileGenerator.end();
        }

        public function reportError(e:ErrorEvent):void
        {
            trace("Something went wrong: " + e.errorID + " - " + e.type + ", " + e.text );
        }
    }
}
```


Chapter 13: Copy and paste

Flash Player 10 and later, Adobe AIR 1.0 and later

Use the classes in the clipboard API to copy information to and from the system clipboard. The data formats that can be transferred into or out of an application running in Adobe® Flash® Player or Adobe® AIR® include:

- Text
- HTML-formatted text
- Rich Text Format data
- Serialized objects
- Object references (valid only within the originating application)
- Bitmaps (AIR only)
- Files (AIR only)
- URL strings (AIR only)

Basics of copy-and-paste

Flash Player 10 and later, Adobe AIR 1.0 and later

The copy-and-paste API contains the following classes.

Package	Classes
flash.desktop	<ul style="list-style-type: none"> • Clipboard • ClipboardFormats • ClipboardTransferMode

The static `Clipboard.generalClipboard` property represents the operating system clipboard. The `Clipboard` class provides methods for reading and writing data to clipboard objects.

The `HTMLLoader` class (in AIR) and the `TextField` class implement default behavior for the normal copy and paste keyboard shortcuts. To implement copy and paste shortcut behavior for custom components, you can listen for these keystrokes directly. You can also use native menu commands along with key equivalents to respond to the keystrokes indirectly.

Different representations of the same information can be made available in a single `Clipboard` object to increase the ability of other applications to understand and use the data. For example, an image might be included as image data, a serialized `Bitmap` object, and as a file. Rendering of the data in a format can be deferred so that the format is not actually created until the data in that format is read.

Reading from and writing to the system clipboard

Flash Player 10 and later, Adobe AIR 1.0 and later

To read the operating system clipboard, call the `getData()` method of the `Clipboard.generalClipboard` object, passing in the name of the format to read:

```
import flash.desktop.Clipboard;
import flash.desktop.ClipboardFormats;

if (Clipboard.generalClipboard.hasFormat (ClipboardFormats.TEXT_FORMAT)) {
    var text:String = Clipboard.generalClipboard.getData (ClipboardFormats.TEXT_FORMAT);
}
```

Note: Content running in Flash Player or in a non-application sandbox in AIR can call the `getData()` method only in an event handler for a `paste` event. In other words, only code running in the AIR application sandbox can call the `getData()` method outside of a `paste` event handler.

To write to the clipboard, add the data to the `Clipboard.generalClipboard` object in one or more formats. Any existing data in the same format is overwritten automatically. Nevertheless, it is a good practice to also clear the system clipboard before writing new data to it to make sure that unrelated data in any other formats is also deleted.

```
import flash.desktop.Clipboard;
import flash.desktop.ClipboardFormats;

var textToCopy:String = "Copy to clipboard.";
Clipboard.generalClipboard.clear();
Clipboard.generalClipboard.setData (ClipboardFormats.TEXT_FORMAT, textToCopy, false);
```

Note: Content running in Flash Player or in a non-application sandbox in AIR can call the `setData()` method only in an event handler for a user event, such as a keyboard or mouse event, or a `copy` or `cut` event. In other words, only code running in the AIR application sandbox can call the `setData()` method outside of a user event handler.

HTML copy and paste in AIR

Adobe AIR 1.0 and later

The HTML environment in Adobe AIR provides its own set of events and default behavior for copy and paste. Only code running in the application sandbox can access the system clipboard directly through the `AIR Clipboard.generalClipboard` object. JavaScript code in a non-application sandbox can access the clipboard through the event object dispatched in response to one of the `copy` or `paste` events dispatched by an element in an HTML document.

Copy and paste events include: `copy`, `cut`, and `paste`. The object dispatched for these events provides access to the clipboard through the `clipboardData` property.

Default behavior

Adobe AIR 1.0 and later

By default, AIR copies selected items in response to the copy command, which can be generated either by a keyboard shortcut or a context menu. Within editable regions, AIR cuts text in response to the cut command or pastes text to the cursor or selection in response to the paste command.

To prevent the default behavior, your event handler can call the `preventDefault()` method of the dispatched event object.

Using the `clipboardData` property of the event object

Adobe AIR 1.0 and later

The `clipboardData` property of the event object dispatched as a result of one of the copy or paste events allows you to read and write clipboard data.

To write to the clipboard when handling a copy or cut event, use the `setData()` method of the `clipboardData` object, passing in the data to copy and the MIME type:

```
function customCopy(event) {
    event.clipboardData.setData("text/plain", "A copied string.");
}
```

To access the data that is being pasted, you can use the `getData()` method of the `clipboardData` object, passing in the MIME type of the data format. The available formats are reported by the `types` property.

```
function customPaste(event) {
    var pastedData = event.clipboardData("text/plain");
}
```

The `getData()` method and the `types` property can only be accessed in the event object dispatched by the `paste` event.

The following example illustrates how to override the default copy and paste behavior in an HTML page. The `copy` event handler italicizes the copied text and copies it to the clipboard as HTML text. The `cut` event handler copies the selected data to the clipboard and removes it from the document. The `paste` handler inserts the clipboard contents as HTML and styles the insertion as bold text.

Copy and paste

```

<html>
<head>
  <title>Copy and Paste</title>
  <script language="javascript" type="text/javascript">
    function onCopy(event){
      var selection = window.getSelection();
      event.clipboardData.setData("text/html","<i>" + selection + "</i>");
      event.preventDefault();
    }

    function onCut(event){
      var selection = window.getSelection();
      event.clipboardData.setData("text/html","<i>" + selection + "</i>");
      var range = selection.getRangeAt(0);
      range.extractContents();

      event.preventDefault();
    }

    function onPaste(event){
      var insertion = document.createElement("b");
      insertion.innerHTML = event.clipboardData.getData("text/html");
      var selection = window.getSelection();
      var range = selection.getRangeAt(0);
      range.insertNode(insertion);
      event.preventDefault();
    }
  </script>
</head>
<body onCopy="onCopy(event)"
      onPaste="onPaste(event)"
      onCut="onCut(event)">
<p>Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium
doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore
veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam
voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur
magni dolores eos qui ratione voluptatem sequi nesciunt.</p>
</body>
</html>

```

Clipboard data formats

Flash Player 10 and later, Adobe AIR 1.0 and later

Clipboard formats describe the data placed in a Clipboard object. Flash Player or AIR automatically translates the standard data formats between ActionScript data types and system clipboard formats. In addition, application objects can be transferred within and between ActionScript-based applications using application-defined formats.

A Clipboard object can contain representations of the same information in different formats. For example, a Clipboard object representing a Sprite could include a reference format for use within the same application, a serialized format for use by another application running in Flash Player or AIR, a bitmap format for use by an image editor, and a file list format, perhaps with deferred rendering to encode a PNG file, for copying or dragging a representation of the Sprite to the file system.

Standard data formats

Flash Player 10 and later, Adobe AIR 1.0 and later

The constants defining the standard format names are provided in the ClipboardFormats class:

Constant	Description
TEXT_FORMAT	Text-format data is translated to and from the ActionScript String class.
HTML_FORMAT	Text with HTML markup.
RICH_TEXT_FORMAT	Rich-text-format data is translated to and from the ActionScript ByteArray class. The RTF markup is not interpreted or translated in any way.
BITMAP_FORMAT	(AIR only) Bitmap-format data is translated to and from the ActionScript BitmapData class.
FILE_LIST_FORMAT	(AIR only) File-list-format data is translated to and from an array of ActionScript File objects.
URL_FORMAT	(AIR only) URL-format data is translated to and from the ActionScript String class.

When copying and pasting data in response to a `copy`, `cut`, or `paste` event in HTML content hosted in an AIR application, MIME types must be used instead of the ClipboardFormat strings. The valid data MIME types are:

MIME type	Description
Text	"text/plain"
URL	"text/uri-list"
Bitmap	"image/x-vnd.adobe.air.bitmap"
File list	"application/x-vnd.adobe.air.file-list"

Note: Rich text format data is not available from the `clipboardData` property of the event object dispatched as a result of a `paste` event within HTML content.

Custom data formats

Flash Player 10 and later, Adobe AIR 1.0 and later

You can use application-defined custom formats to transfer objects as references or as serialized copies. References are valid only within the same application. Serialized objects can be transferred between applications, but can be used only with objects that remain valid when serialized and deserialized. Objects can usually be serialized if their properties are either simple types or serializable objects.

To add a serialized object to a Clipboard object, set the *serializable* parameter to `true` when calling the `Clipboard.setData()` method. The format name can be one of the standard formats or an arbitrary string defined by your application.

Transfer modes

Flash Player 10 and later, Adobe AIR 1.0 and later

When an object is written to the clipboard using a custom data format, the object data can be read from the clipboard either as a reference or as a serialized copy of the original object. There are four transfer modes that determine whether objects are transferred as references or as serialized copies:

Transfer mode	Description
ClipboardTransferModes.ORIGINAL_ONLY	Only a reference is returned. If no reference is available, a null value is returned.
ClipboardTransferModes.ORIGINAL_PREFERRED	A reference is returned, if available. Otherwise a serialized copy is returned.
ClipboardTransferModes.CLONE_ONLY	Only a serialized copy is returned. If no serialized copy is available, a null value is returned.
ClipboardTransferModes.CLONE_PREFERRED	A serialized copy is returned, if available. Otherwise a reference is returned.

Reading and writing custom data formats

Flash Player 10 and later, Adobe AIR 1.0 and later

When writing an object to the clipboard, you can use any string that does not begin with the reserved prefixes `air:` or `flash:` for the *format* parameter. Use the same string as the format to read the object. The following examples illustrate how to read and write objects to the clipboard:

```
public function createClipboardObject(object:Object):Clipboard{
    var transfer:Clipboard = Clipboard.generalClipboard;
    transfer.setData("object", object, true);
}

function createClipboardObject(object){
    var transfer = new air.Clipboard();
    transfer.setData("object", object, true);
}
```

To extract a serialized object from the clipboard object (after a drop or paste operation), use the same format name and the `CLONE_ONLY` or `CLONE_PREFERRED` transfer modes.

```
var transfer:Object = clipboard.getData("object", ClipboardTransferMode.CLONE_ONLY);
var transfer = clipboard.getData("object", air.ClipboardTransferMode.CLONE_ONLY);
```

A reference is always added to the Clipboard object. To extract the reference from the clipboard object (after a drop or paste operation), instead of the serialized copy, use the `ORIGINAL_ONLY` or `ORIGINAL_PREFERRED` transfer modes:

```
var transferredObject:Object =
    clipboard.getData("object", ClipboardTransferMode.ORIGINAL_ONLY);

var transferredObject =
    clipboard.getData("object", air.ClipboardTransferMode.ORIGINAL_ONLY);
```

References are valid only if the Clipboard object originates from the current application. Use the `ORIGINAL_PREFERRED` transfer mode to access the reference when it is available, and the serialized clone when the reference is not available.

Deferred rendering

Flash Player 10 and later, Adobe AIR 1.0 and later

If creating a data format is computationally expensive, you can use deferred rendering by supplying a function that supplies the data on demand. The function is called only if a receiver of the drop or paste operation requests data in the deferred format.

Copy and paste

The rendering function is added to a Clipboard object using the `setDataHandler()` method. The function must return the data in the appropriate format. For example, if you called `setDataHandler(ClipboardFormat.TEXT_FORMAT, writeText)`, then the `writeText()` function must return a string.

If a data format of the same type is added to a Clipboard object with the `setData()` method, that data takes precedence over the deferred version (the rendering function is never called). The rendering function may or may not be called again if the same clipboard data is accessed a second time.

Note: On Mac OS X, deferred rendering works only with custom data formats. With standard data formats, the rendering function is called immediately.

Pasting text using a deferred rendering function**Flash Player 10 and later, Adobe AIR 1.0 and later**

The following example illustrates how to implement a deferred rendering function.

When the user presses the Copy button, the application clears the system clipboard to ensure that no data is left over from previous clipboard operations. The `setDataHandler()` method then sets the `renderData()` function as the clipboard renderer.

When the user selects the Paste command from the context menu of the destination text field, the application accesses the clipboard and sets the destination text. Since the text data format on the clipboard has been set with a function rather than a string, the clipboard calls the `renderData()` function. The `renderData()` function returns the text in the source text, which is then assigned to the destination text.

Notice that if you edit the source text before pressing the Paste button, the edit will be reflected in the pasted text, even when the edit occurs after the copy button was pressed. This is because the rendering function doesn't copy the source text until the paste button is pressed. (When using deferred rendering in a real application, you might want to store or protect the source data in some way to prevent this problem.)

Flash example

```
package {
    import flash.desktop.Clipboard;
    import flash.desktop.ClipboardFormats;
    import flash.desktop.ClipboardTransferMode;
    import flash.display.Sprite;
    import flash.text.TextField;
    import flash.text.TextFormat;
    import flash.text.TextFieldType;
    import flash.events.MouseEvent;
    import flash.events.Event;
    public class DeferredRenderingExample extends Sprite
    {
        private var sourceTextField:TextField;
        private var destination:TextField;
        private var copyText:TextField;
        public function DeferredRenderingExample():void
        {
            sourceTextField = createTextField(10, 10, 380, 90);
            sourceTextField.text = "Neque porro quisquam est qui dolorem "
                + "ipsum quia dolor sit amet, consectetur, adipisci velit.";

            copyText = createTextField(10, 110, 35, 20);
```

```
copyText.htmlText = "<a href='#'>Copy</a>";
copyText.addEventListener(MouseEvent.CLICK, onCopy);

destination = createTextField(10, 145, 380, 90);
destination.addEventListener(Event.PASTE, onPaste);
}
private function createTextField(x:Number, y:Number, width:Number,
    height:Number):TextField
{
    var newTxt:TextField = new TextField();
    newTxt.x = x;
    newTxt.y = y;
    newTxt.height = height;
    newTxt.width = width;
    newTxt.border = true;
    newTxt.multiline = true;
    newTxt.wordWrap = true;
    newTxt.type = TextFieldType.INPUT;
    addChild(newTxt);
    return newTxt;
}
public function onCopy(event:MouseEvent):void
{
    Clipboard.generalClipboard.clear();
    Clipboard.generalClipboard.setDataHandler(ClipboardFormats.TEXT_FORMAT,
        renderData);
}
public function onPaste(event:Event):void
{
    sourceTextField.text =
        Clipboard.generalClipboard.getData(ClipboardFormats.TEXT_FORMAT).toString;
}
public function renderData():String
{
    trace("Rendering data");
    var sourceStr:String = sourceTextField.text;
    if (sourceTextField.selectionEndIndex >
        sourceTextField.selectionBeginIndex)
    {
        return sourceStr.substring(sourceTextField.selectionBeginIndex,
            sourceTextField.selectionEndIndex);
    }
    else
    {
        return sourceStr;
    }
}
}
```


Flex example

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute" width="326"
height="330" applicationComplete="init()">
  <mx:Script>
    <![CDATA[
      import flash.desktop.Clipboard;
      import flash.desktop.ClipboardFormats;

      public function init():void
      {
        destination.addEventListener("paste", doPaste);
      }

      public function doCopy():void
      {
        Clipboard.generalClipboard.clear();
        Clipboard.generalClipboard.setDataHandler(ClipboardFormats.TEXT_FORMAT, renderData);
      }
      public function doPaste(event:Event):void
      {
        destination.text =
Clipboard.generalClipboard.getData(ClipboardFormats.TEXT_FORMAT).toString();
      }

      public function renderData():String{
        trace("Rendering data");
        return source.text;
      }
    ]]>
  </mx:Script>
  <mx:Label x="10" y="10" text="Source"/>
  <mx:TextArea id="source" x="10" y="36" width="300" height="100">
    <mx:text>Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur,
adipisci velit.</mx:text>
  </mx:TextArea>
  <mx:Label x="10" y="181" text="Destination"/>
  <mx:TextArea id="destination" x="12" y="207" width="300" height="100"/>
  <mx:Button click="doCopy();" x="91" y="156" label="Copy"/>
</mx:Application>
```

Chapter 14: Working with local SQL databases in AIR

Adobe AIR 1.0 and later

Adobe® AIR® includes the capability of creating and working with local SQL databases. The runtime includes a SQL database engine with support for many standard SQL features, using the open source SQLite database system. A local SQL database can be used for storing local, persistent data. For example, it can be used for application data, application user settings, documents, or any other type of data that you want your application to save locally.

About local SQL databases

Adobe AIR 1.0 and later

For a quick explanation and code examples of using SQL databases, see the following quick start articles on the Adobe Developer Connection:

- [Working asynchronously with a local SQL database](#)
- [Working synchronously with a local SQL database](#)
- [Using an encrypted database](#)

Adobe AIR includes a SQL-based relational database engine that runs within the runtime, with data stored locally in database files on the computer on which the AIR application runs (for example, on the computer's hard drive). Because the database runs and data files are stored locally, a database can be used by an AIR application regardless of whether a network connection is available. Thus, the runtime's local SQL database engine provides a convenient mechanism for storing persistent, local application data, particularly if you have experience with SQL and relational databases.

Uses for local SQL databases

Adobe AIR 1.0 and later

The AIR local SQL database functionality can be used for any purpose for which you might want to store application data on a user's local computer. Adobe AIR includes several mechanisms for storing data locally, each of which has different advantages. The following are some possible uses for a local SQL database in your AIR application:

- For a data-oriented application (for example an address book), a database can be used to store the main application data.
- For a document-oriented application, where users create documents to save and possibly share, each document could be saved as a database file, in a user-designated location. (Note, however, that unless the database is encrypted any AIR application would be able to open the database file. Encryption is recommended for potentially sensitive documents.)
- For a network-aware application, a database can be used to store a local cache of application data, or to store data temporarily when a network connection isn't available. You could create a mechanism for synchronizing the local database with the network data store.

- For any application, a database can be used to store individual users' application settings, such as user options or application information like window size and position.

More Help topics

[Christophe Coenraets: Employee Directory on AIR for Android](#)

[Raymond Camden: jQuery and AIR - Moving from web page to application](#)

About AIR databases and database files

Adobe AIR 1.0 and later

An individual Adobe AIR local SQL database is stored as a single file in the computer's file system. The runtime includes the SQL database engine that manages creation and structuring of database files and manipulation and retrieval of data from a database file. The runtime does not specify how or where database data is stored on the file system; rather, each database is stored completely within a single file. You specify the location in the file system where the database file is stored. A single AIR application can access one or many separate databases (that is, separate database files). Because the runtime stores each database as a single file on the file system, you can locate your database as needed by the design of your application and file access constraints of the operating system. Each user can have a separate database file for their specific data, or a database file can be accessed by all application users on a single computer for shared data. Because the data is local to a single computer, data is not automatically shared among users on different computers. The local SQL database engine doesn't provide any capability to execute SQL statements against a remote or server-based database.

About relational databases

Adobe AIR 1.0 and later

A relational database is a mechanism for storing (and retrieving) data on a computer. Data is organized into tables: rows represent records or items, and columns (sometimes called "fields") divide each record into individual values. For example, an address book application could contain a "friends" table. Each row in the table would represent a single friend stored in the database. The table's columns would represent data such as first name, last name, birth date, and so forth. For each friend row in the table, the database stores a separate value for each column.

Relational databases are designed to store complex data, where one item is associated with or related to items of another type. In a relational database, any data that has a one-to-many relationship—where a single record can be related to multiple records of a different type—should be divided among different tables. For example, suppose you want your address book application to store multiple phone numbers for each friend; this is a one-to-many relationship. The "friends" table would contain all the personal information for each friend. A separate "phone numbers" table would contain all the phone numbers for all the friends.

In addition to storing the data about friends and phone numbers, each table would need a piece of data to keep track of the relationship between the two tables—to match individual friend records with their phone numbers. This data is known as a primary key—a unique identifier that distinguishes each row in a table from other rows in that table. The primary key can be a "natural key," meaning it's one of the items of data that naturally distinguishes each record in a table. In the "friends" table, if you knew that none of your friends share a birth date, you could use the birth date column as the primary key (a natural key) of the "friends" table. If there isn't a natural key, you would create a separate primary key column such as a "friend id" —an artificial value that the application uses to distinguish between rows.

Using a primary key, you can set up relationships between multiple tables. For example, suppose the “friends” table has a column “friend id” that contains a unique number for each row (each friend). The related “phone numbers” table can be structured with two columns: one with the “friend id” of the friend to whom the phone number belongs, and one with the actual phone number. That way, no matter how many phone numbers a single friend has, they can all be stored in the “phone numbers” table and can be linked to the related friend using the “friend id” primary key. When a primary key from one table is used in a related table to specify the connection between the records, the value in the related table is known as a foreign key. Unlike many databases, the AIR local database engine does not allow you to create foreign key constraints, which are constraints that automatically check that an inserted or updated foreign key value has a corresponding row in the primary key table. Nevertheless, foreign key relationships are an important part of the structure of a relational database, and foreign keys should be used when creating relationships between tables in your database.

About SQL

Adobe AIR 1.0 and later

Structured Query Language (SQL) is used with relational databases to manipulate and retrieve data. SQL is a descriptive language rather than a procedural language. Instead of giving the computer instructions on how it should retrieve data, a SQL statement describes the set of data you want. The database engine determines how to retrieve that data.

The SQL language has been standardized by the American National Standards Institute (ANSI). The Adobe AIR local SQL database supports most of the SQL-92 standard.

For specific descriptions of the SQL language supported in Adobe AIR, see “[SQL support in local databases](#)” on page 341.

About SQL database classes

Adobe AIR 1.0 and later

To work with local SQL databases in JavaScript, you use instances of the following classes. (Note that you need to load the file AIRAliases.js in your HTML document in order to use the air.* aliases for these classes):

Class	Description
air.SQLConnection	Provides the means to create and open databases (database files), as well as methods for performing database-level operations and for controlling database transactions.
air.SQLStatement	Represents a single SQL statement (a single query or command) that is executed on a database, including defining the statement text and setting parameter values.
air.ResultSet	Provides a way to get information about or results from executing a statement, such as the result rows from a SELECT statement, the number of rows affected by an UPDATE or DELETE statement, and so forth.

To obtain schema information describing the structure of a database, you use these classes:

Class	Description
air.SQLSchemaResult	Serves as a container for database schema results generated by calling the <code>SQLConnection.loadSchema()</code> method.
air.SQLTableSchema	Provides information describing a single table in a database.

Class	Description
air.SQLViewSchema	Provides information describing a single view in a database.
air.SQLIndexSchema	Provides information describing a single column of a table or view in a database.
air.SQLTriggerSchema	Provides information describing a single trigger in a database.

The following classes provide constants that are used with the `SQLConnection` class:

Class	Description
air.SQLMode	Defines a set of constants representing the possible values for the <code>openMode</code> parameter of the <code>SQLConnection.open()</code> and <code>SQLConnection.openAsync()</code> methods.
air.SQLColumnNameStyle	Defines a set of constants representing the possible values for the <code>SQLConnection.columnNameStyle</code> property.
air.SQLTransactionLockType	Defines a set of constants representing the possible values for the option parameter of the <code>SQLConnection.begin()</code> method.
air.SQLCollationType	Defines a set of constants representing the possible values for the <code>SQLColumnSchema.defaultCollationType</code> property and the <code>defaultCollationType</code> parameter of the <code>SQLColumnSchema()</code> constructor.

In addition, the following classes represent the events (and supporting constants) that you use:

Class	Description
air.SQLEvent	Defines the events that a <code>SQLConnection</code> or <code>SQLStatement</code> instance dispatches when any of its operations execute successfully. Each operation has an associated event type constant defined in the <code>SQLEvent</code> class.
air.SQLErrorEvent	Defines the event that a <code>SQLConnection</code> or <code>SQLStatement</code> instance dispatches when any of its operations results in an error.
air.SQLUpdateEvent	Defines the event that a <code>SQLConnection</code> instances dispatches when table data in one of its connected databases changes as a result of an <code>INSERT</code> , <code>UPDATE</code> , or <code>DELETE</code> SQL statement being executed.

Finally, the following classes provide information about database operation errors:

Class	Description
air.SQLError	Provides information about a database operation error, including the operation that was being attempted and the cause of the failure.
air.SQLErrorOperation	Defines a set of constants representing the possible values for the <code>SQLError</code> class's <code>operation</code> property, which indicates the database operation that resulted in an error.

About synchronous and asynchronous execution modes

Adobe AIR 1.0 and later

When you're writing code to work with a local SQL database, you specify that database operations execution in one of two execution modes: asynchronous or synchronous execution mode. In general, the code examples show how to perform each operation in both ways, so that you can use the example that's most appropriate for your needs.

In asynchronous execution mode, you give the runtime an instruction and the runtime dispatches an event when your requested operation completes or fails. First you tell the database engine to perform an operation. The database engine does its work in the background while the application continues running. Finally, when the operation is completed (or when it fails) the database engine dispatches an event. Your code, triggered by the event, carries out subsequent operations. This approach has a significant benefit: the runtime performs the database operations in the background while the main application code continues executing. If the database operation takes a notable amount of time, the application continues to run. Most importantly, the user can continue to interact with it without the screen freezing. Nevertheless, asynchronous operation code can be more complex to write than other code. This complexity is usually in cases where multiple dependent operations must be divided up among various event listener methods.

Conceptually, it is simpler to code operations as a single sequence of steps—a set of synchronous operations—rather than a set of operations split into several event listener methods. In addition to asynchronous database operations, Adobe AIR also allows you to execute database operations synchronously. In synchronous execution mode, operations don't run in the background. Instead they run in the same execution sequence as all other application code. You tell the database engine to perform an operation. The code then pauses at that point while the database engine does its work. When the operation completes, execution continues with the next line of your code.

Whether operations execute asynchronously or synchronously is set at the `SQLConnection` level. Using a single database connection, you can't execute some operations or statements synchronously and others asynchronously. You specify whether a `SQLConnection` operates in synchronous or asynchronous execution mode by calling a `SQLConnection` method to open the database. If you call `SQLConnection.open()` the connection operates in synchronous execution mode, and if you call `SQLConnection.openAsync()` the connection operates in asynchronous execution mode. Once a `SQLConnection` instance is connected to a database using `open()` or `openAsync()`, it is fixed to synchronous or asynchronous execution mode unless you close and reopen the connection to the database.

Each execution mode has benefits. While most aspects of each mode are similar, there are some differences you'll want to keep in mind when working in each mode. For more information on these topics, and suggestions for working in each mode, see "[Using synchronous and asynchronous database operations](#)" on page 232.

Creating and modifying a database

Adobe AIR 1.0 and later

Before your application can add or retrieve data, there must be a database with tables defined in it that your application can access. Described here are the tasks of creating a database and creating the data structure within a database. While these tasks are less frequently used than data insertion and retrieval, they are necessary for most applications.

More Help topics

[Mind the Flex: Updating an existing AIR database](#)

Creating a database

Adobe AIR 1.0 and later

To create a database file, you first create a `SQLConnection` instance. You call its `open()` method to open it in synchronous execution mode, or its `openAsync()` method to open it in asynchronous execution mode. The `open()` and `openAsync()` methods are used to open a connection to a database. If you pass a `File` instance that refers to a non-existent file location for the `reference` parameter (the first parameter), the `open()` or `openAsync()` method creates a database file at that file location and open a connection to the newly created database.

Whether you call the `open()` method or the `openAsync()` method to create a database, the database file's name can be any valid filename, with any filename extension. If you call the `open()` or `openAsync()` method with `null` for the `reference` parameter, a new in-memory database is created rather than a database file on disk.

The following code listing shows the process of creating a database file (a new database) using asynchronous execution mode. In this case, the database file is saved in the ["Pointing to the application storage directory"](#) on page 149, with the filename "DBSample.db":

```
// Include AIRAliases.js to use air.* shortcuts

var conn = new air.SQLConnection();

conn.addEventListener(air.SQLEvent.OPEN, openHandler);
conn.addEventListener(air.SQLErrorEvent.ERROR, errorHandler);

// The database file is in the application storage directory
var folder = air.File.applicationStorageDirectory;
var dbFile = folder.resolvePath("DBSample.db");

conn.openAsync(dbFile);

function openHandler(event)
{
    air.trace("the database was created successfully");
}

function errorHandler(event)
{
    air.trace("Error message:", event.error.message);
    air.trace("Details:", event.error.details);
}
```

Note: Although the `File` class lets you point to a specific native file path, doing so can lead to applications that will not work across platforms. For example, the path `C:\Documents and Settings\joe\test.db` only works on Windows. For these reasons, it is best to use the static properties of the `File` class such as `File.applicationStorageDirectory`, as well as the `resolvePath()` method (as shown in the previous example). For more information, see ["Paths of File objects"](#) on page 147.

To execute operations synchronously, when you open a database connection with the `SQLConnection` instance, call the `open()` method. The following example shows how to create and open a `SQLConnection` instance that executes its operations synchronously:

```
// Include AIRAliases.js to use air.* shortcuts

var conn = new air.SQLConnection();

// The database file is in the application storage directory
var folder = air.File.applicationStorageDirectory;
var dbFile = folder.resolvePath("DBSample.db");

try
{
    conn.open(dbFile);
    air.trace("the database was created successfully");
}
catch (error)
{
    air.trace("Error message:", error.message);
    air.trace("Details:", error.details);
}
```

Creating database tables

Adobe AIR 1.0 and later

Creating a table in a database involves executing a SQL statement on that database, using the same process that you use to execute a SQL statement such as `SELECT`, `INSERT`, and so forth. To create a table, you use a `CREATE TABLE` statement, which includes definitions of columns and constraints for the new table. For more information about executing SQL statements, see [“Working with SQL statements”](#) on page 214.

The following example demonstrates creating a table named “employees” in an existing database file, using asynchronous execution mode. Note that this code assumes there is a `SQLConnection` instance named `conn` that is already instantiated and is already connected to a database.


```
// Include AIRAliases.js to use air.* shortcuts

// ... create and open the SQLConnection instance named conn ...

var createStmt = new air.SQLStatement();
createStmt.sqlConnection = conn;

var sql =
    "CREATE TABLE IF NOT EXISTS employees (" +
    "    empId INTEGER PRIMARY KEY AUTOINCREMENT, " +
    "    firstName TEXT, " +
    "    lastName TEXT, " +
    "    salary NUMERIC CHECK (salary > 0) " +
    ")";
createStmt.text = sql;

createStmt.addEventListener(air.SQLEvent.RESULT, createResult);
createStmt.addEventListener(air.SQLErrorEvent.ERROR, createError);

createStmt.execute();

function createResult(event)
{
    air.trace("Table created");
}

function createError(event)
{
    air.trace("Error message:", event.error.message);
    air.trace("Details:", event.error.details);
}
```

The following example demonstrates how to create a table named “employees” in an existing database file, using synchronous execution mode. Note that this code assumes there is a `SQLConnection` instance named `conn` that is already instantiated and is already connected to a database.

```
// Include AIRAliases.js to use air.* shortcuts

// ... create and open the SQLConnection instance named conn ...

var createStmt = new air.SQLStatement();
createStmt.sqlConnection = conn;

var sql =
    "CREATE TABLE IF NOT EXISTS employees (" +
    "    empId INTEGER PRIMARY KEY AUTOINCREMENT, " +
    "    firstName TEXT, " +
    "    lastName TEXT, " +
    "    salary NUMERIC CHECK (salary > 0) " +
    ")";
createStmt.text = sql;

try
{
    createStmt.execute();
    air.trace("Table created");
}
catch (error)
{
    air.trace("Error message:", error.message);
    air.trace("Details:", error.details);
}
```

Manipulating SQL database data

Adobe AIR 1.0 and later

There are some common tasks that you perform when you're working with local SQL databases. These tasks include connecting to a database, adding data to tables, and retrieving data from tables in a database. There are also several issues you'll want to keep in mind while performing these tasks, such as working with data types and handling errors.

Note that there are also several database tasks that are things you'll deal with less frequently, but will often need to do before you can perform these more common tasks. For example, before you can connect to a database and retrieve data from a table, you'll need to create the database and create the table structure in the database. Those less-frequent initial setup tasks are discussed in [“Creating and modifying a database”](#) on page 208.

You can choose to perform database operations asynchronously, meaning the database engine runs in the background and notifies you when the operation succeeds or fails by dispatching an event. You can also perform these operations synchronously. In that case the database operations are performed one after another and the entire application (including updates to the screen) waits for the operations to complete before executing other code. For more information on working in asynchronous or synchronous execution mode, see [“Using synchronous and asynchronous database operations”](#) on page 232.

Connecting to a database

Adobe AIR 1.0 and later

Before you can perform any database operations, first open a connection to the database file. A `SQLConnection` instance is used to represent a connection to one or more databases. The first database that is connected using a `SQLConnection` instance is known as the “main” database. This database is connected using the `open()` method (for synchronous execution mode) or the `openAsync()` method (for asynchronous execution mode).

If you open a database using the asynchronous `openAsync()` operation, register for the `SQLConnection` instance's `open` event in order to know when the `openAsync()` operation completes. Register for the `SQLConnection` instance's `error` event to determine if the operation fails.

The following example shows how to open an existing database file for asynchronous execution. The database file is named “DBSample.db” and is located in the user's [“Pointing to the application storage directory”](#) on page 149.

```
// Include AIRAliases.js to use air.* shortcuts

var conn = new air.SQLConnection();

conn.addEventListener(air.SQLEvent.OPEN, openHandler);
conn.addEventListener(air.SQLErrorEvent.ERROR, errorHandler);

// The database file is in the application storage directory
var folder = air.File.applicationStorageDirectory;
var dbFile = folder.resolvePath("DBSample.db");

conn.openAsync(dbFile, air.SQLMode.UPDATE);

function openHandler(event)
{
    air.trace("the database opened successfully");
}

function errorHandler(event)
{
    air.trace("Error message:", event.error.message);
    air.trace("Details:", event.error.details);
}
```

The following example shows how to open an existing database file for synchronous execution. The database file is named “DBSample.db” and is located in the user's [“Pointing to the application storage directory”](#) on page 149.

```
// Include AIRAliases.js to use air.* shortcuts

var conn = new air.SQLConnection();

// The database file is in the application storage directory
var folder = air.File.applicationStorageDirectory;
var dbFile = folder.resolvePath("DBSample.db");

try
{
    conn.open(dbFile, air.SQLMode.UPDATE);
    air.trace("the database opened successfully");
}
catch (error)
{
    air.trace("Error message:", error.message);
    air.trace("Details:", error.details);
}
```

Notice that in the `openAsync()` method call in the asynchronous example, and the `open()` method call in the synchronous example, the second argument is the constant `SQLMode.UPDATE`. Specifying `SQLMode.UPDATE` for the second parameter (`openMode`) causes the runtime to dispatch an error if the specified file doesn't exist. If you pass `SQLMode.CREATE` for the `openMode` parameter (or if you leave the `openMode` parameter off), the runtime attempts to create a database file if the specified file doesn't exist. However, if the file exists it is opened, which is the same as if you use `SQLMode.Update`. You can also specify `SQLMode.READ` for the `openMode` parameter to open an existing database in a read-only mode. In that case data can be retrieved from the database but no data can be added, deleted, or changed.

Working with SQL statements

Adobe AIR 1.0 and later

An individual SQL statement (a query or command) is represented in the runtime as a `SQLStatement` object. Follow these steps to create and execute a SQL statement:

Create a `SQLStatement` instance.

The `SQLStatement` object represents the SQL statement in your application.

```
var selectData = new air.SQLStatement();
```

Specify which database the query runs against.

To do this, set the `SQLStatement` object's `sqlConnection` property to the `SQLConnection` instance that's connected with the desired database.

```
// A SQLConnection named "conn" has been created previously
selectData.sqlConnection = conn;
```

Specify the actual SQL statement.

Create the statement text as a `String` and assign it to the `SQLStatement` instance's `text` property.

```
selectData.text = "SELECT col1, col2 FROM my_table WHERE col1 = :param1";
```

Define functions to handle the result of the execute operation (asynchronous execution mode only).

Use the `addEventListener()` method to register functions as listeners for the `SQLStatement` instance's `result` and `error` events.

```
// using listener methods and addEventListener()

selectData.addEventListener(air.SQLEvent.RESULT, resultHandler);
selectData.addEventListener(air.SQLErrorEvent.ERROR, errorHandler);

function resultHandler(event)
{
    // do something after the statement execution succeeds
}

function errorHandler(event)
{
    // do something after the statement execution fails
}
```

Alternatively, you can specify listener methods using a Responder object. In that case you create the Responder instance and link the listener methods to it.

```
// using a Responder

var selectResponder = new air.Responder(onResult, onError);

function onResult(result)
{
    // do something after the statement execution succeeds
}

function onError(error)
{
    // do something after the statement execution fails
}
```

If the statement text includes parameter definitions, assign values for those parameters.

To assign parameter values, use the SQLStatement instance's `parameters` associative array property.

```
selectData.parameters[":param1"] = 25;
```

Execute the SQL statement.

Call the SQLStatement instance's `execute()` method.

```
// using synchronous execution mode
// or listener methods in asynchronous execution mode
selectData.execute();
```

Additionally, if you're using a Responder instead of event listeners in asynchronous execution mode, pass the Responder instance to the `execute()` method.

```
// using a Responder in asynchronous execution mode
selectData.execute(-1, selectResponder);
```

For specific examples that demonstrate these steps, see the following topics:

[“Retrieving data from a database”](#) on page 218



[“Inserting data”](#) on page 225



[“Changing or deleting data”](#) on page 228

Using parameters in statements

Adobe AIR 1.0 and later

Using SQL statement parameters allows you to create a reusable SQL statement. When you use statement parameters, values within the statement can change (such as values being added in an `INSERT` statement) but the basic statement text remains unchanged. Consequently, using parameters provides performance benefits and makes it easier to code an application.

Understanding statement parameters

Adobe AIR 1.0 and later

Frequently an application uses a single SQL statement multiple times in an application, with slight variation. For example, consider an inventory-tracking application where a user can add new inventory items to the database. The application code that adds an inventory item to the database executes a SQL `INSERT` statement that actually adds the data to the database. However, each time the statement is executed there is a slight variation. Specifically, the actual values that are inserted in the table are different because they are specific to the inventory item being added.

In cases where you have a SQL statement that's used multiple times with different values in the statement, the best approach is to use a SQL statement that includes parameters rather than literal values in the SQL text. A parameter is a placeholder in the statement text that is replaced with an actual value each time the statement is executed. To use parameters in a SQL statement, you create the `SQLStatement` instance as usual. For the actual SQL statement assigned to the `text` property, use parameter placeholders rather than literal values. You then define the value for each parameter by setting the value of an element in the `SQLStatement` instance's `parameters` property. The `parameters` property is an associative array, so you set a particular value using the following syntax:

```
statement.parameters[parameter_identifier] = value;
```

The `parameter_identifier` is a string if you're using a named parameter, or an integer index if you're using an unnamed parameter.

Using named parameters

Adobe AIR 1.0 and later

A parameter can be a named parameter. A named parameter has a specific name that the database uses to match the parameter value to its placeholder location in the statement text. A parameter name consists of the ":" or "@" character followed by a name, as in the following examples:

```
:itemName  
@firstName
```

The following code listing demonstrates the use of named parameters:

```
var sql =
    "INSERT INTO inventoryItems (name, productCode)" +
    "VALUES (:name, :productCode)";

var addItemStmt = new air.SQLStatement();
addItemStmt.sqlConnection = conn;
addItemStmt.text = sql;

// set parameter values
addItemStmt.parameters[":name"] = "Item name";
addItemStmt.parameters[":productCode"] = "12345";

addItemStmt.execute();
```

Using unnamed parameters

Adobe AIR 1.0 and later

As an alternative to using named parameters, you can also use unnamed parameters. To use an unnamed parameter you denote a parameter in a SQL statement using a “?” character. Each parameter is assigned a numeric index, according to the order of the parameters in the statement, starting with index 0 for the first parameter. The following example demonstrates a version of the previous example, using unnamed parameters:

```
var sql =
    "INSERT INTO inventoryItems (name, productCode)" +
    "VALUES (?, ?)";

var addItemStmt = new air.SQLStatement();
addItemStmt.sqlConnection = conn;
addItemStmt.text = sql;

// set parameter values
addItemStmt.parameters[0] = "Item name";
addItemStmt.parameters[1] = "12345";

addItemStmt.execute();
```

Benefits of using parameters

Adobe AIR 1.0 and later

Using parameters in a SQL statement provides several benefits:

Better performance A `SQLStatement` instance that uses parameters can execute more efficiently compared to one that dynamically creates the SQL text each time it executes. The performance improvement is because the statement is prepared a single time and can then be executed multiple times using different parameter values, without needing to recompile the SQL statement.

Explicit data typing Parameters are used to allow for typed substitution of values that are unknown at the time the SQL statement is constructed. The use of parameters is the only way to guarantee the storage class for a value passed in to the database. When parameters are not used, the runtime attempts to convert all values from their text representation to a storage class based on the associated column's type affinity.

For more information on storage classes and column affinity, see “[Data type support](#)” on page 362.

Greater security The use of parameters helps prevent a malicious technique known as a SQL injection attack. In a SQL injection attack, a user enters SQL code in a user-accessible location (for example, a data entry field). If application

code constructs a SQL statement by directly concatenating user input into the SQL text, the user-entered SQL code is executed against the database. The following listing shows an example of concatenating user input into SQL text. **Do not use this technique:**

```
// assume the variables "username" and "password"
// contain user-entered data

var sql =
    "SELECT userId " +
    "FROM users " +
    "WHERE username = '" + username + "' " +
    "    AND password = '" + password + "'";

var statement = new air.SQLStatement();
statement.text = sql;
```

Using statement parameters instead of concatenating user-entered values into a statement's text prevents a SQL injection attack. SQL injection can't happen because the parameter values are treated explicitly as substituted values, rather than becoming part of the literal statement text. The following is the recommended alternative to the previous listing:

```
// assume the variables "username" and "password"
// contain user-entered data

var sql =
    "SELECT userId " +
    "FROM users " +
    "WHERE username = :username " +
    "    AND password = :password";

var statement = new air.SQLStatement();
statement.text = sql;

// set parameter values
statement.parameters[":username"] = username;
statement.parameters[":password"] = password;
```

Retrieving data from a database

Adobe AIR 1.0 and later

Retrieving data from a database involves two steps. First, you execute a SQL `SELECT` statement, describing the set of data you want from the database. Next, you access the retrieved data and display or manipulate it as needed by your application.

Executing a `SELECT` statement

Adobe AIR 1.0 and later

To retrieve existing data from a database, you use a `SQLStatement` instance. Assign the appropriate SQL `SELECT` statement to the instance's `text` property, then call its `execute()` method.

For details on the syntax of the `SELECT` statement, see [“SQL support in local databases”](#) on page 341.

The following example demonstrates executing a `SELECT` statement to retrieve data from a table named “products,” using asynchronous execution mode:


```
// Include AIRAliases.js to use air.* shortcuts

var selectStmt = new air.SQLStatement();

// A SQLConnection named "conn" has been created previously
selectStmt.sqlConnection = conn;

selectStmt.text = "SELECT itemId, itemName, price FROM products";

selectStmt.addEventListener(air.SQLEvent.RESULT, resultHandler);
selectStmt.addEventListener(air.SQLErrorEvent.ERROR, errorHandler);

selectStmt.execute();

function resultHandler(event)
{
    var result = selectStmt.getResult();

    var numResults = result.data.length;
    for (i = 0; i < numResults; i++)
    {
        var row = result.data[i];
        var output = "itemId: " + row.itemId;
        output += "; itemName: " + row.itemName;
        output += "; price: " + row.price;
        air.trace(output);
    }
}

function errorHandler(event)
{
    // Information about the error is available in the
    // event.error property, which is an instance of
    // the SQLError class.
}
```

The following example demonstrates executing a `SELECT` statement to retrieve data from a table named “products,” using synchronous execution mode:

```
// Include AIRAliases.js to use air.* shortcuts

var selectStmt = new air.SQLStatement();

// A SQLConnection named "conn" has been created previously
selectStmt.sqlConnection = conn;

selectStmt.text = "SELECT itemId, itemName, price FROM products";

try
{
    selectStmt.execute();

    var result = selectStmt.getResult();

    var numResults = result.data.length;
    for (i = 0; i < numResults; i++)
    {
        var row = result.data[i];
        var output = "itemId: " + row.itemId;
        output += "; itemName: " + row.itemName;
        output += "; price: " + row.price;
        air.trace(output);
    }
}
catch (error)
{
    // Information about the error is available in the
    // error variable, which is an instance of
    // the SQLException class.
}
```

In asynchronous execution mode, when the statement finishes executing, the `SQLStatement` instance dispatches a `result` event (`SQLEvent.RESULT`) indicating that the statement was run successfully. Alternatively, if a `Responder` object is passed as an argument to the `execute()` method, the `Responder` object's result handler function is called. In synchronous execution mode, execution pauses until the `execute()` operation completes, then continues on the next line of code.

Accessing SELECT statement result data

Adobe AIR 1.0 and later

Once the `SELECT` statement has finished executing, the next step is to access the data that was retrieved. You retrieve the result data from executing a `SELECT` statement by calling the `SQLStatement` object's `getResult()` method:

```
var result = selectStatement.getResult();
```

The `getResult()` method returns a `SQLResult` object. The `SQLResult` object's `data` property is an `Array` containing the results of the `SELECT` statement:

```
var numResults = result.data.length;
for (var i = 0; i < numResults; i++)
{
    // row is an Object representing one row of result data
    var row = result.data[i];
}
```

Each row of data in the `SELECT` result set becomes an Object instance contained in the `data` Array. That object has properties whose names match the result set's column names. The properties contain the values from the result set's columns. For example, suppose a `SELECT` statement specifies a result set with three columns named "itemId," "itemName," and "price." For each row in the result set, an Object instance is created with properties named `itemId`, `itemName`, and `price`. Those properties contain the values from their respective columns.

The following code listing defines a `SQLStatement` instance whose text is a `SELECT` statement. The statement retrieves rows containing the `firstName` and `lastName` column values of all the rows of a table named `employees`. This example uses asynchronous execution mode. When the execution completes, the `selectResult()` method is called, and the resulting rows of data are accessed using `SQLStatement.getResult()` and displayed using the `trace()` method. Note that this listing assumes there is a `SQLConnection` instance named `conn` that has already been instantiated and is already connected to the database. It also assumes that the "employees" table has already been created and populated with data.

```
// Include AIRAliases.js to use air.* shortcuts

// ... create and open the SQLConnection instance named conn ...

// create the SQL statement
var selectStmt = new air.SQLStatement();
selectStmt.sqlConnection = conn;

// define the SQL text
var sql =
    "SELECT firstName, lastName " +
    "FROM employees";
selectStmt.text = sql;

// register listeners for the result and error events
selectStmt.addEventListener(air.SQLEvent.RESULT, selectResult);
selectStmt.addEventListener(air.SQLErrorEvent.ERROR, selectError);

// execute the statement
selectStmt.execute();

function selectResult(event)
{
```

```
// access the result data
var result = selectStmt.getResult();

var numRows = result.data.length;
for (i = 0; i < numRows; i++)
{
    var output = "";
    for (columnName in result.data[i])
    {
        output += columnName + ": " + result.data[i][columnName] + " ";
    }
    air.trace("row[" + i.toString() + "]\t", output);
}

function selectError(event)
{
    air.trace("Error message:", event.error.message);
    air.trace("Details:", event.error.details);
}
```

The following code listing demonstrates the same techniques as the preceding one, but uses synchronous execution mode. The example defines a `SQLStatement` instance whose text is a `SELECT` statement. The statement retrieves rows containing the `firstName` and `lastName` column values of all the rows of a table named `employees`. The resulting rows of data are accessed using `SQLStatement.getResult()` and displayed using the `trace()` method. Note that this listing assumes there is a `SQLConnection` instance named `conn` that has already been instantiated and is already connected to the database. It also assumes that the “employees” table has already been created and populated with data.

```
// Include AIRAliases.js to use air.* shortcuts

// ... create and open the SQLConnection instance named conn ...

// create the SQL statement
var selectStmt = new air.SQLStatement();
selectStmt.sqlConnection = conn;

// define the SQL text
var sql =
    "SELECT firstName, lastName " +
    "FROM employees";
selectStmt.text = sql;

try
{
    // execute the statement
    selectStmt.execute();

    // access the result data
    var result = selectStmt.getResult();

    var numRows = result.data.length;
    for (i = 0; i < numRows; i++)
    {
        var output = "";
        for (columnName in result.data[i])
        {
            output += columnName + ": " + result.data[i][columnName] + "; ";
        }
        air.trace("row[" + i.toString() + "]\t", output);
    }
}
catch (error)
{
    air.trace("Error message:", error.message);
    air.trace("Details:", error.details);
}
```

Defining the data type of SELECT result data

Adobe AIR 1.0 and later

By default, each row returned by a `SELECT` statement is created as an Object instance with properties named for the result set's column names and with the value of each column as the value of its associated property. However, before executing a SQL `SELECT` statement, you can set the `itemClass` property of the `SQLStatement` instance to a class. By setting the `itemClass` property, each row returned by the `SELECT` statement is created as an instance of the designated class. The runtime assigns result column values to property values by matching the column names in the `SELECT` result set to the names of the properties in the `itemClass` class.

Any class assigned as an `itemClass` property value must have a constructor that does not require any parameters. In addition, the class must have a single property for each column returned by the `SELECT` statement. It is considered an error if a column in the `SELECT` list does not have a matching property name in the `itemClass` class.

Retrieving SELECT results in parts

Adobe AIR 1.0 and later

By default, a `SELECT` statement execution retrieves all the rows of the result set at one time. Once the statement completes, you usually process the retrieved data in some way, such as creating objects or displaying the data on the screen. If the statement returns a large number of rows, processing all the data at once can be demanding for the computer, which in turn will cause the user interface to not redraw itself.

You can improve the perceived performance of your application by instructing the runtime to return a specific number of result rows at a time. Doing so causes the initial result data to return more quickly. It also allows you to divide the result rows into sets, so that the user interface is updated after each set of rows is processed. Note that it's only practical to use this technique in asynchronous execution mode.

To retrieve `SELECT` results in parts, specify a value for the `SQLStatement.execute()` method's first parameter (the `prefetch` parameter). The `prefetch` parameter indicates the number of rows to retrieve the first time the statement executes. When you call a `SQLStatement` instance's `execute()` method, specify a `prefetch` parameter value and only that many rows are retrieved:

```
// Include AIRAliases.js to use air.* shortcuts
var stmt = new air.SQLStatement();
stmt.sqlConnection = conn;

stmt.text = "SELECT ...";

stmt.addEventListener(air.SQLEvent.RESULT, selectResult);

stmt.execute(20); // only the first 20 rows (or fewer) are returned
```

The statement dispatches the `result` event, indicating that the first set of result rows is available. The resulting `SQLResult` instance's `data` property contains the rows of data, and its `complete` property indicates whether there are additional result rows to retrieve. To retrieve additional result rows, call the `SQLStatement` instance's `next()` method. Like the `execute()` method, the `next()` method's first parameter is used to indicate how many rows to retrieve the next time the result event is dispatched.

```
function selectResult(event)
{
    var result = stmt.getResult();
    if (result.data != null)
    {
        // ... loop through the rows or perform other processing ...

        if (!result.complete)
        {
            stmt.next(20); // retrieve the next 20 rows
        }
        else
        {
            stmt.removeEventListener(air.SQLEvent.RESULT, selectResult);
        }
    }
}
```

The `SQLStatement` dispatches a `result` event each time the `next()` method returns a subsequent set of result rows. Consequently, the same listener function can be used to continue processing results (from `next()` calls) until all the rows are retrieved.

For more information, see the descriptions for the `SQLStatement.execute()` method (the `prefetch` parameter description) and the `SQLStatement.next()` method.

Inserting data

Adobe AIR 1.0 and later

Adding data to a database involves executing a SQL `INSERT` statement. Once the statement has finished executing, you can access the primary key for the newly inserted row if the key was generated by the database.

Executing an INSERT statement

Adobe AIR 1.0 and later

To add data to a table in a database, you create and execute a `SQLStatement` instance whose text is a SQL `INSERT` statement.

The following example uses a `SQLStatement` instance to add a row of data to the already-existing `employees` table. This example demonstrates inserting data using asynchronous execution mode. Note that this listing assumes that there is a `SQLConnection` instance named `conn` that has already been instantiated and is already connected to a database. It also assumes that the “employees” table has already been created.

```
// Include AIRAliases.js to use air.* shortcuts

// ... create and open the SQLConnection instance named conn ...

// create the SQL statement
var insertStmt = new air.SQLStatement();
insertStmt.sqlConnection = conn;

// define the SQL text
var sql =
    "INSERT INTO employees (firstName, lastName, salary) " +
    "VALUES ('Bob', 'Smith', 8000)";
insertStmt.text = sql;

// register listeners for the result and failure (status) events
insertStmt.addEventListener(air.SQLEvent.RESULT, insertResult);
insertStmt.addEventListener(air.SQLErrorEvent.ERROR, insertError);

// execute the statement
insertStmt.execute();

function insertResult(event)
{
    air.trace("INSERT statement succeeded");
}

function insertError(event)
{
    air.trace("Error message:", event.error.message);
    air.trace("Details:", event.error.details);
}
```

The following example adds a row of data to the already-existing employees table, using synchronous execution mode. Note that this listing assumes that there is a `SQLConnection` instance named `conn` that has already been instantiated and is already connected to a database. It also assumes that the “employees” table has already been created.

```
// Include AIRAliases.js to use air.* shortcuts

// ... create and open the SQLConnection instance named conn ...

// create the SQL statement
var insertStmt = new air.SQLStatement();
insertStmt.sqlConnection = conn;

// define the SQL text
var sql =
    "INSERT INTO employees (firstName, lastName, salary) " +
    "VALUES ('Bob', 'Smith', 8000)";
insertStmt.text = sql;

try
{
    // execute the statement
    insertStmt.execute();

    air.trace("INSERT statement succeeded");
}
catch (error)
{
    air.trace("Error message:", error.message);
    air.trace("Details:", error.details);
}
```

Retrieving a database-generated primary key of an inserted row

Adobe AIR 1.0 and later

Often after inserting a row of data into a table, your code needs to know a database-generated primary key or row identifier value for the newly inserted row. For example, once you insert a row in one table, you might want to add rows in a related table. In that case you would want to insert the primary key value as a foreign key in the related table. The primary key of a newly inserted row can be retrieved using the `SQLResult` object associated with the statement execution. This is the same object that's used to access result data after a `SELECT` statement is executed. As with any SQL statement, when the execution of an `INSERT` statement completes the runtime creates a `SQLResult` instance. You access the `SQLResult` instance by calling the `SQLStatement` object's `getResult()` method if you're using an event listener or if you're using synchronous execution mode. Alternatively, if you're using asynchronous execution mode and you pass a `Responder` instance to the `execute()` call, the `SQLResult` instance is passed as an argument to the result handler function. In any case, the `SQLResult` instance has a property, `lastInsertRowID`, that contains the row identifier of the most-recently inserted row if the executed SQL statement is an `INSERT` statement.

The following example demonstrates accessing the primary key of an inserted row in asynchronous execution mode:


```
insertStmt.text = "INSERT INTO ...";

insertStmt.addEventListener(air.SQLEvent.RESULT, resultHandler);

insertStmt.execute();

function resultHandler(event)
{
    // get the primary key
    var result = insertStmt.getResult();

    var primaryKey = result.lastInsertRowID;
    // do something with the primary key
}
```

The following example demonstrates accessing the primary key of an inserted row in synchronous execution mode:

```
insertStmt.text = "INSERT INTO ...";

try
{
    insertStmt.execute();

    // get the primary key
    var result = insertStmt.getResult();

    var primaryKey = result.lastInsertRowID;
    // do something with the primary key
}
catch (error)
{
    // respond to the error
}
```

Note that the row identifier may or may not be the value of the column that is designated as the primary key column in the table definition, according to the following rules:

- If the table is defined with a primary key column whose affinity (column data type) is `INTEGER`, the `lastInsertRowID` property contains the value that was inserted into that row (or the value generated by the runtime if it's an `AUTOINCREMENT` column).
- If the table is defined with multiple primary key columns (a composite key) or with a single primary key column whose affinity is not `INTEGER`, behind the scenes the database generates an integer row identifier value for the row. That generated value is the value of the `lastInsertRowID` property.
- The value is always the row identifier of the most-recently inserted row. If an `INSERT` statement causes a trigger to fire which in turn inserts a row, the `lastInsertRowID` property contains the row identifier of the last row inserted by the trigger rather than the row created by the `INSERT` statement.

As a consequence of these rules, if you want to have an explicitly defined primary key column whose value is available after an `INSERT` command through the `SQLResult.lastInsertRowID` property, the column must be defined as an `INTEGER PRIMARY KEY` column. Even if your table does not include an explicit `INTEGER PRIMARY KEY` column, it is equally acceptable to use the database-generated row identifier as a primary key for your table in the sense of defining relationships with related tables. The row identifier column value is available in any SQL statement by using one of the special column names `ROWID`, `_ROWID_`, or `OID`. You can create a foreign key column in a related table and use the row

identifier value as the foreign key column value just as you would with an explicitly declared `INTEGER PRIMARY KEY` column. In that sense, if you are using an arbitrary primary key rather than a natural key, and as long as you don't mind the runtime generating the primary key value for you, it makes little difference whether you use an `INTEGER PRIMARY KEY` column or the system-generated row identifier as a table's primary key for defining a foreign key relationship with between two tables.

For more information about primary keys and generated row identifiers, see [“SQL support in local databases”](#) on page 341.

Changing or deleting data

Adobe AIR 1.0 and later

The process for executing other data manipulation operations is identical to the process used to execute a SQL `SELECT` or `INSERT` statement, as described in [“Working with SQL statements”](#) on page 214. Simply substitute a different SQL statement in the `SQLStatement` instance's `text` property:

- To change existing data in a table, use an `UPDATE` statement.
- To delete one or more rows of data from a table, use a `DELETE` statement.

For descriptions of these statements, see [“SQL support in local databases”](#) on page 341.

Working with multiple databases

Adobe AIR 1.0 and later

Use the `SQLConnection.attach()` method to open a connection to an additional database on a `SQLConnection` instance that already has an open database. You give the attached database a name using the name parameter in the `attach()` method call. When writing statements to manipulate that database, you can then use that name in a prefix (using the form `database-name.table-name`) to qualify any table names in your SQL statements, indicating to the runtime that the table can be found in the named database.

You can execute a single SQL statement that includes tables from multiple databases that are connected to the same `SQLConnection` instance. If a transaction is created on the `SQLConnection` instance, that transaction applies to all SQL statements that are executed using the `SQLConnection` instance. This is true regardless of which attached database the statement runs on.

Alternatively, you can also create multiple `SQLConnection` instances in an application, each of which is connected to one or multiple databases. However, if you do use multiple connections to the same database keep in mind that a database transaction isn't shared across `SQLConnection` instances. Consequently, if you connect to the same database file using multiple `SQLConnection` instances, you can't rely on both connections' data changes being applied in the expected manner. For example, if two `UPDATE` or `DELETE` statements are run against the same database through different `SQLConnection` instances, and an application error occurs after one operation takes place, the database data could be left in an intermediate state that would not be reversible and might affect the integrity of the database (and consequently the application).

Handling database errors

Adobe AIR 1.0 and later

In general, database error handling is like other runtime error handling. You should write code that is prepared for errors that may occur, and respond to the errors rather than leave it up to the runtime to do so. In a general sense, the possible database errors can be divided into three categories: connection errors, SQL syntax errors, and constraint errors.

Connection errors

Adobe AIR 1.0 and later

Most database errors are connection errors, and they can occur during any operation. Although there are strategies for preventing connection errors, there is rarely a simple way to gracefully recover from a connection error if the database is a critical part of your application.

Most connection errors have to do with how the runtime interacts with the operating system, the file system, and the database file. For example, a connection error occurs if the user doesn't have permission to create a database file in a particular location on the file system. The following strategies help to prevent connection errors:

Use user-specific database files Rather than using a single database file for all users who use the application on a single computer, give each user their own database file. The file should be located in a directory that's associated with the user's account. For example, it could be in the application's storage directory, the user's documents folder, the user's desktop, and so forth.

Consider different user types Test your application with different types of user accounts, on different operating systems. Don't assume that the user has administrator permission on the computer. Also, don't assume that the individual who installed the application is the user who's running the application.

Consider various file locations If you allow a user to specify where to save a database file or select a file to open, consider the possible file locations that the users might use. In addition, consider defining limits on where users can store (or from where they can open) database files. For example, you might only allow users to open files that are within their user account's storage location.

If a connection error occurs, it most likely happens on the first attempt to create or open the database. This means that the user is unable to do any database-related operations in the application. For certain types of errors, such as read-only or permission errors, one possible recovery technique is to copy the database file to a different location. The application can copy the database file to a different location where the user does have permission to create and write to files, and use that location instead.

Syntax errors

Adobe AIR 1.0 and later

A syntax error occurs when a SQL statement is incorrectly formed, and the application attempts to execute the statement. Because local database SQL statements are created as strings, compile-time SQL syntax checking is not possible. All SQL statements must be executed to check their syntax. Use the following strategies to prevent SQL syntax errors:

Test all SQL statements thoroughly If possible, while developing your application test your SQL statements separately before encoding them as statement text in the application code. In addition, use a code-testing approach such as unit testing to create a set of tests that exercise every possible option and variation in the code.

Use statement parameters and avoid concatenating (dynamically generating) SQL Using parameters, and avoiding dynamically built SQL statements, means that the same SQL statement text is used each time a statement is executed. Consequently, it's much easier to test your statements and limit the possible variation. If you must dynamically generate a SQL statement, keep the dynamic parts of the statement to a minimum. Also, carefully validate any user input to make sure it won't cause syntax errors.

To recover from a syntax error, an application would need complex logic to be able to examine a SQL statement and correct its syntax. By following the previous guidelines for preventing syntax errors, your code can identify any potential run-time sources of SQL syntax errors (such as user input used in a statement). To recover from a syntax error, provide guidance to the user. Indicate what to correct to make the statement execute properly.

Constraint errors

Adobe AIR 1.0 and later

Constraint errors occur when an `INSERT` or `UPDATE` statement attempts to add data to a column. The error happens if the new data violates one of the defined constraints for the table or column. The set of possible constraints includes:

Unique constraint Indicates that across all the rows in a table, there cannot be duplicate values in one column. Alternatively, when multiple columns are combined in a unique constraint, the combination of values in those columns must not be duplicated. In other words, in terms of the specified unique column or columns, each row must be distinct.

Primary key constraint In terms of the data that a constraint allows and doesn't allow, a primary key constraint is identical to a unique constraint.

Not null constraint Specifies that a single column cannot store a `NULL` value and consequently that in every row, that column must have a value.

Check constraint Allows you to specify an arbitrary constraint on one or more tables. A common check constraint is a rule that define that a column's value must be within certain bounds (for example, that a numeric column's value must be larger than 0). Another common type of check constraint specifies relationships between column values (for example, that a column's value must be different from the value of another column in the same row).

Data type (column affinity) constraint The runtime enforces the data type of columns' values, and an error occurs if an attempt is made to store a value of the incorrect type in a column. However, in many conditions values are converted to match the column's declared data type. See "[Working with database data types](#)" on page 231 for more information.

The runtime does not enforce constraints on foreign key values. In other words, foreign key values aren't required to match an existing primary key value.

In addition to the predefined constraint types, the runtime SQL engine supports the use of triggers. A trigger is like an event handler. It is a predefined set of instructions that are carried out when a certain action happens. For example, a trigger could be defined that runs when data is inserted into or deleted from a particular table. One possible use of a trigger is to examine data changes and cause an error to occur if specified conditions aren't met. Consequently, a trigger can serve the same purpose as a constraint, and the strategies for preventing and recovering from constraint errors also apply to trigger-generated errors. However, the error id for trigger-generated errors is different from the error id for constraint errors.

The set of constraints that apply to a particular table is determined while you're designing an application. Consciously designing constraints makes it easier to design your application to prevent and recover from constraint errors. However, constraint errors are difficult to systematically predict and prevent. Prediction is difficult because constraint errors don't appear until application data is added. Constraint errors occur with data that is added to a database after

it's created. These errors are often a result of the relationship between new data and data that already exists in the database. The following strategies can help you avoid many constraint errors:

Carefully plan database structure and constraints The purpose of constraints is to enforce application rules and help protect the integrity of the database's data. When you're planning your application, consider how to structure your database to support your application. As part of that process, identify rules for your data, such as whether certain values are required, whether a value has a default, whether duplicate values are allowed, and so forth. Those rules guide you in defining database constraints.

Explicitly specify column names An `INSERT` statement can be written without explicitly specifying the columns into which values are to be inserted, but doing so is an unnecessary risk. By explicitly naming the columns into which values are to be inserted, you can allow for automatically generated values, columns with default values, and columns that allow `NULL` values. In addition, by doing so you can ensure that all `NOT NULL` columns have an explicit value inserted.

Use default values Whenever you specify a `NOT NULL` constraint for a column, if at all possible specify a default value in the column definition. Application code can also provide default values. For example, your code can check if a `String` variable is `null` and assign it a value before using it to set a statement parameter value.

Validate user-entered data Check user-entered data ahead of time to make sure that it obeys limits specified by constraints, especially in the case of `NOT NULL` and `CHECK` constraints. Naturally, a `UNIQUE` constraint is more difficult to check for because doing so would require executing a `SELECT` query to determine whether the data is unique.

Use triggers You can write a trigger that validates (and possibly replaces) inserted data or takes other actions to correct invalid data. This validation and correction can prevent a constraint error from occurring.

In many ways constraint errors are more difficult to prevent than other types of errors. Fortunately, there are several strategies to recover from constraint errors in ways that don't make the application unstable or unusable:

Use conflict algorithms When you define a constraint on a column, and when you create an `INSERT` or `UPDATE` statement, you have the option of specifying a conflict algorithm. A conflict algorithm defines the action the database takes when a constraint violation occurs. There are several possible actions the database engine can take. The database engine can end a single statement or a whole transaction. It can ignore the error. It can even remove old data and replace it with the data that the code is attempting to store.

For more information see the section "ON CONFLICT (conflict algorithms)" in the "[SQL support in local databases](#)" on page 341.

Provide corrective feedback The set of constraints that can affect a particular SQL command can be identified ahead of time. Consequently, you can anticipate constraint errors that a statement could cause. With that knowledge, you can build application logic to respond to a constraint error. For example, suppose an application includes a data entry form for entering new products. If the product name column in the database is defined with a `UNIQUE` constraint, the action of inserting a new product row in the database could cause a constraint error. Consequently, the application is designed to anticipate a constraint error. When the error happens, the application alerts the user, indicating that the specified product name is already in use and asking the user to choose a different name. Another possible response is to allow the user to view information about the other product with the same name.

Working with database data types

Adobe AIR 1.0 and later

When a table is created in a database, the SQL statement for creating the table defines the affinity, or data type, for each column in the table. Although affinity declarations can be omitted, it's a good idea to explicitly declare column affinity in your `CREATE TABLE` SQL statements.

As a general rule, any object that you store in a database using an `INSERT` statement is returned as an instance of the same data type when you execute a `SELECT` statement. However, the data type of the retrieved value can be different depending on the affinity of the database column in which the value is stored. When a value is stored in a column, if its data type doesn't match the column's affinity, the database attempts to convert the value to match the column's affinity. For example, if a database column is declared with `NUMERIC` affinity, the database attempts to convert inserted data into a numeric storage class (`INTEGER` or `REAL`) before storing the data. The database throws an error if the data can't be converted. According to this rule, if the String "12345" is inserted into a `NUMERIC` column, the database automatically converts it to the integer value 12345 before storing it in the database. When it's retrieved with a `SELECT` statement, the value is returned as an instance of a numeric data type (such as `Number`) rather than as a `String` instance.

The best way to avoid undesirable data type conversion is to follow two rules. First, define each column with the affinity that matches the type of data that it is intended to store. Next, only insert values whose data type matches the defined affinity. Following these rules provides two benefits. When you insert the data it isn't converted unexpectedly (possibly losing its intended meaning as a result). In addition, when you retrieve the data it is returned with its original data type.

For more information about the available column affinity types and using data types in SQL statements, see the "[Data type support](#)" on page 362.

Using synchronous and asynchronous database operations

Adobe AIR 1.0 and later

Previous sections have described common database operations such as retrieving, inserting, updating, and deleting data, as well as creating a database file and tables and other objects within a database. The examples have demonstrated how to perform these operations both asynchronously and synchronously.

As a reminder, in asynchronous execution mode, you instruct the database engine to perform an operation. The database engine then works in the background while the application keeps running. When the operation finishes the database engine dispatches an event to alert you to that fact. The key benefit of asynchronous execution is that the runtime performs the database operations in the background while the main application code continues executing. This is especially valuable when the operation takes a notable amount of time to run.

On the other hand, in synchronous execution mode operations don't run in the background. You tell the database engine to perform an operation. The code pauses at that point while the database engine does its work. When the operation completes, execution continues with the next line of your code.

A single database connection can't execute some operations or statements synchronously and others asynchronously. You specify whether a `SQLConnection` operates in synchronous or asynchronous when you open the connection to the database. If you call `SQLConnection.open()` the connection operates in synchronous execution mode, and if you call `SQLConnection.openAsync()` the connection operates in asynchronous execution mode. Once a `SQLConnection` instance is connected to a database using `open()` or `openAsync()`, it is fixed to synchronous or asynchronous execution.

Using synchronous database operations

Adobe AIR 1.0 and later

There is little difference in the actual code that you use to execute and respond to operations when using synchronous execution, compared to the code for asynchronous execution mode. The key differences between the two approaches fall into two areas. The first is executing an operation that depends on another operation (such as `SELECT` result rows or the primary key of the row added by an `INSERT` statement). The second area of difference is in handling errors.

Writing code for synchronous operations

Adobe AIR 1.0 and later

The key difference between synchronous and asynchronous execution is that in synchronous mode you write the code as a single series of steps. In contrast, in asynchronous code you register event listeners and often divide operations among listener methods. When a database is connected in synchronous execution mode, you can execute a series of database operations in succession within a single code block. The following example demonstrates this technique:

```
// Include AIRAliases.js to use air.* shortcuts

var conn = new air.SQLConnection();

// The database file is in the application storage directory
var folder = File.applicationStorageDirectory;
var dbFile = folder.resolvePath("DBSample.db");

// open the database
conn.open(dbFile, air.OpenMode.UPDATE);

// start a transaction
conn.begin();

// add the customer record to the database
var insertCustomer = new air.SQLStatement();
insertCustomer.sqlConnection = conn;
insertCustomer.text =
    "INSERT INTO customers (firstName, lastName) " +
    "VALUES ('Bob', 'Jones')";
insertCustomer.execute();

var customerId = insertCustomer.getResult().lastInsertRowID;

// add a related phone number record for the customer
var insertPhoneNumber = new air.SQLStatement();
insertPhoneNumber.sqlConnection = conn;
insertPhoneNumber.text =
    "INSERT INTO customerPhoneNumbers (customerId, number) " +
    "VALUES (:customerId, '800-555-1234')";
insertPhoneNumber.parameters[":customerId"] = customerId;
insertPhoneNumber.execute();

// commit the transaction
conn.commit();
```

As you can see, you call the same methods to perform database operations whether you're using synchronous or asynchronous execution. The key differences between the two approaches are executing an operation that depends on another operation and handling errors.

Executing an operation that depends on another operation

Adobe AIR 1.0 and later

When you're using synchronous execution mode, you don't need to write code that listens for an event to determine when an operation completes. Instead, you can assume that if an operation in one line of code completes successfully, execution continues with the next line of code. Consequently, to perform an operation that depends on the success of another operation, simply write the dependent code immediately following the operation on which it depends. For instance, to code an application to begin a transaction, execute an `INSERT` statement, retrieve the primary key of the inserted row, insert that primary key into another row of a different table, and finally commit the transaction, the code can all be written as a series of statements. The following example demonstrates these operations:

```
// Include AIRAliases.js to use air.* shortcuts

var conn = new air.SQLConnection();

// The database file is in the application storage directory
var folder = File.applicationStorageDirectory;
var dbFile = folder.resolvePath("DBSample.db");

// open the database
conn.open(dbFile, air.OpenMode.UPDATE);

// start a transaction
conn.begin();

// add the customer record to the database
var insertCustomer = new air.SQLStatement();
insertCustomer.sqlConnection = conn;
insertCustomer.text =
    "INSERT INTO customers (firstName, lastName) " +
    "VALUES ('Bob', 'Jones')";
insertCustomer.execute();

var customerId = insertCustomer.getResult().lastInsertRowID;

// add a related phone number record for the customer
var insertPhoneNumber = new air.SQLStatement();
insertPhoneNumber.sqlConnection = conn;
insertPhoneNumber.text =
    "INSERT INTO customerPhoneNumbers (customerId, number) " +
    "VALUES (:customerId, '800-555-1234')";
insertPhoneNumber.parameters[":customerId"] = customerId;
insertPhoneNumber.execute();

// commit the transaction
conn.commit();
```


Handling errors with synchronous execution

Adobe AIR 1.0 and later

In synchronous execution mode, you don't listen for an error event to determine that an operation has failed. Instead, you surround any code that could trigger errors in a set of `try...catch...finally` code blocks. You wrap the error-throwing code in the `try` block. Write the actions to perform in response to each type of error in separate `catch` blocks. Place any code that you want to always execute regardless of success or failure (for example, closing a database connection that's no longer needed) in a `finally` block. The following example demonstrates using `try...catch...finally` blocks for error handling. It builds on the previous example by adding error handling code:

```
// Include AIRAliases.js to use air.* shortcuts

var conn = new air.SQLConnection();

// The database file is in the application storage directory
var folder = File.applicationStorageDirectory;
var dbFile = folder.resolvePath("DBSample.db");

// open the database
conn.open(dbFile, air.SQLMode.UPDATE);

// start a transaction
conn.begin();

try
{
    // add the customer record to the database
    var insertCustomer = new air.SQLStatement();
    insertCustomer.sqlConnection = conn;
    insertCustomer.text =
        "INSERT INTO customers (firstName, lastName)" +
        "VALUES ('Bob', 'Jones')";

    insertCustomer.execute();

    var customerId = insertCustomer.getResult().lastInsertRowID;

    // add a related phone number record for the customer
    var insertPhoneNumber = new air.SQLStatement();
    insertPhoneNumber.sqlConnection = conn;
    insertPhoneNumber.text =
        "INSERT INTO customerPhoneNumbers (customerId, number)" +
        "VALUES (:customerId, '800-555-1234')";
    insertPhoneNumber.parameters[":customerId"] = customerId;

    insertPhoneNumber.execute();

    // if we've gotten to this point without errors, commit the transaction
    conn.commit();
}
catch (error)
{
    // rollback the transaction
    conn.rollback();
}
```

Understanding the asynchronous execution model

Adobe AIR 1.0 and later

One common concern about using asynchronous execution mode is the assumption that you can't start executing a `SQLStatement` instance if another `SQLStatement` is currently executing against the same database connection. In fact, this assumption isn't correct. While a `SQLStatement` instance is executing you can't change the `text` property of the statement. However, if you use a separate `SQLStatement` instance for each different SQL statement that you want to execute, you can call the `execute()` method of a `SQLStatement` while another `SQLStatement` instance is still executing, without causing an error.

Internally, when you're executing database operations using asynchronous execution mode, each database connection (each `SQLConnection` instance) has its own queue or list of operations that it is instructed to perform. The runtime executes each operation in sequence, in the order they are added to the queue. When you create a `SQLStatement` instance and call its `execute()` method, that statement execution operation is added to the queue for the connection. If no operation is currently executing on that `SQLConnection` instance, the statement begins executing in the background. Suppose that within the same block of code you create another `SQLStatement` instance and also call that method's `execute()` method. That second statement execution operation is added to the queue behind the first statement. As soon as the first statement finishes executing, the runtime moves to the next operation in the queue. The processing of subsequent operations in the queue happens in the background, even while the `result` event for the first operation is being dispatched in the main application code. The following code demonstrates this technique:

```
// Using asynchronous execution mode
var stmt1 = new air.SQLStatement();
stmt1.sqlConnection = conn;

// ... Set statement text and parameters, and register event listeners ...

stmt1.execute();

// At this point stmt1's execute() operation is added to conn's execution queue.

var stmt2 = new air.SQLStatement();
stmt2.sqlConnection = conn;

// ... Set statement text and parameters, and register event listeners ...

stmt2.execute();

// At this point stmt2's execute() operation is added to conn's execution queue.
// When stmt1 finishes executing, stmt2 will immediately begin executing
// in the background.
```

There is an important side effect of the database automatically executing subsequent queued statements. If a statement depends on the outcome of another operation, you can't add the statement to the queue (in other words, you can't call its `execute()` method) until the first operation completes. This is because once you've called the second statement's `execute()` method, you can't change the statement's `text` or `parameters` properties. In that case you must wait for the event indicating that the first operation completes before starting the next operation. For example, if you want to execute a statement in the context of a transaction, the statement execution depends on the operation of opening the transaction. After calling the `SQLConnection.begin()` method to open the transaction, you need to wait for the `SQLConnection` instance to dispatch its `begin` event. Only then can you call the `SQLStatement` instance's `execute()` method. In this example the simplest way to organize the application to ensure that the operations are executed properly is to create a method that's registered as a listener for the `begin` event. The code to call the `SQLStatement.execute()` method is placed within that listener method.

Using encryption with SQL databases

Adobe AIR 1.5 and later

All Adobe AIR applications share the same local database engine. Consequently, any AIR application can connect to, read from, and write to an unencrypted database file. Starting with Adobe AIR 1.5, AIR includes the capability of creating and connecting to encrypted database files. When you use an encrypted database, in order to connect to the database an application must provide the correct encryption key. If the incorrect encryption key (or no key) is provided, the application is not able to connect to the database. Consequently, the application can't read data from the database or write to or change data in the database.

To use an encrypted database, you must create the database as an encrypted database. With an existing encrypted database, you can open a connection to the database. You can also change the encryption key of an encrypted database. Other than creating and connecting to encrypted databases, the techniques for working with an encrypted database are the same as for working with an unencrypted one. In particular, executing SQL statements is the same regardless of whether a database is encrypted or not.

Uses for an encrypted database

Adobe AIR 1.5 and later

Encryption is useful any time you want to restrict access to the information stored in a database. The database encryption functionality of Adobe AIR can be used for several purposes. The following are some examples of cases where you would want to use an encrypted database:

- A read-only cache of private application data downloaded from a server
- A local application store for private data that is synchronized with a server (data is sent to and loaded from the server)
- Encrypted files used as the file format for documents created and edited by the application. The files could be private to one user, or could be designed to be shared among all users of the application.
- Any other use of a local data store, such as the ones described in [“Uses for local SQL databases”](#) on page 204, where the data must be kept private from people who have access to the machine or the database files.

Understanding the reason why you want to use an encrypted database helps you decide how to architect your application. In particular, it can affect how your application creates, obtains, and stores the encryption key for the database. For more information about these considerations, see [“Considerations for using encryption with a database”](#) on page 240.

Other than an encrypted database, an alternative mechanism for keeping sensitive data private is the encrypted local store. With the encrypted local store, you store a single ByteArray value using a String key. Only the AIR application that stores the value can access it, and only on the computer on which it is stored. With the encrypted local store, it isn't necessary to create your own encryption key. For these reasons, the encrypted local store is most suitable for easily storing a single value or set of values that can easily be encoded in a ByteArray. An encrypted database is most suitable for larger data sets where structured data storage and querying are desirable. For more information about using the encrypted local store, see [“Encrypted local storage”](#) on page 256.

Creating an encrypted database

Adobe AIR 1.5 and later

To use an encrypted database, the database file must be encrypted when it is created. Once a database is created as unencrypted, it can't be encrypted later. Likewise, an encrypted database can't be unencrypted later. If needed you can change the encryption key of an encrypted database. For details, see [“Changing the encryption key of a database”](#) on page 240. If you have an existing database that's not encrypted and you want to use database encryption, you can create a new encrypted database and copy the existing table structure and data to the new database.

Creating an encrypted database is nearly identical to creating an unencrypted database, as described in [“Creating a database”](#) on page 209. You first create a `SQLConnection` instance that represents the connection to the database. You create the database by calling the `SQLConnection` object's `open()` method or `openAsync()` method, specifying for the database location a file that doesn't exist yet. The only difference when creating an encrypted database is that you provide a value for the `encryptionKey` parameter (the `open()` method's fifth parameter and the `openAsync()` method's sixth parameter).

A valid `encryptionKey` parameter value is a `ByteArray` object containing exactly 16 bytes.

The following examples demonstrate creating an encrypted database. For simplicity, in these examples the encryption key is hard-coded in the application code. However, this technique is strongly discouraged because it is not secure.

```
var conn = new air.SQLConnection();

var encryptionKey = new air.ByteArray();
encryptionKey.writeUTFBytes("Some16ByteString"); // This technique is not secure!

// Create an encrypted database in asynchronous mode
conn.openAsync(dbFile, air.SQLMode.CREATE, null, false, 1024, encryptionKey);

// Create an encrypted database in synchronous mode
conn.open(dbFile, air.SQLMode.CREATE, false, 1024, encryptionKey);
```

For an example demonstrating a recommended way to generate an encryption key, see [“Example: Generating and using an encryption key”](#) on page 242.

Connecting to an encrypted database

Adobe AIR 1.5 and later

Like creating an encrypted database, the procedure for opening a connection to an encrypted database is like connecting to an unencrypted database. That procedure is described in greater detail in [“Connecting to a database”](#) on page 213. You use the `open()` method to open a connection in synchronous execution mode, or the `openAsync()` method to open a connection in asynchronous execution mode. The only difference is that to open an encrypted database, you specify the correct value for the `encryptionKey` parameter (the `open()` method's fifth parameter and the `openAsync()` method's sixth parameter).

If the encryption key that's provided is not correct, an error occurs. For the `open()` method, a `SQLException` exception is thrown. For the `openAsync()` method, the `SQLConnection` object dispatches a `SQLExceptionEvent`, whose `error` property contains a `SQLException` object. In either case, the `SQLException` object generated by the exception has the `errorID` property value 3138. That error ID corresponds to the error message “File opened is not a database file.”

The following example demonstrates opening an encrypted database in asynchronous execution mode. For simplicity, in this example the encryption key is hard-coded in the application code. However, this technique is strongly discouraged because it is not secure.

```
// Include AIRAliases.js to use air.* shortcuts
var conn = new air.SQLConnection();
conn.addEventListener(air.SQLEvent.OPEN, openHandler);
conn.addEventListener(air.SQLErrorEvent.ERROR, errorHandler);
var dbFile = air.File.applicationStorageDirectory.resolvePath("DBSample.db");

var encryptionKey = new air.ByteArray();
encryptionKey.writeUTFBytes("Some16ByteString"); // This technique is not secure!

conn.openAsync(dbFile, air.SQLMode.UPDATE, null, false, 1024, encryptionKey);

function openHandler(event)
{
    air.trace("the database opened successfully");
}

function errorHandler(event)
{
    if (event.error.errorID == 3138)
    {
        air.trace("Incorrect encryption key");
    }
    else
    {
        air.trace("Error message:", event.error.message);
        air.trace("Details:", event.error.details);
    }
}
}
```

The following example demonstrates opening an encrypted database in synchronous execution mode. For simplicity, in this example the encryption key is hard-coded in the application code. However, this technique is strongly discouraged because it is not secure.

```
// Include AIRAliases.js to use air.* shortcuts
var conn = new air.SQLConnection();
var dbFile = air.File.applicationStorageDirectory.resolvePath("DBSample.db");

var encryptionKey = new air.ByteArray();
encryptionKey.writeUTFBytes("Some16ByteString"); // This technique is not secure!

try
{
    conn.open(dbFile, air.SQLMode.UPDATE, false, 1024, encryptionKey);
    air.trace("the database was created successfully");
}
catch (error)
{
    if (error.errorID == 3138)
    {
        air.trace("Incorrect encryption key");
    }
    else
    {
        air.trace("Error message:", error.message);
        air.trace("Details:", error.details);
    }
}
}
```

For an example demonstrating a recommended way to generate an encryption key, see [“Example: Generating and using an encryption key”](#) on page 242.

Changing the encryption key of a database

Adobe AIR 1.5 and later

When a database is encrypted, you can change the encryption key for the database at a later time. To change a database's encryption key, first open a connection to the database by creating a `SQLConnection` instance and calling its `open()` or `openAsync()` method. Once the database is connected, call the `reencrypt()` method, passing the new encryption key as an argument.

Like most database operations, the `reencrypt()` method's behavior varies depending on whether the database connection uses synchronous or asynchronous execution mode. If you use the `open()` method to connect to the database, the `reencrypt()` operation runs synchronously. When the operation finishes, execution continues with the next line of code:

```
var newKey = new air.ByteArray();  
// ... generate the new key and store it in newKey  
conn.reencrypt(newKey);
```

On the other hand, if the database connection is opened using the `openAsync()` method, the `reencrypt()` operation is asynchronous. Calling `reencrypt()` begins the reencryption process. When the operation completes, the `SQLConnection` object dispatches a `reencrypt` event. You use an event listener to determine when the reencryption finishes:

```
var newKey = new air.ByteArray();  
// ... generate the new key and store it in newKey  
  
conn.addEventListener(air.SQLEvent.REENCRYPT, reencryptHandler);  
  
conn.reencrypt(newKey);  
  
function reencryptHandler(event)  
{  
    // save the fact that the key changed  
}
```

The `reencrypt()` operation runs in its own transaction. If the operation is interrupted or fails (for example, if the application is closed before the operation finishes) the transaction is rolled back. In that case, the original encryption key is still the encryption key for the database.

The `reencrypt()` method can't be used to remove encryption from a database. Passing a `null` value or encryption key that's not a 16-byte `ByteArray` to the `reencrypt()` method results in an error.

Considerations for using encryption with a database

Adobe AIR 1.5 and later

The section [“Uses for an encrypted database”](#) on page 237 presents several cases in which you would want to use an encrypted database. It's obvious that the usage scenarios of different applications (including these and other scenarios) have different privacy requirements. The way you architect the use of encryption in your application plays an important part in controlling how private a database's data is. For example, if you are using an encrypted database to keep personal data private, even from other users of the same machine, then each user's database needs its own

encryption key. For the greatest security, your application can generate the key from a user-entered password. Basing the encryption key on a password ensures that even if another person is able to impersonate the user's account on the machine, the data still can't be accessed. On the other end of the privacy spectrum, suppose you want a database file to be readable by any user of your application but not to other applications. In that case every installed copy of the application needs access to a shared encryption key.

You can design your application, and in particular the technique used to generate the encryption key, according to the level of privacy that you want for your application data. The following list provides design suggestions for various levels of data privacy:

- To make a database accessible to any user who has access to the application on any machine, use a single key that's available to all instances of the application. For example, the first time an application runs it can download the shared encryption key from a server using a secure protocol such as SSL. It can then save the key in the encrypted local store for future use. As an alternative, encrypt the data per-user on the machine, and synchronize the data with a remote data store such as a server to make the data portable.
- To make a database accessible to a single user on any machine, generate the encryption key from a user secret (such as a password). In particular, do not use any value that's tied to a particular computer (such as a value stored in the encrypted local store) to generate the key. As an alternative, encrypt the data per-user on the machine, and synchronize the data with a remote data store such as a server to make the data portable.
- To make a database accessible only to a single individual on a single machine, generate the key from a password and a generated salt. For an example of this technique, see [“Example: Generating and using an encryption key”](#) on page 242.

The following are additional security considerations that are important to keep in mind when designing an application to use an encrypted database:

- A system is only as secure as its weakest link. If you are using a user-entered password to generate an encryption key, consider imposing minimum length and complexity restrictions on passwords. A short password that uses only basic characters can be guessed quickly.
- The source code of an AIR application is stored on a user's machine in plain text (for HTML content) or an easily decompilable binary format (for SWF content). Because the source code is accessible, two points to remember are:
 - Never hard-code an encryption key in your source code
 - Always assume that the technique used to generate an encryption key (such as random character generator or a particular hashing algorithm) can be easily worked out by an attacker
- AIR database encryption uses the Advanced Encryption Standard (AES) with Counter with CBC-MAC (CCM) mode. This encryption cipher requires a user-entered key to be combined with a salt value to be secure. For an example of this, see [“Example: Generating and using an encryption key”](#) on page 242.
- When you elect to encrypt a database, all disk files used by the database engine in conjunction with that database are encrypted. However, the database engine holds some data temporarily in an in-memory cache to improve read- and write-time performance in transactions. Any memory-resident data is unencrypted. If an attacker is able to access the memory used by an AIR application, for example by using a debugger, the data in a database that is currently open and unencrypted would be available.

Example: Generating and using an encryption key

Adobe AIR 1.5 and later

This example application demonstrates one technique for generating an encryption key. This application is designed to provide the highest level of privacy and security for users' data. One important aspect of securing private data is to require the user to enter a password each time the application connects to the database. Consequently, as shown in this example, an application that requires this level of privacy should never directly store the database encryption key.

The application consists of two parts: an ActionScript class that generates an encryption key (the `EncryptionKeyGenerator` class), and a basic user interface that demonstrates how to use that class. For the complete source code, see [“Complete example code for generating and using an encryption key”](#) on page 244.

Using the `EncryptionKeyGenerator` class to obtain a secure encryption key

Adobe AIR 1.5 and later

It isn't necessary to understand the details of how the `EncryptionKeyGenerator` class works to use it in your application. If you are interested in the details of how the class generates an encryption key for a database, see [“Understanding the `EncryptionKeyGenerator` class”](#) on page 251.

Follow these steps to use the `EncryptionKeyGenerator` class in your application:

- 1 Download the `EncryptionKeyGenerator` library. The `EncryptionKeyGenerator` class is included in the open-source ActionScript 3.0 core library (`as3corelib`) project. You can download [the `as3corelib` package including source code and documentation](#). You can also download the SWC or source code files from the project page.
- 2 Extract the SWF file from the SWC. To extract the SWF file, rename the SWC file with the “.zip” filename extension and open the ZIP file. Extract the SWF file from the ZIP file and place it in a location where your application source code can find it. For example, you could place it in the folder containing your application's main HTML file. You can rename the SWF file if you desire. In this example, the SWF file is named “`EncryptionKeyGenerator.swf`.”
- 3 In your application source code, import the SWF code library by adding a `<script>` block linking to the SWF file. This technique is explained in [“Using ActionScript libraries within an HTML page”](#) on page 32. The following code makes the SWF file available as a code library:

```
<script type="application/x-shockwave-flash" src="EncryptionKeyGenerator.swf"/>
```

By default the class is available using the code `window.runtime` followed by the full package and class name. For the `EncryptionKeyGenerator`, the full name is:

```
window.runtime.com.adobe.air.crypto.EncryptionKeyGenerator
```

You can create an alias for the class to avoid having to type the full name. The following code creates the alias `ekg.EncryptionKeyGenerator` to represent the `EncryptionKeyGenerator` class:

```
var ekg;  
if (window.runtime)  
{  
    if (!ekg) ekg = {};  
    ekg.EncryptionKeyGenerator = window.runtime.com.adobe.air.crypto.EncryptionKeyGenerator;  
}
```

- 4 Before the point where the code creates the database or opens a connection to it, add code to create an `EncryptionKeyGenerator` instance by calling the `EncryptionKeyGenerator()` constructor.

```
var keyGenerator = new ekg.EncryptionKeyGenerator();
```

- 5 Obtain a password from the user:


```
var password = passwordInput.value;

if (!keyGenerator.validateStrongPassword(password))
{
    // display an error message
    return;
}
```

The `EncryptionKeyGenerator` instance uses this password as the basis for the encryption key (shown in the next step). The `EncryptionKeyGenerator` instance tests the password against certain strong password validation requirements. If the validation fails, an error occurs. As the example code shows, you can check the password ahead of time by calling the `EncryptionKeyGenerator` object's `validateStrongPassword()` method. That way you can determine whether the password meets the minimum requirements for a strong password and avoid an error.

6 Generate the encryption key from the password:

```
var encryptionKey = keyGenerator.getEncryptionKey(password);
```

The `getEncryptionKey()` method generates and returns the encryption key (a 16-byte `ByteArray`). You can then use the encryption key to create your new encrypted database or open your existing one.

The `getEncryptionKey()` method has one required parameter, which is the password obtained in step 5.

Note: *To maintain the highest level of security and privacy for data, an application must require the user to enter a password each time the application connects to the database. Do not store the user's password or the database encryption key directly. Doing so exposes security risks. Instead, as demonstrated in this example, an application should use the same technique to derive the encryption key from the password both when creating the encrypted database and when connecting to it later.*

The `getEncryptionKey()` method also accepts a second (optional) parameter, the `overrideSaltELSKey` parameter. The `EncryptionKeyGenerator` creates a random value (known as a *salt*) that is used as part of the encryption key. In order to be able to re-create the encryption key, the salt value is stored in the Encrypted Local Store (ELS) of your AIR application. By default, the `EncryptionKeyGenerator` class uses a particular String as the ELS key. Although unlikely, it's possible that the key can conflict with another key your application uses. Instead of using the default key, you might want to specify your own ELS key. In that case, specify a custom key by passing it as the second `getEncryptionKey()` parameter, as shown here:

```
var customKey = "My custom ELS salt key";
var encryptionKey = keyGenerator.getEncryptionKey(password, customKey);
```

7 Create or open the database

With an encryption key returned by the `getEncryptionKey()` method, your code can create a new encrypted database or attempt to open the existing encrypted database. In either case you use the `SQLConnection` class's `open()` or `openAsync()` method, as described in [“Creating an encrypted database”](#) on page 238 and [“Connecting to an encrypted database”](#) on page 238.

In this example, the application is designed to open the database in asynchronous execution mode. The code sets up the appropriate event listeners and calls the `openAsync()` method, passing the encryption key as the final argument:

```
conn.addEventListener(air.SQLEvent.OPEN, openHandler);
conn.addEventListener(air.SQLErrorEvent.ERROR, openError);

conn.openAsync(dbFile, air.SQLMode.CREATE, null, false, 1024, encryptionKey);
```

In the listener methods, the code removes the event listener registrations. It then displays a status message indicating whether the database was created, opened, or whether an error occurred. The most noteworthy part of these event handlers is in the `openError()` method. In that method an `if` statement checks if the database exists (meaning that the code is attempting to connect to an existing database) and if the error ID matches the constant `EncryptionKeyGenerator.ENCRYPTED_DB_PASSWORD_ERROR_ID`. If both of these conditions are true, it probably means that the password the user entered is incorrect. (It could also mean that the specified file isn't a database file at all.) The following is the code that checks the error ID:

```
if (!createNewDB && event.error.errorID ==
    ekg.EncryptionKeyGenerator.ENCRYPTED_DB_PASSWORD_ERROR_ID)
{
    statusMsg.innerHTML = "<p class='error'>Incorrect password!</p>";
}
else
{
    statusMsg.innerHTML = "<p class='error'>Error creating or opening database.</p>";
}
```

For the complete code for the example event listeners, see [“Complete example code for generating and using an encryption key”](#) on page 244.

Complete example code for generating and using an encryption key

Adobe AIR 1.5 and later

The following is the complete code for the example application “Generating and using an encryption key.” The code consists of two parts.

The example uses the `EncryptionKeyGenerator` class to create an encryption key from a password. The `EncryptionKeyGenerator` class is included in the open-source ActionScript 3.0 core library (`as3corelib`) project. You can download [the as3corelib package including source code and documentation](#). You can also download the SWC or source code files from the project page.

Flex example

The application MXML file contains the source code for a simple application that creates or opens a connection to an encrypted database:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical"
creationComplete="init();">
  <mx:Script>
    <![CDATA[
      import com.adobe.air.crypto.EncryptionKeyGenerator;

      private const dbFileName:String = "encryptedDatabase.db";

      private var dbFile:File;
      private var createNewDB:Boolean = true;
      private var conn:SQLConnection;

      // ----- Event handling -----

      private function init():void
      {
        conn = new SQLConnection();
        dbFile = File.applicationStorageDirectory.resolvePath(dbFileName);
        if (dbFile.exists)
        {
          createNewDB = false;
          instructions.text = "Enter your database password to open the encrypted
database.";
          openButton.label = "Open Database";
        }
      }

      private function openConnection():void
      {
        var password:String = passwordInput.text;

        var keyGenerator:EncryptionKeyGenerator = new EncryptionKeyGenerator();

        if (password == null || password.length <= 0)
        {
          statusMsg.text = "Please specify a password.";
          return;
        }

        if (!keyGenerator.validateStrongPassword(password))
        {
          statusMsg.text = "The password must be 8-32 characters long. It must
contain at least one lowercase letter, at least one uppercase letter, and at least one number
or symbol.";
          return;
        }

        passwordInput.text = "";
        passwordInput.enabled = false;
        openButton.enabled = false;

        var encryptionKey:ByteArray = keyGenerator.getEncryptionKey(password);

        conn.addEventListener(SQLEvent.OPEN, openHandler);
        conn.addEventListener(SQLErrorEvent.ERROR, openError);
      }
    ]]>
  </mx:Script>
</mx:WindowedApplication>
</?xml>
```

```
        conn.openAsync(dbFile, SQLMode.CREATE, null, false, 1024, encryptionKey);
    }

    private function openHandler(event:SQLEvent):void
    {
        conn.removeEventListener(SQLEvent.OPEN, openHandler);
        conn.removeEventListener(SQLErrorEvent.ERROR, openError);

        statusMsg.setStyle("color", 0x009900);
        if (createNewDB)
        {
            statusMsg.text = "The encrypted database was created successfully.";
        }
        else
        {
            statusMsg.text = "The encrypted database was opened successfully.";
        }
    }

    private function openError(event:SQLErrorEvent):void
    {
        conn.removeEventListener(SQLEvent.OPEN, openHandler);
        conn.removeEventListener(SQLErrorEvent.ERROR, openError);

        if (!createNewDB && event.error.errorID ==
EncryptionKeyGenerator.ENCRYPTED_DB_PASSWORD_ERROR_ID)
        {
            statusMsg.text = "Incorrect password!";
        }
        else
        {
            statusMsg.text = "Error creating or opening database.";
        }
    }
    ]]>
</mx:Script>
<mx:Text id="instructions" text="Enter a password to create an encrypted database. The next
time you open the application, you will need to re-enter the password to open the database
again." width="75%" height="65"/>
<mx:HBox>
    <mx:TextInput id="passwordInput" displayAsPassword="true"/>
    <mx:Button id="openButton" label="Create Database" click="openConnection();"/>
</mx:HBox>
<mx:Text id="statusMsg" color="#990000" width="75%"/>
</mx:WindowedApplication>
```

Flash Professional example

The application FLA file contains the source code for a simple application that creates or opens a connection to an encrypted database. The FLA file has four components placed on the stage:

Instance name	Component type	Description
instructions	Label	Contains the instructions given to the user
passwordInput	TextInput	Input field where the user enters the password
openButton	Button	Button the user clicks after entering the password
statusMsg	Label	Displays status (success or failure) messages

The code for the application is defined on a keyframe on frame 1 of the main timeline. The following is the code for the application:

```
import com.adobe.air.crypto.EncryptionKeyGenerator;

const dbFileName:String = "encryptedDatabase.db";

var dbFile:File;
var createNewDB:Boolean = true;
var conn:SQLConnection;

init();

// ----- Event handling -----

function init():void
{
    passwordInput.displayAsPassword = true;
    openButton.addEventListener(MouseEvent.CLICK, openConnection);
    statusMsg.setStyle("textFormat", new TextFormat(null, null, 0x990000));

    conn = new SQLConnection();
    dbFile = File.applicationStorageDirectory.resolvePath(dbFileName);

    if (dbFile.exists)
    {
        createNewDB = false;
        instructions.text = "Enter your database password to open the encrypted database.";
        openButton.label = "Open Database";
    }
    else
    {
        instructions.text = "Enter a password to create an encrypted database. The next time
you open the application, you will need to re-enter the password to open the database again.";
        openButton.label = "Create Database";
    }
}

function openConnection(event:MouseEvent):void
{
    var keyGenerator:EncryptionKeyGenerator = new EncryptionKeyGenerator();

    var password:String = passwordInput.text;

    if (password == null || password.length <= 0)
    {
        statusMsg.text = "Please specify a password.";
        return;
    }
}
```

```
    }

    if (!keyGenerator.validateStrongPassword(password))
    {
        statusMsg.text = "The password must be 8-32 characters long. It must contain at least
one lowercase letter, at least one uppercase letter, and at least one number or symbol.";
        return;
    }

    passwordInput.text = "";
    passwordInput.enabled = false;
    openButton.enabled = false;

    var encryptionKey:ByteArray = keyGenerator.getEncryptionKey(password);

    conn.addEventListener(SQLEvent.OPEN, openHandler);
    conn.addEventListener(SQLErrorEvent.ERROR, openError);

    conn.openAsync(dbFile, SQLMode.CREATE, null, false, 1024, encryptionKey);
}

function openHandler(event:SQLEvent):void
{
    conn.removeEventListener(SQLEvent.OPEN, openHandler);
    conn.removeEventListener(SQLErrorEvent.ERROR, openError);

    statusMsg.setStyle("textFormat", new TextFormat(null, null, 0x009900));
    if (createNewDB)
    {
        statusMsg.text = "The encrypted database was created successfully.";
    }
    else
    {
        statusMsg.text = "The encrypted database was opened successfully.";
    }
}

function openError(event:SQLErrorEvent):void
{
    conn.removeEventListener(SQLEvent.OPEN, openHandler);
    conn.removeEventListener(SQLErrorEvent.ERROR, openError);

    if (!createNewDB && event.error.errorID ==
EncryptionKeyGenerator.ENCRYPTED_DB_PASSWORD_ERROR_ID)
    {
        statusMsg.text = "Incorrect password!";
    }
    else
    {
        statusMsg.text = "Error creating or opening database.";
    }
}
```

The application HTML file contains the source code for a simple application that creates or opens a connection to an encrypted database:

```
<html>
  <head>
    <title>Encrypted Database Example (HTML)</title>
    <style type="text/css">
      body
      {
        padding-top: 25px;
        font-family: Verdana, Arial;
        font-size: 14px;
      }
      div
      {
        width: 85%;
        margin-left: auto;
        margin-right: auto;
      }
      .error {color: #990000}
      .success {color: #009900}
    </style>

    <script type="text/javascript" src="AIRAliases.js"></script>
    <script type="application/x-shockwave-flash" src="EncryptionKeyGenerator.swf"/>
    <script type="text/javascript">
      // set up the class shortcut
      var ekg;
      if (window.runtime)
      {
        if (!ekg) ekg = {};
        ekg.EncryptionKeyGenerator =
window.runtime.com.adobe.air.crypto.EncryptionKeyGenerator;
      }

      // app globals
      var dbFileName = "encryptedDatabase.db";
      var dbFile;
      var createNewDB = true;
      var conn;

      // UI elements
      var instructions;
      var passwordInput;
      var openButton;
      var statusMsg;

      function init()
      {
        // UI elements
        instructions = document.getElementById("instructions");
        passwordInput = document.getElementById("passwordInput");
        openButton = document.getElementById("openButton");
        statusMsg = document.getElementById("statusMsg");

        conn = new air.SQLConnection();
        dbFile = air.File.applicationStorageDirectory.resolvePath(dbFileName);
        if (dbFile.exists)
        {
          createNewDB = false;

```

```
        instructions.innerHTML = "<p>Enter your database password to open the
encrypted database.</p>";
        openButton.value = "Open Database";
    }
}

function openConnection()
{
    var keyGenerator = new ekg.EncryptionKeyGenerator();

    var password = passwordInput.value;

    if (password == null || password.length <= 0)
    {
        statusMsg.innerHTML = "<p class='error'>Please specify a password.</p>";
        return;
    }

    if (!keyGenerator.validateStrongPassword(password))
    {
        statusMsg.innerHTML = "<p class='error'>The password must be 8-32
characters long. It must contain at least one lowercase letter, at least one uppercase letter,
and at least one number or symbol.</p>";
        return;
    }

    passwordInput.value = "";
    passwordInput.disabled = true;
    openButton.disabled = true;
    statusMsg.innerHTML = "";

    var encryptionKey = keyGenerator.getEncryptionKey(password);

    conn.addEventListener(air.SQLEvent.OPEN, openHandler);
    conn.addEventListener(air.SQLErrorEvent.ERROR, openError);

    conn.openAsync(dbFile, air.SQLMode.CREATE, null, false, 1024, encryptionKey);
}

function openHandler(event)
{
    conn.removeEventListener(air.SQLEvent.OPEN, openHandler);
    conn.removeEventListener(air.SQLErrorEvent.ERROR, openError);

    if (createNewDB)
    {
        statusMsg.innerHTML = "<p class='success'>The encrypted database was
created successfully.</p>";
    }
    else
    {
        statusMsg.innerHTML = "<p class='success'>The encrypted database was
opened successfully.</p>";
    }
}

function openError(event)
```



```
        {
            conn.removeEventListener(air.SQLEvent.OPEN, openHandler);
            conn.removeEventListener(air.SQLErrorEvent.ERROR, openError);

            if (!createNewDB && event.error.errorID ==
ekg.EncryptionKeyGenerator.ENCRYPTED_DB_PASSWORD_ERROR_ID)
            {
                statusMsg.innerHTML = "<p class='error'>Incorrect password!</p>";
            }
            else
            {
                statusMsg.innerHTML = "<p class='error'>Error creating or opening
database.</p>";
            }
        }
    }
</script>
</head>

<body onload="init();">
    <div id="instructions"><p>Enter a password to create an encrypted database. The next
time you open the application, you will need to re-enter the password to open the database
again.</p></div>
    <div><input id="passwordInput" type="password"/><input id="openButton" type="button"
value="Create Database" onclick="openConnection();" /></div>
    <div id="statusMsg"></div>
</body>
</html>
```

Understanding the EncryptionKeyGenerator class

Adobe AIR 1.5 and later

It isn't necessary to understand the inner workings of the EncryptionKeyGenerator class to use it to create a secure encryption key for your application database. The process for using the class is explained in [“Using the EncryptionKeyGenerator class to obtain a secure encryption key”](#) on page 242. However, you might find it valuable to understand the techniques that the class uses. For example, you might want to adapt the class or incorporate some of its techniques for situations where a different level of data privacy is desired.

The EncryptionKeyGenerator class is included in the open-source ActionScript 3.0 core library (as3corelib) project. You can download [the as3corelib package including source code and documentation](#). You can also view the source code on the project site or download it to follow along with the explanations.

When code creates an EncryptionKeyGenerator instance and calls its `getEncryptionKey()` method, several steps are taken to ensure that only the rightful user can access the data. The process is the same to generate an encryption key from a user-entered password before the database is created as well as to re-create the encryption key to open the database.

Obtain and validate a strong password

Adobe AIR 1.5 and later

When code calls the `getEncryptionKey()` method, it passes in a password as a parameter. The password is used as the basis for the encryption key. By using a piece of information that only the user knows, this design ensures that only the user who knows the password can access the data in the database. Even if an attacker accesses the user's account on the computer, the attacker can't get into the database without knowing the password. For maximum security, the application never stores the password.

An application's code creates an `EncryptionKeyGenerator` instance and calls its `getEncryptionKey()` method, passing a user-entered password as an argument (the variable `password` in this example):

```
var keyGenerator = new ekg.EncryptionKeyGenerator();  
var encryptionKey = keyGenerator.getEncryptionKey(password);
```

The first step the `EncryptionKeyGenerator` class takes when the `getEncryptionKey()` method is called is to check the user-entered password to ensure that it meets the password strength requirements. The `EncryptionKeyGenerator` class requires a password to be 8 - 32 characters long. The password must contain a mix of uppercase and lowercase letters and at least one number or symbol character.

Internally the `getEncryptionKey()` method calls the `EncryptionKeyGenerator` class's `validateStrongPassword()` method and, if the password isn't valid, throws an exception. The `validateStrongPassword()` method is a public method so that application code can check a password without calling the `getEncryptionKey()` method to avoid causing an error.

Expand the password to 256 bits

Adobe AIR 1.5 and later

Later in the process, the password is required to be 256 bits long. Rather than require each user to enter a password that's exactly 256 bits (32 characters) long, the code creates a longer password by repeating the password characters.

The following is the code for the `concatenatePassword()` method:

If the password is less than 256 bits, the code concatenates the password with itself to make it 256 bits. If the length doesn't work out exactly, the last repetition is shortened to get exactly 256 bits.

Generate or retrieve a 256-bit salt value

Adobe AIR 1.5 and later

The next step is to get a 256-bit salt value that in a later step is combined with the password. A *salt* is a random value that is added to or combined with a user-entered value to form a password. Using a salt with a password ensures that even if a user chooses a real word or common term as a password, the password-plus-salt combination that the system uses is a random value. This randomness helps guard against a dictionary attack, where an attacker uses a list of words to attempt to guess a password. In addition, by generating the salt value and storing it in the encrypted local store, it is tied to the user's account on the machine on which the database file is located.

If the application is calling the `getEncryptionKey()` method for the first time, the code creates a random 256-bit salt value. Otherwise, the code loads the salt value from the encrypted local store.

Combine the 256-bit password and salt using the XOR operator

Adobe AIR 1.5 and later

The code now has a 256-bit password and a 256-bit salt value. It next uses a bitwise XOR operation to combine the salt and the concatenated password into a single value. In effect, this technique creates a 256-bit password consisting of characters from the entire range of possible characters. This principle is true even though most likely the actual password input consists of primarily alphanumeric characters. This increased randomness provides the benefit of making the set of possible passwords large without requiring the user to enter a long complex password.

Hash the key

Adobe AIR 1.5 and later

Once the concatenated password and the salt have been combined, the next step is to further secure this value by hashing it using the SHA-256 hashing algorithm. Hashing the value makes it more difficult for an attacker to reverse-engineer it.

Extract the encryption key from the hash

Adobe AIR 1.5 and later

The encryption key must be a ByteArray that is exactly 16 bytes (128 bits) long. The result of the SHA-256 hashing algorithm is always 256 bits long. Consequently, the final step is to select 128 bits from the hashed result to use as the actual encryption key.

It isn't necessary to use the first 128 bits as the encryption key. You could select a range of bits starting at some arbitrary point, you could select every other bit, or use some other way of selecting bits. The important thing is that the code selects 128 distinct bits, and that the same 128 bits are used each time.

Strategies for working with SQL databases

Adobe AIR 1.0 and later

There are various ways that an application can access and work with a local SQL database. The application design can vary in terms of how the application code is organized, the sequence and timing of how operations are performed, and so on. The techniques you choose can have an impact on how easy it is to develop your application. They can affect how easy it is to modify the application in future updates. They can also affect how well the application performs from the users' perspective.

Distributing a pre-populated database

Adobe AIR 1.0 and later

When you use an AIR local SQL database in your application, the application expects a database with a certain structure of tables, columns, and so forth. Some applications also expect certain data to be pre-populated in the database file. One way to ensure that the database has the proper structure is to create the database within the application code. When the application loads it checks for the existence of its database file in a particular location. If the file doesn't exist, the application executes a set of commands to create the database file, create the database structure, and populate the tables with the initial data.

The code that creates the database and its tables is frequently complex. It is often only used once in the installed lifetime of the application, but still adds to the size and complexity of the application. As an alternative to creating the database, structure, and data programmatically, you can distribute a pre-populated database with your application. To distribute a predefined database, include the database file in the application's AIR package.

Like all files that are included in an AIR package, a bundled database file is installed in the application directory (the directory represented by the `File.applicationDirectory` property). However, files in that directory are read only. Use the file from the AIR package as a "template" database. The first time a user runs the application, copy the original database file into the user's "[Pointing to the application storage directory](#)" on page 149 (or another location), and use that database within the application.

Best practices for working with local SQL databases

Adobe AIR 1.0 and later

The following list is a set of suggested techniques you can use to improve the performance, security, and ease of maintenance of your applications when working with local SQL databases.

Pre-create database connections

Adobe AIR 1.0 and later

Even if your application doesn't execute any statements when it first loads, instantiate a `SQLConnection` object and call its `open()` or `openAsync()` method ahead of time (such as after the initial application startup) to avoid delays when running statements. See "[Connecting to a database](#)" on page 213.

Reuse database connections

Adobe AIR 1.0 and later

If you access a certain database throughout the execution time of your application, keep a reference to the `SQLConnection` instance, and reuse it throughout the application, rather than closing and reopening the connection. See "[Connecting to a database](#)" on page 213.

Favor asynchronous execution mode

Adobe AIR 1.0 and later

When writing data-access code, it can be tempting to execute operations synchronously rather than asynchronously, because using synchronous operations frequently requires shorter and less complex code. However, as described in "[Using synchronous and asynchronous database operations](#)" on page 232, synchronous operations can have a performance impact that is obvious to users and detrimental to their experience with an application. The amount of time a single operation takes varies according to the operation and particularly the amount of data it involves. For instance, a SQL `INSERT` statement that only adds a single row to the database takes less time than a `SELECT` statement that retrieves thousands of rows of data. However, when you're using synchronous execution to perform multiple operations, the operations are usually strung together. Even if the time each single operation takes is very short, the application is frozen until all the synchronous operations finish. As a result, the cumulative time of multiple operations strung together may be enough to stall your application.

Use asynchronous operations as a standard approach, especially with operations that involve large numbers of rows. There is a technique for dividing up the processing of large sets of `SELECT` statement results, described in [“Retrieving SELECT results in parts”](#) on page 224. However, this technique can only be used in asynchronous execution mode. Only use synchronous operations when you can't achieve certain functionality using asynchronous programming, when you've considered the performance trade-off that your application's users will face, and when you've tested your application so that you know how your application's performance is affected. Using asynchronous execution can involve more complex coding. However, remember that you only have to write the code once, but the application's users have to use it repeatedly, fast or slow.

In many cases, by using a separate `SQLStatement` instance for each SQL statement to be executed, multiple SQL operations can be queued up at one time, which makes asynchronous code like synchronous code in terms of how the code is written. For more information, see [“Understanding the asynchronous execution model”](#) on page 236.

Use separate SQL statements and don't change the `SQLStatement`'s `text` property

Adobe AIR 1.0 and later

For any SQL statement that is executed more than once in an application, create a separate `SQLStatement` instance for each SQL statement. Use that `SQLStatement` instance each time that SQL command executes. For example, suppose you are building an application that includes four different SQL operations that are performed multiple times. In that case, create four separate `SQLStatement` instances and call each statement's `execute()` method to run it. Avoid the alternative of using a single `SQLStatement` instance for all SQL statements, redefining its `text` property each time before executing the statement.

Use statement parameters

Adobe AIR 1.0 and later

Use `SQLStatement` parameters—never concatenate user input into statement text. Using parameters makes your application more secure because it prevents the possibility of SQL injection attacks. It makes it possible to use objects in queries (rather than only SQL literal values). It also makes statements run more efficiently because they can be reused without needing to be recompiled each time they're executed. See [“Using parameters in statements”](#) on page 216 for more information.

Chapter 15: Encrypted local storage

The Adobe® AIR® runtime provides a persistent encrypted local store (ELS) for each AIR application installed on a user's computer. This lets you save and retrieve data that is stored on the user's local hard drive in an encrypted format that cannot easily be deciphered by other users. A separate encrypted local store is used for each AIR application, and each AIR application uses a separate encrypted local store for each user.

Note: In addition to the encrypted local store, AIR also provides encryption for content stored in SQL databases. For details, see [“Using encryption with SQL databases”](#) on page 237.

You may want to use the encrypted local store to cache information that must be secured, such as login credentials for web services. The ELS is appropriate for storing information that must be kept private from other users. It does not, however, protect the data from other processes run under the same user account. It is thus not appropriate for protecting secret application data, such as DRM or encryption keys.

On desktop platforms, AIR uses DPAPI on Windows, KeyChain on Mac OS and iOS, and KeyRing or KWallet on Linux to associate the encrypted local store to each application and user. The encrypted local store uses AES-CBC 128-bit encryption.

On Android, the data stored by the EncryptedLocalStorage class are not encrypted. Instead the data is protected by the user-level security provided by the operating system. The Android operating system assigns every application a separate user ID. Applications can only access their own files and files created in public locations (such as the removable storage card). Note that on “rooted” Android devices, applications running with root privileges CAN access the files of other applications. Thus on a rooted device, the encrypted local store does not provide as high a level of data protection as it does on a non-rooted device.

Information in the encrypted local store is only available to AIR application content in the application security sandbox.

If you update an AIR application, the updated version retains access to any existing data in the encrypted local store unless:

- The items were added with the `stronglyBound` parameter set to `true`
- The existing and update versions are both published prior to AIR 1.5.3 and the update is signed with a migration signature.

Limitations of the encrypted local store

The data in the encrypted local store is protected by the user's operating system account credentials. Other entities cannot access the data in the store unless they can login as that user. However, the data is not secure against access by other applications run by an authenticated user.

Because the user must be authenticated for these attacks to work, the user's private data is still protected (unless the user account itself is compromised). However, data that your application may want to keep secret from users, such as keys used for licensing or digital rights management, is not secure. Thus the ELS is not an appropriate location for storing such information. It is only an appropriate place for storing a user's private data, such as passwords.

Data in the ELS can be lost for a variety of reasons. For example, the user could uninstall the application and delete the encrypted file. Or, the publisher ID could be changed as a result of an update. Thus the ELS should be treated as a private cache, not a permanent data storage.

The `stronglyBound` parameter is deprecated and should not be set to `true`. Setting the parameter to `true` does not provide any additional protection for data. At the same time, access to the data is lost whenever the application is updated — even if the publisher ID stays the same.

The encrypted local store may perform more slowly if the stored data exceeds 10MB.

When you uninstall an AIR application, the uninstaller does not delete data stored in the encrypted local store.

The best practices for using the ELS include:

- Use the ELS to store sensitive user data such as passwords (setting `stronglyBound` to `false`)
- Do not use the ELS to store applications secrets such as DRM keys or licensing tokens.
- Provide a way for your application to recreate the data stored in the ELS if the ELS data is lost. For example, by prompting the user to re-enter their account credentials when necessary.
- Do not use the `stronglyBound` parameter.
- If you do set `stronglyBound` to `true`, do not migrate stored items during an update. Recreate the data after the update instead.
- Only store relatively small amounts of data. For larger amounts of data, use an AIR SQL database with encryption.

More Help topics

[flash.data.EncryptedLocalStore](#)

Adding data to the encrypted local store

Use the `setItem()` static method of the `EncryptedLocalStore` class to store data in the local store. The data is stored in a hash table, using strings as keys, with the data stored as byte arrays.

For example, the following code stores a string in the encrypted local store:

```
var str = "Bob";
var bytes = new air.ByteArray();
bytes.writeUTFBytes(str);
air.EncryptedLocalStore.setItem("firstName", bytes);
```

The third parameter of the `setItem()` method, the `stronglyBound` parameter, is optional. When this parameter is set to `true`, the encrypted local store binds the stored item to the storing AIR application's digital signature and bits:

```
var str = "Bob";
var bytes = new air.ByteArray();
bytes.writeUTFBytes(str);
air.EncryptedLocalStore.setItem("firstName", bytes, false);
```

For an item that is stored with `stronglyBound` set to `true`, subsequent calls to `getItem()` only succeed if the calling AIR application is identical to the storing application (if no data in files in the application directory have changed). If the calling AIR application is different from the storing application, the application throws an `Error` exception when you call `getItem()` for a strongly bound item. If you update your application, it will not be able to read strongly bound data previously written to the encrypted local store. Setting `stronglyBound` to `true` on mobile devices is ignored; the parameter is always treated as `false`.

If the `stronglyBound` parameter is set to `false` (the default), only the publisher ID needs to stay the same for the application to read the data. The bits of the application may change (and they need to be signed by the same publisher), but they do not need to be the exact same bits as were in application that stored the data. Updated applications with the same publisher ID as the original can continue to access the data.

Note: In practice, setting `stronglyBound` to `true` does not add any additional data protection. A “malicious” user could still alter an application to gain access to items stored in the ELS. Furthermore, data is protected from external, non-user threats just as strongly whether `stronglyBound` is set to `true` or `false`. For these reasons, setting `stronglyBound` to `true` is discouraged.

Accessing data in the encrypted local store

Adobe AIR 1.0 and later

You can retrieve a value from the encrypted local store by using the `EncryptedLocalStore.getItem()` method, as in the following example:

```
var storedValue = air.EncryptedLocalStore.getItem("firstName");  
air.trace(storedValue.readUTFBytes(storedValue.length)); // "foo"
```

Removing data from the encrypted local store

Adobe AIR 1.0 and later

You can delete a value from the encrypted local store by using the `EncryptedLocalStore.removeItem()` method, as in the following example:

```
air.EncryptedLocalStore.removeItem("firstName");
```

You can clear all data from the encrypted local store by calling the `EncryptedLocalStore.reset()` method, as in the following example:

```
air.EncryptedLocalStore.reset();
```


Chapter 16: Working with byte arrays

Flash Player 9 and later, Adobe AIR 1.0 and later

The `ByteArray` class allows you to read from and write to a binary stream of data, which is essentially an array of bytes. This class provides a way to access data at the most elemental level. Because computer data consists of bytes, or groups of 8 bits, the ability to read data in bytes means that you can access data for which classes and access methods do not exist. The `ByteArray` class allows you to parse any stream of data, from a bitmap to a stream of data traveling over the network, at the byte level.

The `writeObject()` method allows you to write an object in serialized Action Message Format (AMF) to a `ByteArray`, while the `readObject()` method allows you to read a serialized object from a `ByteArray` to a variable of the original data type. You can serialize any object except for display objects, which are those objects that can be placed on the display list. You can also assign serialized objects back to custom class instances if the custom class is available to the runtime. After converting an object to AMF, you can efficiently transfer it over a network connection or save it to a file.

The sample Adobe® AIR® application described here reads a .zip file as an example of processing a byte stream, extracting a list of the files that the .zip file contains and writing them to the desktop.

More Help topics

[flash.utils.ByteArray](#)

[flash.utils.IExternalizable](#)

[Action Message Format specification](#)

Reading and writing a ByteArray

Flash Player 9 and later, Adobe AIR 1.0 and later

The `ByteArray` class is part of the `flash.utils` package; you can also use the alias `air.ByteArray` to refer to the `ByteArray` class if your code includes the `AIRAliases.js` file. To create a `ByteArray`, invoke the `ByteArray` constructor as shown in the following example:

```
var stream = new air.ByteArray();
```

ByteArray methods

Flash Player 9 and later, Adobe AIR 1.0 and later

Any meaningful data stream is organized into a format that you can analyze to find the information that you want. A record in a simple employee file, for example, would probably include an ID number, a name, an address, a phone number, and so on. An MP3 audio file contains an ID3 tag that identifies the title, author, album, publishing date, and genre of the file that's being downloaded. The format allows you to know the order in which to expect the data on the data stream. It allows you to read the byte stream intelligently.

The `ByteArray` class includes several methods that make it easier to read from and write to a data stream. Some of these methods include `readBytes()` and `writeBytes()`, `readInt()` and `writeInt()`, `readFloat()` and `writeFloat()`, `readObject()` and `writeObject()`, and `readUTFBytes()` and `writeUTFBytes()`. These methods enable you to read data from the data stream into variables of specific data types and write from specific data types directly to the binary data stream.

For example, the following code reads a simple array of strings and floating-point numbers and writes each element to a `ByteArray`. The organization of the array allows the code to call the appropriate `ByteArray` methods (`writeUTFBytes()` and `writeFloat()`) to write the data. The repeating data pattern makes it possible to read the array with a loop.

```
// The following example reads a simple Array (groceries), made up of strings
// and floating-point numbers, and writes it to a ByteArray.

// define the grocery list Array
var groceries = ["milk", 4.50, "soup", 1.79, "eggs", 3.19, "bread", 2.35]
// define the ByteArray
var bytes = new air.ByteArray();
// for each item in the array
for (i = 0; i < groceries.length; i++) {
    bytes.writeUTFBytes(groceries[i]); //write the string and position to the next item
    bytes.writeFloat(groceries[i]); // write the float
    air.trace("bytes.position is: " + bytes.position); //display the position in ByteArray
}
air.trace("bytes length is: " + bytes.length); // display the length
```

The position property

Flash Player 9 and later, Adobe AIR 1.0 and later

The position property stores the current position of the pointer that indexes the `ByteArray` during reading or writing. The initial value of the position property is 0 (zero) as shown in the following code:

```
var bytes = new air.ByteArray();
air.trace("bytes.position is initially: " + bytes.position); // 0
```

When you read from or write to a `ByteArray`, the method that you use updates the position property to point to the location immediately following the last byte that was read or written. For example, the following code writes a string to a `ByteArray` and afterward the position property points to the byte immediately following the string in the `ByteArray`:

```
var bytes = new air.ByteArray();
air.trace("bytes.position is initially: " + bytes.position); // 0
bytes.writeUTFBytes("Hello World!");
air.trace("bytes.position is now: " + bytes.position); // 12
```

Likewise, a read operation increments the position property by the number of bytes read.

```
var bytes = new air.ByteArray();

air.trace("bytes.position is initially: " + bytes.position); // 0
bytes.writeUTFBytes("Hello World!");
air.trace("bytes.position is now: " + bytes.position); // 12
bytes.position = 0;
air.trace("The first 6 bytes are: " + (bytes.readUTFBytes(6))); //Hello
air.trace("And the next 6 bytes are: " + (bytes.readUTFBytes(6))); // World!
```

Notice that you can set the position property to a specific location in the ByteArray to read or write at that offset.

The bytesAvailable and length properties

Flash Player 9 and later, Adobe AIR 1.0 and later

The `length` and `bytesAvailable` properties tell you how long a ByteArray is and how many bytes remain in it from the current position to the end. The following example illustrates how you can use these properties. The example writes a String of text to the ByteArray and then reads the ByteArray one byte at a time until it encounters either the character "a" or the end (`bytesAvailable <= 0`).

```
var bytes = new air.ByteArray();
var text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus etc.";

bytes.writeUTFBytes(text); // write the text to the ByteArray
air.trace("The length of the ByteArray is: " + bytes.length); // 70
bytes.position = 0; // reset position
while (bytes.bytesAvailable > 0 && (bytes.readUTFBytes(1) != 'a')) {
    //read to letter a or end of bytes
}
if (bytes.position < bytes.bytesAvailable) {
    air.trace("Found the letter a; position is: " + bytes.position); // 23
    air.trace("and the number of bytes available is: " + bytes.bytesAvailable); // 47
}
```

The endian property

Flash Player 9 and later, Adobe AIR 1.0 and later

Computers can differ in how they store multibyte numbers, that is, numbers that require more than 1 byte of memory to store them. An integer, for example, can take 4 bytes, or 32 bits, of memory. Some computers store the most significant byte of the number first, in the lowest memory address, and others store the least significant byte first. This attribute of a computer, or of byte ordering, is referred to as being either *big endian* (most significant byte first) or *little endian* (least significant byte first). For example, the number 0x31323334 would be stored as follows for big endian and little endian byte ordering, where a0 represents the lowest memory address of the 4 bytes and a3 represents the highest:

Big Endian	Big Endian	Big Endian	Big Endian
a0	a1	a2	a3
31	32	33	34

Little Endian	Little Endian	Little Endian	Little Endian
a0	a1	a2	a3
34	33	32	31

The `endian` property of the ByteArray class allows you to denote this byte order for multibyte numbers that you are processing. The acceptable values for this property are either "bigEndian" or "littleEndian" and the Endian class defines the constants `BIG_ENDIAN` and `LITTLE_ENDIAN` for setting the `endian` property with these strings.

The compress() and uncompress() methods

Flash Player 9 and later, Adobe AIR 1.0 and later

The `compress()` method allows you to compress a `ByteArray` in accordance with a compression algorithm that you specify as a parameter. The `uncompress()` method allows you to uncompress a compressed `ByteArray` in accordance with a compression algorithm. After calling `compress()` and `uncompress()`, the length of the byte array is set to the new length and the position property is set to the end.

The `CompressionAlgorithm` class (AIR) defines constants that you can use to specify the compression algorithm. The `ByteArray` class supports both the deflate (AIR-only) and zlib algorithms. The deflate compression algorithm is used in several compression formats, such as zlib, gzip, and some zip implementations. The zlib compressed data format is described at <http://www.ietf.org/rfc/rfc1950.txt> and the deflate compression algorithm is described at <http://www.ietf.org/rfc/rfc1951.txt>.

The following example compresses a `ByteArray` called `bytes` using the deflate algorithm:

```
bytes.compress(air.CompressionAlgorithm.DEFLATE);
```

The following example uncompresses a compressed `ByteArray` using the deflate algorithm:

```
bytes.uncompress(CompressionAlgorithm.DEFLATE);
```

Reading and writing objects

Flash Player 9 and later, Adobe AIR 1.0 and later

The `readObject()` and `writeObject()` methods read an object from and write an object to a `ByteArray`, encoded in serialized Action Message Format (AMF). AMF is a proprietary message protocol created by Adobe and used by various ActionScript 3.0 classes, including `Netstream`, `NetConnection`, `NetStream`, `LocalConnection`, and `Shared Objects`.

A one-byte type marker describes the type of the encoded data that follows. AMF uses the following 13 data types:

```
value-type = undefined-marker | null-marker | false-marker | true-marker | integer-type |  
            double-type | string-type | xml-doc-type | date-type | array-type | object-type |  
            xml-type | byte-array-type
```

The encoded data follows the type marker unless the marker represents a single possible value, such as null or true or false, in which case nothing else is encoded.

There are two versions of AMF: AMF0 and AMF3. AMF 0 supports sending complex objects by reference and allows endpoints to restore object relationships. AMF 3 improves AMF 0 by sending object traits and strings by reference, in addition to object references, and by supporting new data types that were introduced in ActionScript 3.0. The `ByteArray.objectEncoding` property specifies the version of AMF that is used to encode the object data. The `flash.net.ObjectEncoding` class defines constants for specifying the AMF version: `ObjectEncoding.AMF0` and `ObjectEncoding.AMF3`.

The following example calls `writeObject()` to write an XML object to a `ByteArray`, which it then writes to the `order` file on the desktop. The example displays the message “Wrote order file to desktop!” in the AIR window when it is finished.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"><html
xmlns="http://www.w3.org/1999/xhtml">
<head>
<style type="text/css">
    #taFiles
    {
        border: 1px solid black;
        font-family: Courier, monospace;
        white-space: pre;
        width: 95%;
        height: 95%;
        overflow-y: scroll;
    }
</style>
<script type="text/javascript" src="AIRAliases.js" ></script>
<script type="text/javascript">

//define ByteArray
var inBytes = new air.ByteArray();
//add objectEncoding value and file heading to output text
var output = "Object encoding is: " + inBytes.objectEncoding + "\n\n" + "order file: \n\n";

function init() {

    readFile("order", inBytes);
    inBytes.position = 0;//reset position to beginning
    // read XML from ByteArray
    var orderXML = inBytes.readObject();
    // convert to XML Document object
    var myXML = (new DOMParser()).parseFromString(orderXML, "text/xml");
    document.write(myXML.getElementsByTagName("menuName")[0].childNodes[0].nodeValue + ": ");
    document.write(myXML.getElementsByTagName("price")[0].childNodes[0].nodeValue +
"<br/>"); // burger: 3.95
    document.write(myXML.getElementsByTagName("menuName")[1].childNodes[0].nodeValue + ":
");
    document.write(myXML.getElementsByTagName("price")[1].childNodes[0].nodeValue +
"<br/>"); // fries: 1.45
} // end of init()

// read specified file into byte array
function readFile(fileName, data) {
    var inFile = air.File.desktopDirectory; // source folder is desktop
    inFile = inFile.resolvePath(fileName); // name of file to read
    var inStream = new air.FileStream();
    inStream.open(inFile, air.FileMode.READ);
    inStream.readBytes(data, 0, data.length);
    inStream.close();
}
</script>
</head>

<body onload = "init();">
    <div id="taFiles"></div>
</body>
</html>
```

The `readObject()` method reads an object in serialized AMF from a `ByteArray` and stores it in an object of the specified type. The following example reads the `order` file from the desktop into a `ByteArray` (`inBytes`) and calls `readObject()` to store it in `orderXML`, which it then converts to an XML object document, `myXML`, and displays the values of two item and price elements. The example also displays the value of the `objectEncoding` property along with a header for the contents of the `order` file.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<style type="text/css">
    #taFiles
    {
        border: 1px solid black;
        font-family: Courier, monospace;
        white-space: pre;
        width: 95%;
        height: 95%;
        overflow-y: scroll;
    }
</style>
<script type="text/javascript" src="AIRAliases.js" ></script>
<script type="text/javascript">

//define ByteArray
var inBytes = new air.ByteArray();
//add objectEncoding value and file heading to output text
var output = "Object encoding is: " + inBytes.objectEncoding + "<br/><br/>" + "order file
items:" + "<br/><br/>";

function init() {

    readFile("order", inBytes);
    inBytes.position = 0;//reset position to beginning
    // read XML from ByteArray
    var orderXML = inBytes.readObject();
    // convert to XML Document object
    var myXML = (new DOMParser()).parseFromString(orderXML, "text/xml");
    document.write(output);
    document.write(myXML.getElementsByTagName("menuName")[0].childNodes[0].nodeValue + ": ");
        document.write(myXML.getElementsByTagName("price")[0].childNodes[0].nodeValue +
"<br/>"); // burger: 3.95
    document.write(myXML.getElementsByTagName("menuName")[1].childNodes[0].nodeValue + "
");
    document.write(myXML.getElementsByTagName("price")[1].childNodes[0].nodeValue +
```

```
"<br/>");          // fries: 1.45
} // end of init()

// read specified file into byte array
function readFile(fileName, data) {
    var inFile = air.File.desktopDirectory; // source folder is desktop
    inFile = inFile.resolvePath(fileName); // name of file to read
    var inStream = new air.FileStream();
    inStream.open(inFile, air.FileMode.READ);
    inStream.readBytes(data, 0, data.length);
    inStream.close();
}
</script>
</head>

<body onload = "init();">
    <div id="taFiles"></div>
</body>
</html>
```

ByteArray example: Reading a .zip file

Adobe AIR 1.0 and later

This example demonstrates how to read a simple .zip file containing several files of different types. It does so by extracting relevant data from the metadata for each file, uncompressing each file into a ByteArray and writing the file to the desktop.

The general structure of a .zip file is based on the specification by PKWARE Inc., which is maintained at <http://www.pkware.com/documents/casestudies/APPNOTE.TXT>. First is a file header and file data for the first file in the .zip archive, followed by a file header and file data pair for each additional file. (The structure of the file header is described later.) Next, the .zip file optionally includes a data descriptor record (usually when the output zip file was created in memory rather than saved to a disk). Next are several additional optional elements: archive decryption header, archive extra data record, central directory structure, Zip64 end of central directory record, Zip64 end of central directory locator, and end of central directory record.

The code in this example is written to only parse zip files that do not contain folders and it does not expect data descriptor records. It ignores all information following the last file data.

The format of the file header for each file is as follows:

file header signature	4 bytes
required version	2 bytes
general-purpose bit flag	2 bytes
compression method	2 bytes (8=DEFLATE; 0=UNCOMPRESSED)
last modified file time	2 bytes
last modified file date	2 bytes
crc-32	4 bytes

compressed size	4 bytes
uncompressed size	4 bytes
file name length	2 bytes
extra field length	2 bytes
file name	variable
extra field	variable

Following the file header is the actual file data, which can be either compressed or uncompressed, depending on the compression method flag. The flag is 0 (zero) if the file data is uncompressed, 8 if the data is compressed using the DEFLATE algorithm, or another value for other compression algorithms.

The user interface for this example consists of a label and a text area (`taFiles`). The application writes the following information to the text area for each file it encounters in the `.zip` file: the file name, the compressed size, and the uncompressed size. The following MXML document defines the user interface for the Flex version of the application:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical"
creationComplete="init();" >
  <mx:Script>
    <![CDATA[
      // The application code goes here
    ]]>
  </mx:Script>
  <mx:Form>
    <mx:FormItem label="Output">
      <mx:TextArea id="taFiles" width="320" height="150"/>
    </mx:FormItem>
  </mx:Form>
</mx:WindowedApplication>
```

The user interface for this example consists of a label and a text area (`taFiles`). The application writes the following information to the text area for each file it encounters in the `.zip` file: the file name, the compressed size, and the uncompressed size. The following HTML page defines the user interface for the application:


```
<html>
  <head>
    <style type="text/css">
      #taFiles
      {
        border: 1px solid black;
        font-family: Courier, monospace;
        white-space: pre;
        width: 95%;
        height: 95%;
        overflow-y: scroll;
      }
    </style>
    <script type="text/javascript" src="AIRAliases.js"></script>
    <script type="text/javascript">
      // The application code goes here
    </script>
  </head>
  <body onload="init();" >
    <div id="taFiles"></div>
  </body>
</html>
```

The beginning of the program performs the following tasks:

- Defines the bytes ByteArray

```
var bytes = new air.ByteArray();
```

- Defines variables to store metadata from the file header

```
// variables for reading fixed portion of file header
var fileName = new String();
var flNameLength;
var xfldLength;
var offset;
var compSize;
var uncompSize;
var compMethod;
var signature;
```

```
var output;
```

- Defines File (zfile) and FileStream (zStream) objects to represent the .zip file, and specifies the location of the .zip file from which the files are extracted—a file named “HelloAIR.zip” in the desktop directory.

```
// File variables for accessing .zip file
var zfile = air.File.desktopDirectory.resolvePath("HelloAIR.zip");
var zStream = new air.FileStream();
```

In Flex, the program code starts in the `init()` method, which is called as the `creationComplete` handler for the root `mx:WindowedApplication` tag.

The program code starts in the `init()` method, which is called as the `onload` event handler for the `body` tag.

```
function init()
{
```

The program begins by opening the .zip file in READ mode.

```
zStream.open(zfile, air.FileMode.READ);
```

It then sets the `endian` property of `bytes` to `LITTLE_ENDIAN` to indicate that the byte order of numeric fields has the least significant byte first.

```
bytes.endian = air.Endian.LITTLE_ENDIAN;
```

Next, a `while()` statement begins a loop that continues until the current position in the file stream is greater than or equal to the size of the file.

```
while (zStream.position < zfile.size)
{
```

The first statement inside the loop reads the first 30 bytes of the file stream into the `ByteArray bytes`. The first 30 bytes make up the fixed-size part of the first file header.

```
    // read fixed metadata portion of local file header
    zStream.readBytes(bytes, 0, 30);
```

Next, the code reads an integer (`signature`) from the first bytes of the 30-byte header. The ZIP format definition specifies that the signature for every file header is the hexadecimal value `0x04034b50`; if the signature is different it means that the code has moved beyond the file portion of the `.zip` file and there are no more files to extract. In that case the code exits the `while` loop immediately rather than waiting for the end of the byte array.

```
    bytes.position = 0;
    signature = bytes.readInt();
    // if no longer reading data files, quit
    if (signature != 0x04034b50)
    {
        break;
    }
```

The next part of the code reads the header byte at offset position 8 and stores the value in the variable `compMethod`. This byte contains a value indicating the compression method that was used to compress this file. Several compression methods are allowed, but in practice nearly all `.zip` files use the DEFLATE compression algorithm. If the current file is compressed with DEFLATE compression, `compMethod` is 8; if the file is uncompressed, `compMethod` is 0.

```
    bytes.position = 8;
    compMethod = bytes.readByte(); // store compression method (8 == Deflate)
```

Following the first 30 bytes is a variable-length portion of the header that contains the file name and, possibly, an extra field. The variable `offset` stores the size of this portion. The size is calculated by adding the file name length and extra field length, read from the header at offsets 26 and 28.

```
    offset = 0; // stores length of variable portion of metadata
    bytes.position = 26; // offset to file name length
    flNameLength = bytes.readShort(); // store file name
    offset += flNameLength; // add length of file name
    bytes.position = 28; // offset to extra field length
    xfldLength = bytes.readShort();
    offset += xfldLength; // add length of extra field
```

Next the program reads the variable-length portion of the file header for the number of bytes stored in the `offset` variable.

```
    // read variable length bytes between fixed-length header and compressed file data
    zStream.readBytes(bytes, 30, offset);
```

The program reads the file name from the variable length portion of the header and displays it in the text area along with the compressed (zipped) and uncompressed (original) sizes of the file.

Working with byte arrays

```

bytes.position = 30;
fileName = bytes.readUTFBytes(fileNameLength); // read file name
output += fileName + "<br />"; // write file name to text area
bytes.position = 18;
compSize = bytes.readUnsignedInt(); // store size of compressed portion
output += "\tCompressed size is: " + compSize + '<br />';
bytes.position = 22; // offset to uncompressed size
uncompSize = bytes.readUnsignedInt(); // store uncompressed size
output += "\tUncompressed size is: " + uncompSize + '<br />';

```

The example reads the rest of the file from the file stream into `bytes` for the length specified by the compressed size, overwriting the file header in the first 30 bytes. The compressed size is accurate even if the file is not compressed because in that case the compressed size is equal to the uncompressed size of the file.

```

// read compressed file to offset 0 of bytes; for uncompressed files
// the compressed and uncompressed size is the same
if (compSize == 0) continue;
zStream.readBytes(bytes, 0, compSize);

```

Next, the example uncompresses the compressed file and calls the `outfile()` function to write it to the output file stream. It passes `outfile()` the file name and the byte array containing the file data.

```

if (compMethod == 8) // if file is compressed, uncompress
{
    bytes.uncompress(air.CompressionAlgorithm.DEFLATE);
}
outfile(fileName, bytes); // call outfile() to write out the file

```

The closing braces indicate the end of the `while` loop and of the `init()` method and the application code, except for the `outfile()` method. Execution loops back to the beginning of the `while` loop and continues processing the next bytes in the `.zip` file—either extracting another file or ending processing of the `.zip` file if the last file has been processed. When all the files have been processed, the example writes the contents of the `output` variable to the `div` element `taFiles` to display the file information on the screen.

```

} // end of while loop

document.getElementById("taFiles").innerHTML = output;
} // end of init() method

```

The `outfile()` function opens an output file in `WRITE` mode on the desktop, giving it the name supplied by the `filename` parameter. It then writes the file data from the `data` parameter to the output file stream (`outStream`) and closes the file.

```

function outfile(fileName, data)
{
    var outFile = air.File.desktopDirectory; // dest folder is desktop
    outFile = outFile.resolvePath(fileName); // name of file to write
    var outStream = new air.FileStream();
    // open output file stream in WRITE mode
    outStream.open(outFile, air.FileMode.WRITE);
    // write out the file
    outStream.writeBytes(data, 0, data.length);
    // close it
    outStream.close();
}

```

Chapter 17: Adding PDF content in AIR

Adobe AIR 1.0 and later

Applications running in Adobe® AIR® can render not only SWF and HTML content, but also PDF content. AIR applications render PDF content using the `HTMLLoader` class, the WebKit engine, and the Adobe® Reader® browser plug-in. In an AIR application, PDF content can either stretch across the full height and width of your application or alternatively as a portion of the interface. The Adobe Reader browser plug-in controls display of PDF files in an AIR application. modifications to the Reader toolbar interface (such as controls for position, anchoring, and visibility) persist in subsequent viewing of PDF files in both AIR applications and the browser.

Important: To render PDF content in AIR, the user must have Adobe Reader or Adobe® Acrobat® version 8.1 or higher installed.

Detecting PDF Capability

Adobe AIR 1.0 and later

If the user does not have Adobe Reader or Adobe Acrobat 8.1 or higher, PDF content is not displayed in an AIR application. To detect if a user can render PDF content, first check the `HTMLLoader.pdfCapability` property. This property is set to one of the following constants of the `HTMLPDFCapability` class:

Constant	Description
<code>HTMLPDFCapability.STATUS_OK</code>	A sufficient version (8.1 or greater) of Adobe Reader is detected and PDF content can be loaded into an <code>HTMLLoader</code> object.
<code>HTMLPDFCapability.ERROR_INSTALLED_READER_NOT_FOUND</code>	No version of Adobe Reader is detected. An <code>HTMLLoader</code> object cannot display PDF content.
<code>HTMLPDFCapability.ERROR_INSTALLED_READER_TOO_OLD</code>	Adobe Reader has been detected, but the version is too old. An <code>HTMLLoader</code> object cannot display PDF content.
<code>HTMLPDFCapability.ERROR_PREFERRED_READER_TOO_OLD</code>	A sufficient version (8.1 or later) of Adobe Reader is detected, but the version of Adobe Reader that is set up to handle PDF content is older than Reader 8.1. An <code>HTMLLoader</code> object cannot display PDF content.

On Windows, if Adobe Acrobat or Adobe Reader version 7.x or above is running on the user's system, that version is used even if a later version that supports loading PDF is installed. In this case, if the value of the `pdfCapability` property is `HTMLPDFCapability.STATUS_OK`, when an AIR application attempts to load PDF content, the older version of Acrobat or Reader displays an alert (and no exception is thrown in the AIR application). If this is a possible situation for your end users, consider providing them with instructions to close Acrobat while running your application. You may want to display these instructions if the PDF content does not load within an acceptable time frame.

On Linux, AIR looks for Adobe Reader in the `PATH` exported by the user (if it contains the `acroread` command) and in the `/opt/Adobe/Reader` directory.

The following code detects whether a user can display PDF content in an AIR application. If the user cannot display PDF, the code traces the error code that corresponds to the `HTMLPDFCapability` error object:

```
if (air.HTMLLoader.pdfCapability == air.HTMLPDFCapability.STATUS_OK)
{
    air.trace("PDF content can be displayed");
}
else
{
    air.trace("PDF cannot be displayed. Error code:", HTMLLoader.pdfCapability);
}
```

Loading PDF content

Adobe AIR 1.0 and later

You can add a PDF to an AIR application by creating an HTMLLoader instance, setting its dimensions, and loading the path of a PDF.

You can add a PDF to an AIR application just as you would in a browser. For example, you can load PDF into the top-level HTML content of a window, into an object tag, in a frame, or in an iframe.

The following example loads a PDF from an external site. Replace the value of the src property of the iframe with the path to an available external PDF.

```
<html>
  <body>
    <h1>PDF test</h1>
    <iframe id="pdfFrame"
      width="100%"
      height="100%"
      src="http://www.example.com/test.pdf"/>
  </body>
</html>
```

You can also load content from file URLs and AIR-specific URL schemes, such as app and app-storage. For example, the following code loads the test.pdf file in the PDFs subdirectory of the application directory:

```
app:/js_api_reference.pdf
```

For more information on AIR URL schemes, see “[URI schemes](#)” on page 318.

Scripting PDF content

Adobe AIR 1.0 and later

You can use JavaScript to control PDF content just as you can in a web page in the browser.

JavaScript extensions to Acrobat provide the following features, among others:

- Controlling page navigation and magnification
- Processing forms within the document
- Controlling multimedia events

Full details on JavaScript extensions for Adobe Acrobat are provided at the Adobe Acrobat Developer Connection at <http://www.adobe.com/devnet/acrobat/javascript.html>.

HTML-PDF communication basics

Adobe AIR 1.0 and later

JavaScript in an HTML page can send a message to JavaScript in PDF content by calling the `postMessage()` method of the DOM object representing the PDF content. For example, consider the following embedded PDF content:

```
<object id="PDFObj" data="test.pdf" type="application/pdf" width="100%" height="100%"/>
```

The following JavaScript code in the containing HTML content sends a message to the JavaScript in the PDF file:

```
pdfObject = document.getElementById("PDFObj");  
pdfObject.postMessage(["testMsg", "hello"]);
```

The PDF file can include JavaScript for receiving this message. You can add JavaScript code to PDF files in some contexts, including the document-, folder-, page-, field-, and batch-level contexts. Only the document-level context, which defines scripts that are evaluated when the PDF document opens, is discussed here.

A PDF file can add a `messageHandler` property to the `hostContainer` object. The `messageHandler` property is an object that defines handler functions to respond to messages. For example, the following code defines the function to handle messages received by the PDF file from the host container (which is the HTML content embedding the PDF file):

```
this.hostContainer.messageHandler = {onMessage: myOnMessage};  
  
function myOnMessage(aMessage)  
{  
    if(aMessage[0] == "testMsg")  
    {  
        app.alert("Test message: " + aMessage[1]);  
    }  
    else  
    {  
        app.alert("Error");  
    }  
}
```

JavaScript code in the HTML page can call the `postMessage()` method of the PDF object contained in the page. Calling this method sends a message ("Hello from HTML") to the document-level JavaScript in the PDF file:

```
<html>
  <head>
    <title>PDF Test</title>
    <script>
      function init()
      {
        pdfObject = document.getElementById("PDFObj");
        try {
          pdfObject.postMessage(["alert", "Hello from HTML"]);
        }
        catch (e)
        {
          alert( "Error: \n name = " + e.name + "\n message = " + e.message );
        }
      }
    </script>
  </head>
  <body onload='init()''>
    <object
      id="PDFObj"
      data="test.pdf"
      type="application/pdf"
      width="100%" height="100%"/>
  </body>
</html>
```

For a more advanced example, and for information on using Acrobat 8 to add JavaScript to a PDF file, see [Cross-scripting PDF content in Adobe AIR](#).

Known limitations for PDF content in AIR

Adobe AIR 1.0 and later

PDF content in Adobe AIR has the following limitations:

- PDF content does not display in a window (a `NativeWindow` object) that is transparent (where the `transparent` property is set to `true`).
- The display order of a PDF file operates differently than other display objects in an AIR application. Although PDF content clips correctly according to HTML display order, it will always sit on top of content in the AIR application's display order.
- If certain visual properties of an `HTMLLoader` object that contains a PDF document are changed, the PDF document will become invisible. These properties include the `filters`, `alpha`, `rotation`, and `scaling` properties. Changing these properties renders the PDF content invisible until the properties are reset. The PDF content is also invisible if you change these properties of display object containers that contain the `HTMLLoader` object.
- PDF content is visible only when the `scaleMode` property of the `Stage` object of the `NativeWindow` object containing the PDF content (the `window.nativeWindow.stage` property) is set to `air.StageScaleMode.NO_SCALE`. When it is set to any other value, the PDF content is not visible.

- Clicking links to content within the PDF file update the scroll position of the PDF content. Clicking links to content outside the PDF file redirect the HTMLLoader object that contains the PDF (even if the target of a link is a new window).
- PDF commenting workflows do not function in AIR.

Chapter 18: Working with sound

The Adobe® AIR® classes include many capabilities not available to HTML content running in the browser, including capabilities for loading and playing sound content.

More Help topics

[flash.media.Sound](#)

[flash.media.Microphone](#)

[flash.events.SampleDataEvent](#)

Basics of working with sound

Before you can control a sound, you need to load the sound into the Adobe AIR application. There are five ways you can get audio data into AIR:

- You can load an external sound file such as an mp3 file into the application.
- You can embed the sound information into a SWF file, load it (using `<script src="[swfFile].swf" type="application/x-shockwave-flash"/>`) and play it.
- You can get audio input using a microphone attached to a user's computer.
- You can access sound data that's streamed from a server.
- You can dynamically generate sound data.

When you load sound data from an external sound file, you can begin playing back the start of the sound file while the rest of the sound data is still loading.

Although there are various sound file formats used to encode digital audio, AIR supports sound files that are stored in the mp3 format. It cannot directly load or play sound files in other formats like WAV or AIFF.

While you're working with sound in AIR, you'll likely work with several classes from the `runtime.flash.media` package. The `Sound` class is the class you use to get access to audio information by loading a sound file or assigning a function to an event that samples sound data and then starting playback. Once you start playing a sound, AIR gives you access to a `SoundChannel` object. An audio file that you've loaded can only be one of several sounds that an application plays simultaneously. Each individual sound that's playing uses its own `SoundChannel` object; the combined output of all the `SoundChannel` objects mixed together is what actually plays over the speakers. You use this `SoundChannel` instance to control properties of the sound and to stop its playback. Finally, if you want to control the combined audio, the `SoundMixer` class gives you control over the mixed output.

You can also use several other runtime classes to perform more specific tasks when you're working with sound in AIR. For more information on all the sound-related classes, see "[Understanding the sound architecture](#)" on page 276.

The Adobe AIR developer's center provides a sample application: [Using Sound in an HTML-based Application](#) (http://www.adobe.com/go/learn_air_qs_sound_html_en).

Understanding the sound architecture

Your applications can load sound data from five main sources:

- External sound files loaded at run time
- Sound resources embedded within a SWF file
- Sound data from a microphone attached to the user's system
- Sound data streamed from a remote media server, such as Flash Media Server
- Sound data being generated dynamically by using the `sampleData` event handler

Sound data can be fully loaded before it is played back, or it can be streamed, meaning that it is played back while it is still loading.

Adobe AIR supports sound files that are stored in the mp3 format. They cannot directly load or play sound files in other formats like WAV or AIFF. (However, AIR can also load and play AAC audio files using the `NetStream` class.)

The AIR sound architecture includes the following classes:

Class	Description
Sound	The <code>Sound</code> class handles the loading of sound, manages basic sound properties, and starts a sound playing.
SoundChannel	When an application plays a <code>Sound</code> object, a new <code>SoundChannel</code> object is created to control the playback. The <code>SoundChannel</code> object controls the volume of both the left and right playback channels of the sound. Each sound that plays has its own <code>SoundChannel</code> object.
SoundLoaderContext	The <code>SoundLoaderContext</code> class specifies how many seconds of buffering to use when loading a sound, and whether the runtime looks for a cross-domain policy file from the server when loading a file. A <code>SoundLoaderContext</code> object is used as a parameter to the <code>Sound.load()</code> method.
SoundMixer	The <code>SoundMixer</code> class controls playback and security properties that pertain to all sounds in an application. In effect, multiple sound channels are mixed through a common <code>SoundMixer</code> object. Property values in the <code>SoundMixer</code> object affect all <code>SoundChannel</code> objects that are currently playing.
SoundTransform	The <code>SoundTransform</code> class contains values that control sound volume and panning. A <code>SoundTransform</code> object can be applied to an individual <code>SoundChannel</code> object, to the global <code>SoundMixer</code> object, or to a <code>Microphone</code> object, among others.
ID3Info	An <code>ID3Info</code> object contains properties that represent ID3 metadata information that is often stored in MP3 sound files.
Microphone	The <code>Microphone</code> class represents a microphone or other sound input device attached to the user's computer. Audio input from a microphone can be routed to local speakers or sent to a remote server. The <code>Microphone</code> object controls the gain, sampling rate, and other characteristics of its own sound stream.

Each sound that is loaded and played needs its own instance of the `Sound` class and the `SoundChannel` class. During playback, the `SoundMixer` class mixes the output from multiple `SoundChannel` instances.

The `Sound`, `SoundChannel`, and `SoundMixer` classes are not used for sound data obtained from a microphone or from a streaming media server like Flash Media Server.

Loading external sound files

Each instance of the `Sound` class exists to load and trigger the playback of a specific sound resource. An application can't reuse a `Sound` object to load more than one sound. To load a new sound resource, the application needs to create another `Sound` object.

Creating a sound object

If you are loading a small sound file, such as a click sound to be attached to a button, your application can create a `Sound` and have it automatically load the sound file, as the following example shows:

```
var req = new air.URLRequest("click.mp3");  
var s = new air.Sound(req);
```

The `Sound()` constructor accepts a `URLRequest` object as its first parameter. When a value for the `URLRequest` parameter is supplied, the new `Sound` object starts loading the specified sound resource automatically.

In all but the simplest cases, your application should pay attention to the sound's loading progress and watch for errors during loading. For example, if the click sound is fairly large, the application may not completely load it by the time the user clicks the button that triggers the sound. Trying to play an unloaded sound could cause a run-time error. It's safer to wait for the sound to load completely before letting users take actions that can start sounds playing.

About sound events

A `Sound` object dispatches a number of different events during the sound loading process. Your application can listen for these events to track loading progress and make sure that the sound loads completely before playing. The following table lists the events that the `Sound` object is able to dispatch:

Event	Description
<code>open</code> (<code>air.Event.OPEN</code>)	Dispatched right before the sound loading operation begins.
<code>progress</code> (<code>air.ProgressEvent.PROGRESS</code>)	Dispatched periodically during the sound loading process when data is received from the file or stream.
<code>id3</code> (<code>air.Event.ID3</code>)	Dispatched when ID3 data is available for an mp3 sound.
<code>complete</code> (<code>air.Event.COMPLETE</code>)	Dispatched when all of the sound resource's data has been loaded.
<code>ioError</code> (<code>air.IOErrorEvent.IO_ERROR</code>)	Dispatched when a sound file cannot be located or when the loading process is interrupted before all sound data can be received.

The following code illustrates how to play a sound after it has finished loading:

```
var s = new air.Sound();  
s.addEventListener(air.Event.COMPLETE, onSoundLoaded);  
var req = new air.URLRequest("bigSound.mp3");  
s.load(req);  
  
function onSoundLoaded(event)  
{  
    var localSound = event.target;  
    localSound.play();  
}
```

First, the code sample creates a new `Sound` object without giving it an initial value for the `URLRequest` parameter. Then, it listens for the `complete` event from the `Sound` object, which causes the `onSoundLoaded()` method to execute when all the sound data is loaded. Next, it calls the `Sound.load()` method with a new `URLRequest` value for the sound file.

The `onSoundLoaded()` method executes when the sound loading is complete. The `target` property of the `Event` object is a reference to the `Sound` object. Calling the `play()` method of the `Sound` object then starts the sound playback.

Monitoring the sound loading process

Sound files can be large and take a long time to load, especially if they are loaded from the Internet. An application can play sounds before they are fully loaded. You might want to give the user an indication of how much of the sound data has been loaded and how much of the sound has already been played.

The `Sound` class dispatches two events that make it relatively easy to display the loading progress of a sound: `progress` and `complete`. The following example shows how to use these events to display progress information about the sound being loaded:

```
var s = new Sound();
s.addEventListener(air.ProgressEvent.PROGRESS,
    onLoadProgress);
s.addEventListener(air.Event.COMPLETE,
    onLoadComplete);
s.addEventListener(air.IOErrorEvent.IO_ERROR,
    onIOError);

var req = new air.URLRequest("bigSound.mp3");
s.load(req);

function onLoadProgress(event)
{
    var loadedPct = Math.round(100 * (event.bytesLoaded / event.bytesTotal));
    air.trace("The sound is " + loadedPct + "% loaded.");
}

function onLoadComplete(event)
{
    var localSound = event.target;
    localSound.play();
}

function onIOError(event)
{
    air.trace("The sound could not be loaded: " + event.text);
}
```

This code first creates a `Sound` object and then adds listeners to that object for the `progress` and `complete` events. After the `Sound.load()` method has been called and the first data is received from the sound file, a `progress` event occurs, and triggers the `onSoundLoadProgress()` method.

The fraction of the sound data that has been loaded is equal to the value of the `bytesLoaded` property of the `ProgressEvent` object divided by the value of the `bytesTotal` property. The same `bytesLoaded` and `bytesTotal` properties are available on the `Sound` object as well.

This example also shows how an application can recognize and respond to an error when loading sound files. For example, if a sound file with the given filename cannot be located, the `Sound` object dispatches an `ioError` event. In the previous code, the `onIOError()` method executes and displays a brief error message when an error occurs.

Working with embedded sounds

In AIR, you can use JavaScript to access sounds embedded in SWF files. You can load these SWF files into the application using any of the following means:

- By loading the SWF file with a `<script>` tag in the HTML page
- By loading a SWF file using the `runtime.flash.display.Loader` class

The exact method of embedding a sound file into your application's SWF file varies according to your SWF content development environment. For information on embedding media in SWF files, see the documentation for your SWF content development environment

To use the embedded sound, you reference the class name for that sound in ActionScript. For example, the following code starts by creating an instance of the automatically generated `DrumSound` class:

```
var drum = new DrumSound();  
var channel = drum.play();
```

`DrumSound` is a subclass of the `flash.media.Sound` class, so it inherits the methods and properties of the `Sound` class. The `play()` method included, as the preceding example shows.

Working with streaming sound files

When a sound file or video file is playing back while its data is still being loaded, it is said to be *streaming*. Sound files loaded from a remote server are often streamed so that the user doesn't have to wait for all the sound data to load before listening to the sound.

The `SoundMixer.bufferTime` property represents the number of milliseconds of sound data that an application gathers before letting the sound play. In other words, if the `bufferTime` property is set to 5000, the application loads at least 5000 milliseconds worth of data from the sound file before the sound begins to play. The default `SoundMixer.bufferTime` value is 1000.

Your application can override the global `SoundMixer.bufferTime` value for an individual sound by explicitly specifying a new `bufferTime` value when loading the sound. To override the default buffer time, first create an instance of the `SoundLoaderContext` class, set its `bufferTime` property, and then pass it as a parameter to the `Sound.load()` method. The following example shows this:

```
var s = new air.Sound();  
var url = "http://www.example.com/sounds/bigSound.mp3";  
var req = new air.URLRequest(url);  
var context = new air.SoundLoaderContext(8000, true);  
s.load(req, context);  
s.play();
```

As playback continues, AIR tries to keep the sound buffer at the same size or greater. If the sound data loads faster than the playback speed, playback continues without interruption. However, if the data loading rate slows down because of network limitations, the playhead could reach the end of the sound buffer. If this happens, playback is suspended, though it automatically resumes once more sound data has been loaded.

To find out if playback is suspended because AIR is waiting for data to load, use the `Sound.isBuffering` property.

Working with dynamically generated audio

Instead of loading or streaming an existing sound, you can generate audio data dynamically. You can generate audio data when you assign an event listener for the `sampleData` event of a `Sound` object. (The `sampleData` event is defined in the `SampleDataEvent` class.) In this environment, the `Sound` object doesn't load sound data from a file. Instead, it acts as a socket for sound data that is being streamed to it by using the function you assign to this event.

When you add a `sampleData` event listener to a `Sound` object, the object periodically requests data to add to the sound buffer. This buffer contains data for the `Sound` object to play. When you call the `play()` method of the `Sound` object, it dispatches the `sampleData` event when requesting new sound data. (This is true only when the `Sound` object has not loaded mp3 data from a file.)

The `SampleDataEvent` object includes a `data` property. In your event listener, you write `ByteArray` objects to this `data` object. The byte arrays you write to this object add to buffered sound data that the `Sound` object plays. The byte array in the buffer is a stream of floating-point values from -1 to 1. Each floating-point value represents the amplitude of one channel (left or right) of a sound sample. Sound is sampled at 44,100 samples per second. Each sample contains a left and right channel, interleaved as floating-point data in the byte array.

In your handler function, you use the `ByteArray.writeFloat()` method to write to the `data` property of the `sampleData` event. For example, the following code generates a sine wave:

```
var mySound = new air.Sound();
mySound.addEventListener(air.SampleDataEvent.SAMPLE_DATA, sineWaveGenerator);
mySound.play();
function sineWaveGenerator(event)
{
    for (i = 0; i < 8192; i++)
    {
        var n = Math.sin((i + event.position) / Math.PI / 4);
        event.data.writeFloat(n);
        event.data.writeFloat(n);
    }
}
```

When you call `sound.play()`, the application starts calling your event handler, requesting sound sample data. The application continues to send events as the sound plays back until you stop providing data, or until you call `SoundChannel.stop()`.

The latency of the event varies from platform to platform, and could change in future versions of AIR. Do not depend on a specific latency; calculate it instead. To calculate the latency, use the following formula:

```
(SampleDataEvent.position / 44.1) - SoundChannelObject.position
```

Provide from 2048 through 8192 samples to the `data` property of the `SampleDataEvent` object (for each call to the event listener). For best performance, provide as many samples as possible (up to 8192). The fewer samples you provide, the more likely it is that clicks and pops occur during playback. This behavior can differ on various platforms and can occur in various situations—for example, when resizing the browser. Code that works on one platform when you provide only 2048 sample might not work as well when run on a different platform. If you require the lowest latency possible, consider making the amount of data user-selectable.

If you provide fewer than 2048 samples (per call to the `sampleData` event listener), the application stops after playing the remaining samples. It then dispatches a `SoundComplete` event.

Modifying sound from mp3 data

You use the `Sound.extract()` method to extract data from a `Sound` object. You can use (and modify) that data to write to the dynamic stream of another `Sound` object for playback. For example, the following code uses the bytes of a loaded mp3 file and passes them through a filter function, `upOctave()`:

```
var mySound = new air.Sound();
var sourceSnd = new air.Sound();
var urlReq = new air.URLRequest("test.mp3");
sourceSnd.load(urlReq);
sourceSnd.addEventListener(air.Event.COMPLETE, loaded);
function loaded(event)
{
    mySound.addEventListener(SampleDataEvent.SAMPLE_DATA, processSound);
    mySound.play();
}
function processSound(event)
{
    var bytes = new air.ByteArray();
    sourceSnd.extract(bytes, 8192);
    event.data.writeBytes(upOctave(bytes));
}
function upOctave(bytes)
{
    var returnBytes = new air.ByteArray();
    bytes.position = 0;
    while(bytes.bytesAvailable > 0)
    {
        returnBytes.writeFloat(bytes.readFloat());
        returnBytes.writeFloat(bytes.readFloat());
        if (bytes.bytesAvailable > 0)
        {
            bytes.position += 8;
        }
    }
    return returnBytes;
}
```

Limitations on generated sounds

When you use a `sampleData` event listener with a `Sound` object, the only other `Sound` methods that are enabled are `Sound.extract()` and `Sound.play()`. Calling any other methods or properties results in an exception. All methods and properties of the `SoundChannel` object are still enabled.

Playing sounds

Playing a loaded sound can be as simple as calling the `Sound.play()` method for a `Sound` object, as follows:

```
var req = new air.URLRequest("smallSound.mp3");
var snd = new air.Sound(req);
snd.play();
```

Sound playback operations

When playing back sounds, you can perform the following operations:

- Play a sound from a specific starting position
- Pause a sound and resume playback from the same position later
- Know exactly when a sound finishes playing
- Track the playback progress of a sound
- Change volume or panning while a sound plays

To perform these operations during playback, use the `SoundChannel`, `SoundMixer`, and `SoundTransform` classes.

The `SoundChannel` class controls the playback of a single sound. The `SoundChannel.position` property can be thought of as a playhead, indicating the current point in the sound data that's being played.

When an application calls the `Sound.play()` method, a new instance of the `SoundChannel` class is created to control the playback.

Your application can play a sound from a specific starting position by passing that position, in terms of milliseconds, as the `startTime` parameter of the `Sound.play()` method. It can also specify a fixed number of times to repeat the sound in rapid succession by passing a numeric value in the `loops` parameter of the `Sound.play()` method.

When the `Sound.play()` method is called with both a `startTime` parameter and a `loops` parameter, the sound is played back repeatedly from the same starting point each time. The following code shows this:

```
var req = new air.URLRequest("repeatingSound.mp3");  
var snd = new air.Sound();  
snd.play(1000, 3);
```

In this example, the sound is played from a point one second after the start of the sound, three times in succession.

Pausing and resuming a sound

If your application plays long sounds, like songs or podcasts, you probably want to let users pause and resume the playback of those sounds. A sound cannot literally be paused during playback; it can only be stopped. However, a sound can be played starting from any point. You can record the position of the sound at the time it was stopped, and then replay the sound starting at that position later.

For example, let's say your code loads and plays a sound file like this:

```
var req = new air.URLRequest("bigSound.mp3");  
var snd = new air.Sound(req);  
var channel = snd.play();
```

While the sound plays, the `position` property of the `channel` object indicates the point in the sound file that is currently being played. Your application can store the position value before stopping the sound from playing, as follows:

```
var pausePosition = channel.position;  
channel.stop();
```

To resume playing the sound, pass the previously stored position value to restart the sound from the same point it stopped at before.

```
channel = snd.play(pausePosition);
```


Monitoring playback

Your application might want to know when a sound stops playing. Then it can start playing another sound or clean up some resources used during the previous playback. The `SoundChannel` class dispatches a `soundComplete` event when its sound finishes playing. Your application can listen for this event and take appropriate action, as the following example shows:

```
var snd = new air.Sound("smallSound.mp3");
var channel = snd.play();
s.addEventListener(air.Event.SOUND_COMPLETE, onPlaybackComplete);

public function onPlaybackComplete(event)
{
    air.trace("The sound has finished playing.");
}
```

The `SoundChannel` class does not dispatch `progress` events during playback. To report on playback progress, your application can set up its own timing mechanism and track the position of the sound playhead.

To calculate what percentage of a sound has been played, you can divide the value of the `SoundChannel.position` property by the length of the sound data that's being played:

```
var playbackPercent = 100 * (channel.position / snd.length);
```

However, this code only reports accurate playback percentages if the sound data was fully loaded before playback began. The `Sound.length` property shows the size of the sound data that is currently loaded, not the eventual size of the entire sound file. To track the playback progress of a streaming sound that is still loading, your application should estimate the eventual size of the full sound file and use that value in its calculations. You can estimate the eventual length of the sound data using the `bytesLoaded` and `bytesTotal` properties of the `Sound` object, as follows:

```
var estimatedLength = Math.ceil(snd.length / (snd.bytesLoaded / snd.bytesTotal));
var playbackPercent = 100 * (channel.position / estimatedLength);
```

The following code loads a larger sound file and uses the `setInterval()` function as its timing mechanism for showing playback progress. It periodically reports on the playback percentage, which is the current position value divided by the total length of the sound data:

```
var snd = new air.Sound();
var url = "http://www.example.com/sounds/test.mp3";
var req = new air.URLRequest(url);
snd.load(req);

var channel = snd.play();
var timer = setInterval(monitorProgress, 100);
snd.addEventListener(air.Event.SOUND_COMPLETE, onPlaybackComplete);

function monitorProgress(event)
{
    var estimatedLength = Math.ceil(snd.length / (snd.bytesLoaded / snd.bytesTotal));
    var playbackPercent = Math.round(100 * (channel.position / estimatedLength));
    air.trace("Sound playback is " + playbackPercent + "% complete.");
}

function onPlaybackComplete(event)
{
    air.trace("The sound has finished playing.");
    clearInterval(timer);
}
```

After the sound data starts loading, this code calls the `snd.play()` method and stores the resulting `SoundChannel` object in the `channel` variable. Then it adds a `monitorProgress()` method, which the `setInterval()` function calls repeatedly. The code uses an event listener to the `SoundChannel` object for the `soundComplete` event that occurs when playback is complete.

The `monitorProgress()` method estimates the total length of the sound file based on the amount of data that has already been loaded. It then calculates and displays the current playback percentage.

When the entire sound has been played, the `onPlaybackComplete()` function executes. This function removes the callback method for the `setInterval()` function, so that the application doesn't display progress updates after playback is done.

Stopping streaming sounds

There is a quirk in the playback process for sounds that are streaming—that is, for sounds that are still loading while they are being played. When you call the `stop()` method on a `SoundChannel` instance that is playing back a streaming sound, the sound playback stops and then it restarts from the beginning of the sound. This occurs because the sound loading process is still underway. To stop both the loading and the playback of a streaming sound, call the `Sound.close()` method.

Controlling sound volume and panning

An individual `SoundChannel` object controls both the left and the right stereo channels for a sound. If an mp3 sound is a monaural sound, the left and right stereo channels of the `SoundChannel` object contain identical waveforms.

You can find out the amplitude of each stereo channel of the sound being played using the `leftPeak` and `rightPeak` properties of the `SoundChannel` object. These properties show the peak amplitude of the sound waveform itself. They do not represent the actual playback volume. The actual playback volume is a function of the amplitude of the sound wave and the volume values set in the `SoundChannel` object and the `SoundMixer` class.

The `pan` property of a `SoundChannel` object can be used to specify a different volume level for each of the left and right channels during playback. The `pan` property can have a value ranging from -1 to 1. A value of -1 means the left channel plays at top volume while the right channel is silent. A value of 1 means the right channel plays at top volume while the left channel is silent. Values in between -1 and 1 set proportional values for the left and right channel values. A value of 0 means that both channels play at a balanced, mid-volume level.

The following code example creates a `SoundTransform` object with a volume value of 0.6 and a pan value of -1 (upper-left channel volume and no right channel volume). It passes the `SoundTransform` object as a parameter to the `play()` method. The `play()` method applies that `SoundTransform` object to the new `SoundChannel` object that is created to control the playback.

```
var req = new air.URLRequest("bigSound.mp3");
var snd = new air.Sound(req);
var trans = new air.SoundTransform(0.6, -1);
var channel = snd.play(0, 1, trans);
```

You can alter the volume and panning while a sound plays. Set the `pan` or `volume` properties of a `SoundTransform` object and then apply that object as the `soundTransform` property of a `SoundChannel` object.

You can also set global volume and pan values for all sounds at once, using the `soundTransform` property of the `SoundMixer` class. The following example shows this:

```
SoundMixer.soundTransform = new air.SoundTransform(1, -1);
```

You can also use a `SoundTransform` object to set volume and pan values for a `Microphone` object (see “[Capturing sound input](#)” on page 289).

The following example alternates the panning of the sound from the left channel to the right channel and back while the sound plays:

```
var snd = new air.Sound();
var req = new air.URLRequest("bigSound.mp3");
snd.load(req);

var panCounter = 0;

var trans = new air.SoundTransform(1, 0);
var channel = snd.play(0, 1, trans);
channel.addEventListener(air.Event.SOUND_COMPLETE,
                        onPlaybackComplete);

var timer = setInterval(panner, 100);

function panner()
{
    trans.pan = Math.sin(panCounter);
    channel.soundTransform = trans; // or SoundMixer.soundTransform = trans;
    panCounter += 0.05;
}

function onPlaybackComplete(event)
{
    clearInterval(timer);
}
```

The code starts by loading a sound file and then creating a `SoundTransform` object with volume set to 1 (full volume) and pan set to 0 (evenly balanced between left and right). Then it calls the `snd.play()` method, passing the `SoundTransform` object as a parameter.

While the sound plays, the `panner()` method executes repeatedly. The `panner()` method uses the `Math.sin()` function to generate a value between -1 and 1. This range corresponds to the acceptable values of the `SoundTransform.pan` property. The `SoundTransform` object's `pan` property is set to the new value, and then the channel's `soundTransform` property is set to use the altered `SoundTransform` object.

To run this example, replace the filename `bigSound.mp3` with the name of a local mp3 file. Then run the example. You should hear the left channel volume getting louder while the right channel volume gets softer, and vice versa.

In this example, the same effect could be achieved by setting the `soundTransform` property of the `SoundMixer` class. However, that would affect the panning of all sounds currently playing, not just the single sound this `SoundChannel` object plays.

Working with sound metadata

Sound files that use the mp3 format can contain additional data about the sound in the form of ID3 tags.

Not every mp3 file contains ID3 metadata. When a `Sound` object loads an mp3 sound file, it dispatches an `Event.ID3` event if the sound file contains ID3 metadata. To prevent run-time errors, your application should wait to receive the `Event.ID3` event before accessing the `Sound.id3` property for a loaded sound.

The following code shows how to recognize when the ID3 metadata for a sound file has been loaded:

```
var s = new air.Sound();
s.addEventListener(air.Event.ID3, onID3InfoReceived);
var urlReq = new air.URLRequest("mySound.mp3");
s.load(urlReq);

function onID3InfoReceived(event)
{
    var id3 = event.target.id3;

    air.trace("Received ID3 Info:");
    for (propName in id3)
    {
        air.trace(propName + " = " + id3[propName]);
    }
}
```

This code starts by creating a `Sound` object and telling it to listen for the `id3` event. When the sound file's ID3 metadata is loaded, the `onID3InfoReceived()` method is called. The target of the Event object that is passed to the `onID3InfoReceived()` method is the original `Sound` object. The method then gets the `Sound` object's `id3` property and iterates through its named properties to trace their values.

Accessing raw sound data

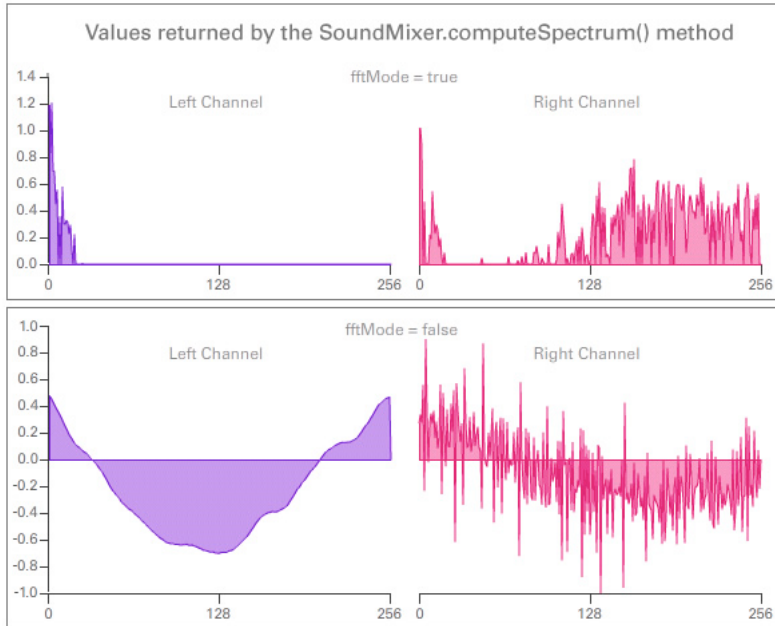
The `SoundMixer.computeSpectrum()` method lets an application read the raw sound data for the waveform that is currently being played. If more than one `SoundChannel` object is currently playing, the `SoundMixer.computeSpectrum()` method shows the combined sound data of every `SoundChannel` object mixed together.

How sound data is returned

The sound data is returned as a `ByteArray` object containing 512 four-byte sets of data, each of which represents a floating point value between -1 and 1. These values represent the amplitude of the points in the sound waveform being played. The values are delivered in two groups of 256, the first group for the left stereo channel and the second group for the right stereo channel.

The `SoundMixer.computeSpectrum()` method returns frequency spectrum data rather than waveform data if the `FFTModes` parameter is set to `true`. The frequency spectrum shows amplitude arranged by sound frequency, from lowest frequency to highest. A Fast Fourier Transform (FFT) is used to convert the waveform data into frequency spectrum data. The resulting frequency spectrum values range from 0 to roughly 1.414 (the square root of 2).

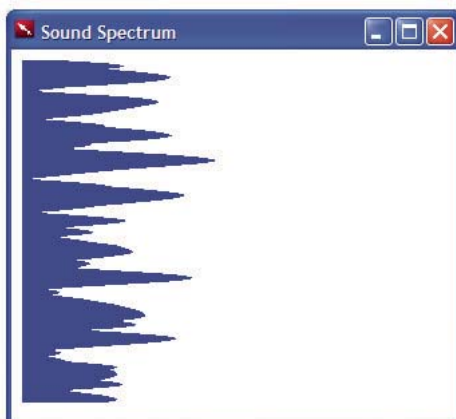
The following diagram compares the data returned from the `computeSpectrum()` method when the `FFTMMode` parameter is set to `true` and when it is set to `false`. The sound used for this diagram contains a loud bass sound in the left channel and a drum hit sound in the right channel.



The `computeSpectrum()` method can also return data that has been resampled at a lower bit rate. Generally, this results in smoother waveform data or frequency data at the expense of detail. The `stretchFactor` parameter controls the rate at which the `computeSpectrum()` method data is sampled. When the `stretchFactor` parameter is set to 0, the default, the sound data is sampled at a rate of 44.1 kHz. The rate is halved at each successive value of the `stretchFactor` parameter. So a value of 1 specifies a rate of 22.05 kHz, a value of 2 specifies a rate of 11.025 kHz, and so on. The `computeSpectrum()` method still returns 256 floating point values per stereo channel when a higher `stretchFactor` value is used.

Building a simple sound visualizer

The following example uses the `SoundMixer.computeSpectrum()` method to show a chart of the sound waveform that animates periodically:



```
<html>
  <title>Sound Spectrum</title>
  <script src="AIRAliases.js" />
  <script>
    const PLOT_WIDTH = 600;
    const CHANNEL_LENGTH = 256;

    var snd = new air.Sound();
    var req = new air.URLRequest("test.mp3");
    var bytes = new air.ByteArray();
    var divStyles = new Array;

    /**
     * Initializes the application. It draws 256 DIV elements to the document body,
     * and sets up a divStyles array that contains references to the style objects of
     * each DIV element. It then calls the playSound() function.
     */
    function init()
    {
      var div;
      for (i = 0; i < CHANNEL_LENGTH; i++)
      {
        div = document.createElement("div");
        div.style.height = "1px";
        div.style.width = "0px";
        div.style.backgroundColor = "blue";
        document.body.appendChild(div);
        divStyles[i] = div.style;
      }
      playSound();
    }
    /**
     * Plays a sound, and calls setInterval() to call the setMeter() function
     * periodically, to display the sound spectrum data.
     */
    function playSound()
    {
      if (snd.url != null)
      {
        snd.close();
      }
      snd.load(req);
      var channel = snd.play();
      timer = setInterval(setMeter, 100);
      snd.addEventListener(air.Event.SOUND_COMPLETE, onPlaybackComplete);
    }

    /**
     * Computes the width of each of the 256 colored DIV tags in the document,
     * based on data returned by the call to SoundMixer.computeSpectrum(). The
     * first 256 floating point numbers in the byte array represent the data from
     * the left channel, and then next 256 floating point numbers represent the
     * data from the right channel.
     */
    function setMeter()
    {
      air.SoundMixer.computeSpectrum(bytes, false, 0);
    }
  </script>
</html>
```

```
var n;
for (var i = 0; i < CHANNEL_LENGTH; i++)
{
    bytes.position = i * 4;
    n = Math.abs(bytes.readFloat());
    bytes.position = 256*4 + i * 4;
    n += Math.abs(bytes.readFloat());
    divStyles[i].width = n * PLOT_WIDTH;
}
}
/**
 * When the sound is done playing, remove the intermediate process
 * started by setInterval().
 */
function onPlaybackComplete(event)
{
    clearInterval(interval);
}
</script>
<body onload="init()">
</body>
</html>
```

This example first loads and plays a sound file and then uses the `setInterval()` function to monitor the `SoundMixer.computeSpectrum()` method, which stores the sound wave data in the `bytes` `ByteArray` object.

The sound waveform is plotted by setting the width of `div` elements representing a bar graph.

Capturing sound input

The `Microphone` class lets your application connect to a microphone or other sound input device on the user's system. An application can broadcast the input audio to that system's speakers or send the audio data to a remote server, such as the Flash Media Server. You cannot access raw audio data from the microphone; you can only send audio to the system's speakers or send compressed audio data to a remote server. You can use either Speex or Nellymoser codec for data sent to a remote server. (The Speex codec is available in AIR 1.5.)

Accessing a microphone

The `Microphone` class does not have a constructor method. Instead, you use the static `Microphone.getMicrophone()` method to obtain a new `Microphone` instance, as the following example shows:

```
var mic = air.Microphone.getMicrophone();
```

Calling the `Microphone.getMicrophone()` method without a parameter returns the first sound input device discovered on the user's system.

A system can have more than one sound input device attached to it. Your application can use the `Microphone.names` property to get an array of the names of all available sound input devices. Then it can call the `Microphone.getMicrophone()` method with an `index` parameter that matches the index value of a device's name in the array.

A system might not have a microphone or other sound input device attached to it. You can use the `Microphone.names` property or the `Microphone.getMicrophone()` method to check whether the user has a sound input device installed. If the user doesn't have a sound input device installed, the `names` array has a length of zero, and the `getMicrophone()` method returns a value of `null`.

Routing microphone audio to local speakers

Audio input from a microphone can be routed to the local system speakers by calling the `Microphone.setLoopback()` method with a parameter value of `true`.

When sound from a local microphone is routed to local speakers, there is a risk of creating an audio feedback loop. This can cause loud squealing sounds and can potentially damage sound hardware. Calling the `Microphone.setUseEchoSuppression()` method with a parameter value of `true` reduces, but does not completely eliminate, the risk that audio feedback will occur. Adobe recommends that you always call `Microphone.setUseEchoSuppression(true)` before calling `Microphone.setLoopback(true)`, unless you are certain that the user is playing back the sound using headphones or something other than speakers.

The following code shows how to route the audio from a local microphone to the local system speakers:

```
var mic = air.Microphone.getMicrophone();  
mic.setUseEchoSuppression(true);  
mic.setLoopBack(true);
```

Altering microphone audio

Your application can alter the audio data that comes from a microphone in two ways. First, it can change the gain of the input sound, which effectively multiplies the input values by a specified amount. This creates a louder or quieter sound. The `Microphone.gain` property accepts numeric values from 0 through 100. A value of 50 acts like a multiplier of one and specifies normal volume. A value of zero acts like a multiplier of zero and effectively silences the input audio. Values above 50 specify higher than normal volume.

Your application can also change the sample rate of the input audio. Higher sample rates increase sound quality, but they also create denser data streams that use more resources for transmission and storage. The `Microphone.rate` property represents the audio sample rate measured in kilohertz (kHz). The default sample rate is 8 kHz. You can set the `Microphone.rate` property to a value higher than 8 kHz if your microphone supports the higher rate. For example, setting the `Microphone.rate` property to 11 sets the sample rate to 11 kHz; setting it to 22 sets the sample rate to 22 kHz, and so on. The sample rates available depend on the selected codec. When you use the Nellymoser codec, you can specify 5, 8, 11, 16, 22 and 44 kHz as the sample rate. When you use Speex codec (available in AIR 1.5), you can only use 16 kHz.

Detecting microphone activity

To conserve bandwidth and processing resources, the runtime tries to detect when a microphone transmits no sound. When the microphone's activity level stays below the silence level threshold for a period of time, the runtime stops transmitting the audio input and dispatches an `activity` event. If you use the Speex codec (available in AIR 1.5), set the silence level to 0, to ensure that the application continuously transmits audio data. Speex voice activity detection automatically reduces bandwidth.

Three properties of the `Microphone` class monitor and control the detection of activity:

- The read-only `activityLevel` property indicates the amount of sound the microphone is detecting, on a scale from 0 to 100.

- The `silenceLevel` property specifies the amount of sound needed to activate the microphone and dispatch an activity event. The `silenceLevel` property also uses a scale from 0 to 100, and the default value is 10.
- The `silenceTimeout` property describes the number of milliseconds that the activity level must stay below the silence level before an activity event is dispatched. The default `silenceTimeout` value is 2000.

Both the `Microphone.silenceLevel` property and the `Microphone.silenceTimeout` property are read only, but their values can be changed by using the `Microphone.setSilenceLevel()` method.

In some cases, the process of activating the microphone when new activity is detected can cause a short delay. Keeping the microphone active at all times can remove such activation delays. Your application can call the `Microphone.setSilenceLevel()` method with the `silenceLevel` parameter set to zero. This keeps the microphone active and gathering audio data, even when no sound is detected. Conversely, setting the `silenceLevel` parameter to 100 prevents the microphone from being activated at all.

The following example displays information about the microphone and reports on activity events and status events dispatched by a `Microphone` object:

```
var deviceArray = air.Microphone.names;
air.trace("Available sound input devices:");
for (i = 0; i < deviceArray.length; i++)
{
    air.trace("    " + deviceArray[i]);
}

var mic = air.Microphone.getMicrophone();
mic.gain = 60;
mic.rate = 11;
mic.setUseEchoSuppression(true);
mic.setLoopBack(true);
mic.setSilenceLevel(5, 1000);

mic.addEventListener(air.ActivityEvent.ACTIVITY, this.onMicActivity);

var micDetails = "Sound input device name: " + mic.name + '\n';
micDetails += "Gain: " + mic.gain + '\n';
micDetails += "Rate: " + mic.rate + " kHz" + '\n';
micDetails += "Muted: " + mic.muted + '\n';
micDetails += "Silence level: " + mic.silenceLevel + '\n';
micDetails += "Silence timeout: " + mic.silenceTimeout + '\n';
micDetails += "Echo suppression: " + mic.useEchoSuppression + '\n';
air.trace(micDetails);

function onMicActivity(event)
{
    air.trace("activating=" + event.activating + ", activityLevel=" +
        mic.activityLevel);
}
```

When you run the preceding example, speak or make noises into your system microphone and watch the resulting trace statements appear in the console.

Sending audio to and from a media server

Additional audio capabilities are available when using a streaming media server such as Flash Media Server.

In particular, your application can attach a `Microphone` object to a `runtime.flash.net.NetStream` object and transmit data directly from the user's microphone to the server. Audio data can also be streamed from the server to an AIR application.

AIR 1.5 introduces support for the Speex codec. To set the codec used for compressed audio sent to the media server, set the `codec` property of the `Microphone` object. This property can have two values, which are enumerated in the `SoundCodec` class. Setting the `codec` property to `SoundCodec.SPEEX` selects the Speex codec for compressing audio. Setting the property to `SoundCodec.NELLYMOSER` (the default) selects the Nellymoser codec for compressing audio.

For more information, see the Flash Media Server documentation online at <http://www.adobe.com/support/documentation>.

Chapter 19: Client system environment

Flash Player 9 and later, Adobe AIR 1.0 and later

This discussion explains how to interact with the user's system. It shows you how to determine what features are supported and how to build multilingual applications using the user's installed input method editor (IME) if available. It also shows typical uses for application domains.

More Help topics

[flash.system.System](#)

[flash.system.Capabilities](#)

Basics of the client system environment

Flash Player 9 and later, Adobe AIR 1.0 and later

As you build more advanced applications, you may find a need to know details about—and access functions of—your users' operating systems. The `flash.system` package contains a collection of classes that allow you to access system-level functionality such as the following:

- Determining which application and security domain code is executing in
- Determining the capabilities of the user's Flash runtime (such as Flash® Player or Adobe® AIR™) instance, such as the screen size (resolution) and whether certain functionality is available, such as mp3 audio
- Building multilingual sites using the IME
- Interacting with the Flash runtime's container (which could be an HTML page or a container application).
- Saving information to the user's clipboard

The `flash.system` package also includes the `IMEConversionMode` and `SecurityPanel` classes. These classes contain static constants that you use with the IME and Security classes, respectively.

Important concepts and terms

The following reference list contains important terms:

Operating system The main program that runs on a computer, within which all other applications run—such as Microsoft Windows, Mac OS X, or Linux®.

Clipboard The operating system's container for holding text or items that are copied or cut, and from which items are pasted into applications.

Application domain A mechanism for separating classes used in different SWF files, so that if the SWF files include different classes with the same name, the classes don't overwrite each other.

IME (input method editor) A program (or operating system tool) that is used to enter complex characters or symbols using a standard keyboard.

Client system In programming terms, a client is the part of an application (or whole application) that runs on an individual's computer and is used by a single user. The client system is the underlying operating system on the user's computer.

Using the System class

Flash Player 9 and later, Adobe AIR 1.0 and later

The `System` class contains methods and properties that allow you to interact with the user's operating system and retrieve the current memory usage of the runtime. The methods and properties of the `System` class also allow you to listen for `imeComposition` events, instruct the runtime to load external text files using the user's current code page or to load them as Unicode, or set the contents of the user's clipboard.

Getting data about the user's system at run time

Flash Player 9 and later, Adobe AIR 1.0 and later

By checking the `System.totalMemory` property, you can determine the amount of memory (in bytes) that the runtime is currently using. This property allows you to monitor memory usage and optimize your applications based on how the memory level changes. For example, if a particular visual effect causes a large increase in memory usage, you may want to consider modifying the effect or eliminating it altogether.

The `System.ime` property is a reference to the currently installed Input Method Editor (IME). This property allows you to listen for `imeComposition` events (`flash.events.IMEEvent.IME_COMPOSITION`) by using the `addEventListener()` method.

The third property in the `System` class is `useCodePage`. When `useCodePage` is set to `true`, the runtime uses the traditional code page of the operating system to load external text files. If you set this property to `false`, you tell the runtime to interpret the external file as Unicode.

If you set `System.useCodePage` to `true`, remember that the traditional code page of the operating system must include the characters used in your external text file in order for the text to display. For example, if you load an external text file that contains Chinese characters, those characters cannot display on a system that uses the English Windows code page because that code page does not include Chinese characters.

To ensure that users on all platforms can view the external text files that are used in your application, you should encode all external text files as Unicode and leave `System.useCodePage` set to `false` by default. This way, the runtime interprets the text as Unicode.

Using the Capabilities class

Flash Player 9 and later, Adobe AIR 1.0 and later

The `Capabilities` class allows developers to determine the environment in which an application is being run. Using various properties of the `Capabilities` class, you can find out the resolution of the user's system, whether the user's system supports accessibility software, and the language of the user's operating system, as well as the currently installed version of the Flash runtime.

By checking the properties in the Capabilities class, you can customize your application to work best with the specific user's environment. For example, by checking the `Capabilities.screenResolutionX` and `Capabilities.screenResolutionY` properties, you can determine the display resolution the user's system is using and decide which video size may be most appropriate. Or you can check the `Capabilities.hasMP3` property to see if the user's system supports mp3 playback before attempting to load an external mp3 file.

The following code uses a regular expression to parse the Flash runtime version that the client is using:

```
var versionString = air.Capabilities.version;
var pattern = /^(\w*) (\d*), (\d*), (\d*), (\d*)$/;
var result = pattern.exec(versionString);
if (result != null)
{
    air.trace("input: " + result.input);
    air.trace("platform: " + result[1]);
    air.trace("majorVersion: " + result[2]);
    air.trace("minorVersion: " + result[3]);
    air.trace("buildNumber: " + result[4]);
    air.trace("internalBuildNumber: " + result[5]);
}
else
{
    air.trace("Unable to match RegExp.");
}
```

Chapter 20: AIR application invocation and termination

Adobe AIR 1.0 and later

This section discusses the ways in which an installed Adobe® AIR® application can be invoked, as well as options and considerations for closing a running application.

Note: The `NativeApplication`, `InvokeEvent`, and `BrowserInvokeEvent` objects are only available to SWF content running in the AIR application sandbox. SWF content running in the Flash Player runtime, within the browser or the standalone player (projector), or in an AIR application outside the application sandbox, cannot access these classes.

For a quick explanation and code examples of invoking and terminating AIR applications, see the following quick start articles on the Adobe Developer Connection:

- [Startup Options](#)
- [Startup Options](#)

More Help topics

[air.NativeApplication](#)

[flash.events.InvokeEvent](#)

[flash.events.BrowserInvokeEvent](#)

Application invocation

Adobe AIR 1.0 and later

An AIR application is invoked when the user (or the operating system):

- Launches the application from the desktop shell.
- Uses the application as a command on a command line shell.
- Opens a type of file for which the application is the default opening application.
- (Mac OS X) clicks the application icon in the dock/taskbar (whether or not the application is currently running).
- Chooses to launch the application from the installer (either at the end of a new installation process, or after double-clicking the AIR file for an already installed application).
- Begins an update of an AIR application when the installed version has signaled that it is handling application updates itself (by including a `<customUpdateUI>true</customUpdateUI>` declaration in the application descriptor file).
- Visits a web page hosting a Flash badge or application that calls `com.adobe.air.AIR.launchApplication()` method specifying the identifying information for the AIR application. (The application descriptor must also include a `<allowBrowserInvocation>true</allowBrowserInvocation>` declaration for browser invocation to succeed.)

Whenever an AIR application is invoked, AIR dispatches an `InvokeEvent` object of type `invoke` through the singleton `NativeApplication` object. To allow an application time to initialize itself and register an event listener, `invoke` events are queued instead of discarded. As soon as a listener is registered, all the queued events are delivered.

Note: *When an application is invoked using the browser invocation feature, the `NativeApplication` object only dispatches an `invoke` event if the application is not already running.*

To receive `invoke` events, call the `addEventListener()` method of the `NativeApplication` object (`NativeApplication.nativeApplication`). When an event listener registers for an `invoke` event, it also receives all `invoke` events that occurred before the registration. Queued `invoke` events are dispatched one at a time on a short interval after the call to `addEventListener()` returns. If a new `invoke` event occurs during this process, it may be dispatched before one or more of the queued events. This event queuing allows you to handle any `invoke` events that have occurred before your initialization code executes. Keep in mind that if you add an event listener later in execution (after application initialization), it will still receive all `invoke` events that have occurred since the application started.

Only one instance of an AIR application is started. When an already running application is invoked again, AIR dispatches a new `invoke` event to the running instance. It is the responsibility of an AIR application to respond to an `invoke` event and take the appropriate action (such as opening a new document window).

An `InvokeEvent` object contains any arguments passed to the application, as well as the directory from which the application has been invoked. If the application was invoked because of a file-type association, then the full path to the file is included in the `arguments` array. Likewise, if the application was invoked because of an application update, the full path to the update AIR file is provided.

When multiple files are opened in one operation a single `InvokeEvent` object is dispatched on Mac OS X. Each file is included in the `arguments` array. On Windows and Linux, a separate `InvokeEvent` object is dispatched for each file.

Your application can handle `invoke` events by registering a listener with its `NativeApplication` object:

```
air.NativeApplication.nativeApplication.addEventListener(air.InvokeEvent.INVOKE,  
onInvokeEvent);
```

And defining an event listener:

```
var arguments;  
var currentDir;  
function onInvokeEvent(invocation) {  
    arguments = invocation.arguments;  
    currentDir = invocation.currentDirectory;  
}
```

Capturing command line arguments

Adobe AIR 1.0 and later

The command line arguments associated with the invocation of an AIR application are delivered in the `InvokeEvent` object dispatched by the `NativeApplication` object. The `InvokeEvent` `arguments` property contains an array of the arguments passed by the operating system when an AIR application is invoked. If the arguments contain relative file paths, you can typically resolve the paths using the `currentDirectory` property.

The arguments passed to an AIR program are treated as white-space delimited strings, unless enclosed in double quotes:

Arguments	Array
tick tock	{tick,tock}
tick "tick tock"	{tick,tick tock}
"tick" "tock"	{tick,tock}
\"tick\" \"tock\"	{\"tick\",\"tock\"}

The `currentDirectory` property of an `InvokeEvent` object contains a `File` object representing the directory from which the application was launched.

When an application is invoked because a file of a type registered by the application is opened, the native path to the file is included in the command line arguments as a string. (Your application is responsible for opening or performing the intended operation on the file.) Likewise, when an application is programmed to update itself (rather than relying on the standard AIR update user interface), the native path to the AIR file is included when a user double-clicks an AIR file containing an application with a matching application ID.

You can access the file using the `resolve()` method of the `currentDirectory` `File` object:

```
if((invokeEvent.currentDirectory != null)&&(invokeEvent.arguments.length > 0)){
    dir = invokeEvent.currentDirectory;
    fileToOpen = dir.resolvePath(invokeEvent.arguments[0]);
}
```

You should also validate that an argument is indeed a path to a file.

Example: Invocation event log

Adobe AIR 1.0 and later

The following example demonstrates how to register listeners for and handle the `invoke` event. The example logs all the invocation events received and displays the current directory and command line arguments.

Note: This example uses the `AIRAliases.js` file, which you can find in the `frameworks` folder of the SDK.

```
<html>
<head>
<title>Invocation Event Log</title>
<script src="AIRAliases.js" />
<script type="text/javascript">
function appLoad() {
    air.trace("Invocation Event Log.");
    air.NativeApplication.nativeApplication.addEventListener(
        air.InvokeEvent.INVOKE, onInvoke);
}

function onInvoke(invokeEvent) {
    logEvent("Invoke event received.");
    if (invokeEvent.currentDirectory) {
        logEvent("Current directory=" + invokeEvent.currentDirectory.nativePath);
    } else {
        logEvent("--no directory information available--");
    }

    if (invokeEvent.arguments.length > 0) {
        logEvent("Arguments: " + invokeEvent.arguments.toString());
    }
}
```



```
    } else {
        logEvent("--no arguments--");
    }
}

function logEvent(message) {
    var logger = document.getElementById('log');
    var line = document.createElement('p');
    line.innerHTML = message;
    logger.appendChild(line);
    air.trace(message);
}

window.unload = function() {
    air.NativeApplication.nativeApplication.removeEventListener(
        air.InvokeEvent.INVOKE, onInvoke);
}
</script>
</head>

<body onLoad="appLoad();" >
    <div id="log"/>
</body>
</html>
```

Invoking an AIR application on user login

Adobe AIR 1.0 and later

An AIR application can be set to launch automatically when the current user logs in by setting the `NativeApplication.startAtLogin` property to `true`. Once set, the application automatically starts whenever the user logs in. It continues to launch at login until the setting is changed to `false`, the user manually changes the setting through the operating system, or the application is uninstalled. Launching at login is a run-time setting. The setting only applies to the current user. The application must be installed to successfully set the `startAtLogin` property to `true`. An error is thrown if the property is set when an application is not installed (such as when it is launched with ADL).

Note: *The application does not launch when the computer system starts. It launches when the user logs in.*

To determine whether an application has launched automatically or as a result of a user action, you can examine the `reason` property of the `InvokeEvent` object. If the property is equal to `InvokeEventReason.LOGIN`, then the application started automatically. For any other invocation path, the `reason` property equals `InvokeEventReason.STANDARD`. To access the `reason` property, your application must target AIR 1.5.1 (by setting the correct namespace value in the application descriptor file).

The following, simplified application uses the `InvokeEvent` reason property to decide how to behave when an invoke event occurs. If the reason property is "login", then the application remains in the background. Otherwise, it makes the main application visible. An application using this pattern typically starts at login so that it can carry out background processing or event monitoring and opens a window in response to a user-triggered invoke event.

```
<html>
<head>
<script src="AIRAliases.js"></script>
<script language="javascript">
try
{
    air.NativeApplication.nativeApplication.startAtLogin = true;
}
catch ( e )
{
    air.trace( "Cannot set startAtLogin: " + e.message );
}

air.NativeApplication.nativeApplication.addEventListener( air.InvokeEvent.INVOKE, onInvoke );

function onInvoke( event )
{
    if( event.reason == air.InvokeEventReason.LOGIN )
    {
        //do background processing...
        air.trace( "Running in background..." );
    }
    else
    {
        window.nativeWindow.activate();
    }
}
</script>
</head>
<body>
</body>
</html>
```

Note: To see the difference in behavior, package and install the application. The `startAtLogin` property can only be set for installed applications.

Invoking an AIR application from the browser

Adobe AIR 1.0 and later

Using the browser invocation feature, a web site can launch an installed AIR application to be launched from the browser. Browser invocation is only permitted if the application descriptor file sets `allowBrowserInvocation` to `true`:

```
<allowBrowserInvocation>true</allowBrowserInvocation>
```

When the application is invoked via the browser, the application's `NativeApplication` object dispatches a `BrowserInvokeEvent` object.

To receive `BrowserInvokeEvent` events, call the `addEventListener()` method of the `NativeApplication` object (`NativeApplication.nativeApplication`) in the AIR application. When an event listener registers for a `BrowserInvokeEvent` event, it also receives all `BrowserInvokeEvent` events that occurred before the registration. These events are dispatched after the call to `addEventListener()` returns, but not necessarily before other `BrowserInvokeEvent` events that might be received after registration. This allows you to handle `BrowserInvokeEvent` events that have occurred before your initialization code executes (such as when the application was initially invoked from the browser). Keep in mind that if you add an event listener later in execution (after application initialization) it still receives all `BrowserInvokeEvent` events that have occurred since the application started.

The `BrowserInvokeEvent` object includes the following properties:

Property	Description
<code>arguments</code>	An array of arguments (strings) to pass to the application.
<code>isHTTPS</code>	Whether the content in the browser uses the <code>https</code> URL scheme (<code>true</code>) or not (<code>false</code>).
<code>isUserEvent</code>	Whether the browser invocation resulted in a user event (such as a mouse click). In AIR 1.0, this is always set to <code>true</code> ; AIR requires a user event to the browser invocation feature.
<code>sandboxType</code>	The sandbox type for the content in the browser. Valid values are defined the same as those that can be used in the <code>Security.sandboxType</code> property, and can be one of the following: <ul style="list-style-type: none">• <code>Security.APPLICATION</code> — The content is in the application security sandbox.• <code>Security.LOCAL_TRUSTED</code> — The content is in the local-with-filesystem security sandbox.• <code>Security.LOCAL_WITH_FILE</code> — The content is in the local-with-filesystem security sandbox.• <code>Security.LOCAL_WITH_NETWORK</code> — The content is in the local-with-networking security sandbox.• <code>Security.REMOTE</code> — The content is in a remote (network) domain.
<code>securityDomain</code>	The security domain for the content in the browser, such as <code>"www.adobe.com"</code> or <code>"www.example.org"</code> . This property is only set for content in the remote security sandbox (for content from a network domain). It is not set for content in a local or application security sandbox.

If you use the browser invocation feature, be sure to consider security implications. When a web site launches an AIR application, it can send data via the `arguments` property of the `BrowserInvokeEvent` object. Be careful using this data in any sensitive operations, such as file or code loading APIs. The level of risk depends on what the application is doing with the data. If you expect only a specific web site to invoke the application, the application should check the `securityDomain` property of the `BrowserInvokeEvent` object. You can also require the web site invoking the application to use HTTPS, which you can verify by checking the `isHTTPS` property of the `BrowserInvokeEvent` object.

The application should validate the data passed in. For example, if an application expects to be passed URLs to a specific domain, it should validate that the URLs really do point to that domain. This can prevent an attacker from tricking the application into sending it sensitive data.

No application should use `BrowserInvokeEvent` arguments that might point to local resources. For example, an application should not create `File` objects based on a path passed from the browser. If remote paths are expected to be passed from the browser, the application should ensure that the paths do not use the `file://` protocol instead of a remote protocol.

Application termination

Adobe AIR 1.0 and later

The quickest way to terminate an application is to call the `NativeApplication.exit()` method. This works fine when your application has no data to save or external resources to clean up. Calling `exit()` closes all windows and then terminates the application. However, to allow windows or other components of your application to interrupt the termination process, perhaps to save vital data, dispatch the proper warning events before calling `exit()`.

Another consideration in gracefully shutting down an application is providing a single execution path, no matter how the shut-down process starts. The user (or operating system) can trigger application termination in the following ways:

- By closing the last application window when `NativeApplication.nativeApplication.autoExit` is `true`.
- By selecting the application exit command from the operating system; for example, when the user chooses the exit application command from the default menu. (This only happens on Mac OS; Windows and Linux do not provide an application exit command through system chrome.)
- By shutting down the computer.

When an exit command is mediated through the operating system by one of these routes, the `NativeApplication` dispatches an `exiting` event. If no listeners cancel the `exiting` event, any open windows are closed. Each window dispatches a `closing` and then a `close` event. If any of the windows cancel the `closing` event, the shut-down process stops.

If the order of window closure is an issue for your application, listen for the `exiting` event from the `NativeApplication` and close the windows in the proper order yourself. You might need to do this, for example, if you have a document window with tool palettes. It could be inconvenient, or worse, if the system closed the palettes, but the user decided to cancel the exit command to save some data. On Windows, the only time you will get the `exiting` event is after closing the last window (when the `autoExit` property of the `NativeApplication` object is set to `true`).

To provide consistent behavior on all platforms, whether the exit sequence is initiated via operating system chrome, menu commands, or application logic, observe the following good practices for exiting the application:

- 1 Always dispatch an `exiting` event through the `NativeApplication` object before calling `exit()` in application code and check that another component of your application doesn't cancel the event.

```
function applicationExit(){
    var exitingEvent = new air.Event(air.Event.EXITING, false, true);
    air.NativeApplication.nativeApplication.dispatchEvent(exitingEvent);
    if (!exitingEvent.isDefaultPrevented()) {
        air.NativeApplication.nativeApplication.exit();
    }
}
```

- 2 Listen for the application `exiting` event from the `NativeApplication.nativeApplication` object and, in the handler, close any windows (dispatching a `closing` event first). Perform any needed clean-up tasks, such as saving application data or deleting temporary files, after all windows have been closed. Only use synchronous methods during cleanup to ensure that they finish before the application quits.

If the order in which your windows are closed doesn't matter, then you can loop through the `NativeApplication.nativeApplication.openedWindows` array and close each window in turn. If order *does* matter, provide a means of closing the windows in the correct sequence.

```
function onExiting(exitingEvent) {
    var winClosingEvent;
    for (var i = 0; i < air.NativeApplication.nativeApplication.openedWindows.length; i++) {
        var win = air.NativeApplication.nativeApplication.openedWindows[i];
        winClosingEvent = new air.Event(air.Event.CLOSING, false, true);
        win.dispatchEvent(winClosingEvent);
        if (!winClosingEvent.isDefaultPrevented()) {
            win.close();
        } else {
            exitingEvent.preventDefault();
        }
    }

    if (!exitingEvent.isDefaultPrevented()) {
        //perform cleanup
    }
}
```

- 3 Windows should always handle their own clean up by listening for their own `closing` events.
- 4 Only use one `exiting` listener in your application since handlers called earlier cannot know whether subsequent handlers will cancel the `exiting` event (and it would be unwise to rely on the order of execution).

Chapter 21: Working with AIR runtime and operating system information

Adobe AIR 1.0 and later

This section discusses ways that an AIR application can manage operating system file associations, detect user activity, and get information about the Adobe® AIR® runtime.

More Help topics

[flash.desktop.NativeApplication](#)

Managing file associations

Adobe AIR 1.0 and later

Associations between your application and a file type must be declared in the application descriptor. During the installation process, the AIR application installer associates the AIR application as the default opening application for each of the declared file types, unless another application is already the default. The AIR application install process does not override an existing file type association. To take over the association from another application, call the `NativeApplication.setAsDefaultApplication()` method at run time.

It is a good practice to verify that the expected file associations are in place when your application starts up. This is because the AIR application installer does not override existing file associations, and because file associations on a user's system can change at any time. When another application has the current file association, it is also a polite practice to ask the user before taking over an existing association.

The following methods of the `NativeApplication` class let an application manage file associations. Each of the methods takes the file type extension as a parameter:

Method	Description
<code>isSetAsDefaultApplication()</code>	Returns true if the AIR application is currently associated with the specified file type.
<code>setAsDefaultApplication()</code>	Creates the association between the AIR application and the open action of the file type.
<code>removeAsDefaultApplication()</code>	Removes the association between the AIR application and the file type.
<code>getDefaultApplication()</code>	Reports the path of the application that is currently associated with the file type.

AIR can only manage associations for the file types originally declared in the application descriptor. You cannot get information about the associations of a non-declared file type, even if a user has manually created the association between that file type and your application. Calling any of the file association management methods with the extension for a file type not declared in the application descriptor causes the application to throw a runtime exception.

Getting the runtime version and patch level

Adobe AIR 1.0 and later

The `NativeApplication` object has a `runtimeVersion` property, which is the version of the runtime in which the application is running (a string, such as "1.0.5"). The `NativeApplication` object also has a `runtimePatchLevel` property, which is the patch level of the runtime (a number, such as 2960). The following code uses these properties:

```
air.trace(air.NativeApplication.nativeApplication.runtimeVersion);  
air.trace(air.NativeApplication.nativeApplication.runtimePatchLevel);
```

Detecting AIR capabilities

Adobe AIR 1.0 and later

For a file that is bundled with the Adobe AIR application, the `Security.sandboxType` property is set to the value defined by the `Security.APPLICATION` constant. You can load content (which may or may not contain APIs specific to AIR) based on whether a file is in the Adobe AIR security sandbox, as illustrated in the following code:

```
if (window.runtime)  
{  
    if (air.Security.sandboxType == air.Security.APPLICATION)  
    {  
        alert("In AIR application security sandbox.");  
    }  
    else  
    {  
        alert("Not in AIR application security sandbox.")  
    }  
}  
else  
{  
    alert("Not in the Adobe AIR runtime.")  
}
```

All resources that are not installed with the AIR application are put in security sandboxes based on their domains of origin. For example, content served from `www.example.com` is put in a security sandbox for that domain.

You can check if the `window.runtime` property is set to see if content is executing in the runtime.

For more information, see [“AIR security”](#) on page 69.

Tracking user presence

Adobe AIR 1.0 and later

The `NativeApplication` object dispatches two events that help you detect when a user is actively using a computer. If no mouse or keyboard activity is detected in the interval determined by the `NativeApplication.idleThreshold` property, the `NativeApplication` dispatches a `userIdle` event. When the next keyboard or mouse input occurs, the `NativeApplication` object dispatches a `userPresent` event. The `idleThreshold` interval is measured in seconds and has a default value of 300 (5 minutes). You can also get the number of seconds since the last user input from the `NativeApplication.nativeApplication.lastUserInput` property.

The following lines of code set the idle threshold to 2 minutes and listen for both the `userIdle` and `userPresent` events:

```
air.NativeApplication.nativeApplication.idleThreshold = 120;
air.NativeApplication.nativeApplication.addEventListener(air.Event.USER_IDLE, function(event) {
    air.trace("Idle");
});
air.NativeApplication.nativeApplication.addEventListener(air.Event.USER_PRESENT,
function(event) {
    air.trace("Present");
});
```

Note: Only a single `userIdle` event is dispatched between any two `userPresent` events.

Chapter 22: Sockets

Flash Player 9 and later, Adobe AIR 1.0 and later

A socket is a type of network connection established between two computer processes. Typically, the processes are running on two different computers attached to the same Internet Protocol (IP) network. However, the connected processes can be running on the same computer using the special “local host” IP address.

Adobe Flash Player supports client-side Transport Control Protocol (TCP) sockets. A Flash Player application can connect to another process acting as a socket server, but cannot accept incoming connection requests from other processes. In other words, a Flash Player application can connect to a TCP server, but cannot serve as one.

The Flash Player API also includes the XMLSocket class. The XMLSocket class uses a Flash Player-specific protocol that allows you to exchange XML messages with a server that understands that protocol. The XMLSocket class was introduced in ActionScript 1 and is still supported to provide backward compatibility. In general, the Socket class should be used for new applications unless you are connecting to a server specifically created to communicate with Flash XMLSockets.

Adobe AIR adds several additional classes for socket-based network programming. AIR applications can act as TCP socket servers with the ServerSocket class and can connect to socket servers requiring SSL or TLS security with the SecureSocket class. AIR applications can also send and receive Universal Datagram Protocol (UDP) messages with the DatagramSocket class.

TCP sockets

Flash Player 9 and later, Adobe AIR 1.0 and later

The Transmission Control Protocol (TCP) provides a way to exchange messages over a persistent network connection. TCP guarantees that any messages sent arrive in the correct order (barring major network problems). TCP connections require a “client” and a “server.” Flash Player can create client sockets. Adobe AIR can, additionally, create server sockets.

The following ActionScript APIs provide TCP connections:

- Socket — allows a client application to connect to a server. The Socket class cannot listen for incoming connections.
- SecureSocket (AIR) — allows a client application to connect to a trusted server and engage in encrypted communications.
- ServerSocket (AIR) — allows an application to listen for incoming connections and act as a server.
- XMLSocket — allows a client application to connect to an XMLSocket server.

Binary client sockets

Flash Player 9 and later, Adobe AIR 1.0 and later


A binary socket connection is similar to an XML socket except that the client and server are not limited to exchanging XML messages. Instead, the connection can transfer data as binary information. Thus, you can connect to a wider range of services, including mail servers (POP3, SMTP, and IMAP), and news servers (NNTP).

Socket class

Flash Player 9 and later, Adobe AIR 1.0 and later

The `Socket` class enables you to make socket connections and to read and write raw binary data. The `Socket` class is useful for interoperating with servers that use binary protocols. By using binary socket connections, you can write code that interacts with several different Internet protocols, such as POP3, SMTP, IMAP, and NNTP. This interaction, in turn, enables your applications to connect to mail and news servers.

Flash Player can interface with a server by using the binary protocol of that server directly. Some servers use the big-endian byte order, and some use the little-endian byte order. Most servers on the Internet use the big-endian byte order because “network byte order” is big-endian. The little-endian byte order is popular because the Intel® x86 architecture uses it. You should use the endian byte order that matches the byte order of the server that is sending or receiving data. All operations that are performed by the `IDataInput` and `IDataOutput` interfaces, and the classes that implement those interfaces (`ByteArray`, `Socket`, and `URLStream`), are encoded by default in big-endian format; that is, with the most significant byte first. This default byte order was chosen to match Java and the official network byte order. To change whether big-endian or little-endian byte order is used, you can set the `endian` property to `Endian.BIG_ENDIAN` or `Endian.LITTLE_ENDIAN`.

 *The `Socket` class inherits all the methods defined by the `IDataInput` and `IDataOutput` interfaces (located in the `flash.utils` package). Those methods must be used to write to and read from the `Socket`.*

For more information, see:

- `Socket`
- `IDataInput`
- `IDataOutput`
- `socketData` event

Secure client sockets (AIR)

Adobe AIR 2 and later

You can use the `SecureSocket` class to connect to socket servers that use Secure Sockets Layer version 4 (SSLv4) or Transport Layer Security version 1 (TLSv1). A secure socket provides three benefits: server authentication, data integrity, and message confidentiality. The runtime authenticates a server using the server certificate and its relationship to the root or intermediate certificate authority certificates in the user's trust store. The runtime relies on the cryptography algorithms used by the SSL and TLS protocol implementations to provide data integrity and message confidentiality.

When you connect to a server using the `SecureSocket` object, the runtime validates the server certificate using the certificate trust store. On Windows and Mac, the operating system provides the trust store. On Linux, the runtime provides its own trust store.

If the server certificate is not valid or not trusted, the runtime dispatches an `ioError` event. You can check the `serverCertificateStatus` property of the `SecureSocket` object to determine why validation failed. No provision is provided for communicating with a server that does not have a valid and trusted certificate.

The `CertificateStatus` class defines string constants that represent the possible validation results:

- `Expired`—the certificate expiration date has passed.
- `Invalid`—there are a number of reasons that a certificate can be invalid. For example, the certificate could have been altered, corrupted, or it could be the wrong type of certificate.

Sockets

- Invalid chain—one or more of the certificates in the server's chain of certificates are invalid.
- Principal mismatch—the host name of the server and the certificate common name do not match. In other words, the server is using the wrong certificate.
- Revoked—the issuing certificate authority has revoked the certificate.
- Trusted—the certificate is valid and trusted. A SecureSocket object can only connect to a server that uses a valid, trusted certificate.
- Unknown—the SecureSocket object has not validated the certificate yet. The `serverCertificateStatus` property has this status value before you call `connect()` and before either a `connect` or an `ioError` event is dispatched.
- Untrusted signers—the certificate does not “chain” to a trusted root certificate in the trust store of the client computer.

Communicating with a SecureSocket object requires a server that uses a secure protocol and has a valid, trusted certificate. In other respects, using a SecureSocket object is the same as using a Socket object.

The SecureSocket object is not supported on all platforms. Use the SecureSocket class `isSupported` property to test whether the runtime supports use of the SecureSocket object on the current client computer.

For more information, see:

- SecureSocket
- CertificateStatus
- IDataInput
- IDataOutput
- socketData event

XML sockets

Flash Player 9 and later, Adobe AIR 1.0 and later

An XML socket lets you create a connection to a remote server that remains open until explicitly closed. You can exchange string data, such as XML, between the server and client. A benefit of using an XML socket server is that the client does not need to explicitly request data. The server can send data without waiting for a request and can send data to every connected client connected.

In Flash Player, and in Adobe AIR content outside the application sandbox, XML socket connections require the presence of a socket policy file on the target server. For more information, see [Website controls \(policy files\)](#) and [Connecting to sockets](#).

The XMLSocket class cannot tunnel through firewalls automatically because, unlike the Real-Time Messaging Protocol (RTMP), XMLSocket has no HTTP tunneling capability. If you need to use HTTP tunneling, consider using [Flash Remoting](#) or [Flash Media Server](#) (which supports RTMP) instead.

The following restrictions apply to how and where content in Flash Player or in an AIR application outside of the application security sandbox can use an `XMLSocket` object to connect to the server:

- For content outside of the application security sandbox, the `XMLSocket.connect()` method can connect only to TCP port numbers greater than or equal to 1024. One consequence of this restriction is that the server daemons that communicate with the `XMLSocket` object must also be assigned to port numbers greater than or equal to 1024. Port numbers below 1024 are often used by system services such as FTP (21), Telnet (23), SMTP (25), HTTP (80), and POP3 (110), so `XMLSocket` objects are barred from these ports for security reasons. The port number restriction limits the possibility that these resources will be inappropriately accessed and abused.
- For content outside of the application security sandbox, the `XMLSocket.connect()` method can connect only to computers in the same domain where the content resides. (This restriction is identical to the security rules for `URLLoader.load()`.) To connect to a server daemon running in a domain other than the one where the content resides, you can create a cross-domain policy file on the server that allows access from specific domains. For details on cross-domain policy files, see “[AIR security](#)” on page 69.

Note: *Setting up a server to communicate with the `XMLSocket` object can be challenging. If your application does not require real-time interactivity, use the `URLLoader` class instead of the `XMLSocket` class.*

You can use the `XMLSocket.connect()` and `XMLSocket.send()` methods of the `XMLSocket` class to transfer XML to and from a server over a socket connection. The `XMLSocket.connect()` method establishes a socket connection with a web server port. The `XMLSocket.send()` method passes an XML object to the server specified in the socket connection.

When you invoke the `XMLSocket.connect()` method, the application opens a TCP/IP connection to the server and keeps that connection open until one of the following occurs:

- The `XMLSocket.close()` method of the `XMLSocket` class is called.
- No more references to the `XMLSocket` object exist.
- The connection is broken (for example, the modem disconnects).

Connecting to a server with the `XMLSocket` class

Flash Player 9 and later, Adobe AIR 1.0 and later

To create a socket connection, you must create a server-side application to wait for the socket connection request and send a response to the Flash Player or AIR application. This type of server-side application can be written in AIR or in another programming language such as Java, Python, or Perl. To use the `XMLSocket` class, the server computer must run a daemon that understands the simple protocol used by the `XMLSocket` class:

- XML messages are sent over a full-duplex TCP/IP stream socket connection.
- Each XML message is a complete XML document, terminated by a zero (0) byte.
- An unlimited number of XML messages can be sent and received over a single `XMLSocket` connection.

Server sockets

Adobe AIR 2 and later

Use the `ServerSocket` class to allow other processes to connect to your application using a Transport Control Protocol (TCP) socket. The connecting process can be running on the local computer or on another network-connected computer. When a `ServerSocket` object receives a connection request, it dispatches a `connect` event. The `ServerSocketConnectEvent` object dispatched with the event contains a `Socket` object. You can use this `Socket` object for subsequent communication with the other process.

To listen for incoming socket connections:

- 1 Create a `ServerSocket` object and bind it to a local port
- 2 Add event listeners for the `connect` event
- 3 Call the `listen()` method
- 4 Respond to the `connect` event, which provides a `Socket` object for each incoming connection

The `ServerSocket` object continues to listen for new connections until you call the `close()` method.

The following code example illustrates how to create a socket server application. The example listens for incoming connections on port 8087. When a connection is received, the example sends a message (the string "Connected.") to the client socket. Thereafter, the server echoes any messages received back to the client.

```
<html>
<head>
<script src="AIRAliases.js"></script>
<script language="javascript">
    var serverSocket;
    var clientSockets = new Array();
    function startServer()
    {
        try
        {
            // Create the server socket
            serverSocket = new air.ServerSocket();

            // Add the event listener
            serverSocket.addEventListener( air.Event.CONNECT, connectHandler );
            serverSocket.addEventListener( air.Event.CLOSE, onClose );

            // Bind to local port 8087
            serverSocket.bind( 8087, "127.0.0.1" );

            // Listen for connections
            serverSocket.listen();
            air.trace( "Listening on " + serverSocket.localPort );
        }
        catch( e )
        {
            air.trace( e );
        }
    }
    function connectHandler( event )
    {
        //The socket is provided by the event object
        var socket = event.socket;
        clientSockets.push( socket );

        socket.addEventListener( air.ProgressEvent.SOCKET_DATA, socketDataHandler);
        socket.addEventListener( air.Event.CLOSE, onClientClose );
        socket.addEventListener( air.IOErrorEvent.IO_ERROR, onIOError );

        //Send a connect message
        socket.writeUTFBytes("Connected.");
        socket.flush();
    }
</script>
</head>
</html>
```

```
        air.trace( "Sending connect message" );
    }

    function socketDataHandler( event )
    {
        var socket = event.target

        //Read the message from the socket
        var message = socket.readUTFBytes( socket.bytesAvailable );
        air.trace( "Received: " + message);
        // Echo the received message back to the sender
        message = "Echo -- " + message;
        socket.writeUTFBytes( message );
        socket.flush();
        air.trace( "Sending: " + message );
    }

    function onClientClose( event )
    {
        air.trace( "Connection to client closed." );
        //Should also remove from clientSockets array...
    }
    function onIOError( errorEvent )
    {
        air.trace( "IOError: " + errorEvent.text );
    }
    function onClose( event )
    {
        air.trace( "Server socket closed by OS." );
    }
}
</script>
</head>
<body onload="startServer()">
</body>
</html>
```

For more information, see:

- [ServerSocket](#)
- [ServerSocketConnectEvent](#)
- [Socket](#)

UDP sockets (AIR)

Adobe AIR 2 and later

The Universal Datagram Protocol (UDP) provides a way to exchange messages over a stateless network connection. UDP provides no guarantees that messages are delivered in order or even that messages are delivered at all. With UDP, the operating system's network code usually spends less time marshaling, tracking, and acknowledging messages. Thus, UDP messages typically arrive at the destination application with a shorter delay than do TCP messages.

UDP socket communication is helpful when you must send real-time information such as position updates in a game, or sound packets in an audio chat application. In such applications, some data loss is acceptable, and low transmission latency is more important than guaranteed arrival. For almost all other purposes, TCP sockets are a better choice.

Your AIR application can send and receive UDP messages with the `DatagramSocket` and `DatagramSocketDataEvent` classes. To send or receive a UDP message:

- 1 Create a `DatagramSocket` object
- 2 Add an event listener for the `data` event
- 3 Bind the socket to a local IP address and port using the `bind()` method
- 4 Send messages by calling the `send()` method, passing in the IP address and port of the target computer
- 5 Receive messages by responding to the `data` event. The `DatagramSocketDataEvent` object dispatched for this event contains a `ByteArray` object containing the message data.

The following code example illustrates how an application can send and receive UDP messages. The example sends a single message containing the string, "Hello.", to the target computer. It also traces the contents of any messages received.

```
<html>
<head>
<script src="AIRAliases.js"></script>
<script language="javascript">
    var datagramSocket;

    //The IP and port for this computer
    var localIP = "192.168.0.1";
    var localPort = 55555;

    //The IP and port for the target computer
    var targetIP = "192.168.0.2";
    var targetPort = 55555;

    function createDatagramSocket()
    {
        //Create the socket
        datagramSocket = new air.DatagramSocket();
        datagramSocket.addEventListener( air.DatagramSocketDataEvent.DATA, dataReceived );

        //Bind the socket to the local network interface and port
        datagramSocket.bind( localPort, localIP );

        //Listen for incoming datagrams
        datagramSocket.receive();
    }
</script>
</head>
</html>
```

```
//Create a message in a ByteArray
var data = new air.ByteArray();
data.writeUTFBytes("Hello.");

//Send the datagram message
datagramSocket.send( data, 0, 0, targetIP, targetPort);
}

function dataReceived( event )
{
    //Read the data from the datagram
    air.trace("Received from " + event.srcAddress + ":" + event.srcPort + "> " +
        event.data.readUTFBytes( event.data.bytesAvailable ) );
}
</script>
</head>
<body onload="createDatagramSocket()" >
</body>
</html>
```

Keep in mind the following considerations when using UDP sockets:

- A single packet of data cannot be larger than the smallest maximum transmission unit (MTU) of the network interface or any network nodes between the sender and the recipient. All of the data in the ByteArray object passed to the send() method is sent as a single packet. (In TCP, large messages are broken up into separate packets.)
- There is no handshaking between the sender and the target. Messages are discarded without error if the target does not exist or does not have an active listener at the specified port.
- When you use the connect() method, messages sent from other sources are ignored. A UDP connection provides convenient packet filtering only. It does not mean that there is necessarily a valid, listening process at the target address and port.
- UDP traffic can swamp a network. Network administrators might need to implement quality-of-service controls if network congestion occurs. (TCP has built-in traffic control to reduce the impact of network congestion.)

For more information, see:

- DatagramSocket
- DatagramSocketDataEvent
- ByteArray

IPv6 addresses

Flash Player 9 and later, Adobe AIR 1.0 and later

Flash Player 9.0.115.0 and later support IPv6 (Internet Protocol version 6). IPv6 is a version of Internet Protocol that supports 128-bit addresses (an improvement on the earlier IPv4 protocol that supports 32-bit addresses). You might need to activate IPv6 on your networking interfaces. For more information, see the Help for the operating system hosting the data.

If IPv6 is supported on the hosting system, you can specify numeric IPv6 literal addresses in URLs enclosed in brackets ([]), as in the following:


```
[2001:db8:ccc3:ffff:0:444d:555e:666f]
```

Flash Player returns literal IPv6 values, according to the following rules:

- Flash Player returns the long form of the string for IPv6 addresses.
- The IP value has no double-colon abbreviations.
- Hexadecimal digits are lowercase only.
- IPv6 addresses are enclosed in square brackets ([]).
- Each address quartet is output as 0 to 4 hexadecimal digits, with the leading zeros omitted.
- An address quartet of all zeros is output as a single zero (not a double colon) except as noted in the following list of exceptions.

The IPv6 values that Flash Player returns have the following exceptions:

- An unspecified IPv6 address (all zeros) is output as [::].
- The loopback or localhost IPv6 address is output as [::1].
- IPv4 mapped (converted to IPv6) addresses are output as [::ffff:a.b.c.d], where a.b.c.d is a typical IPv4 dotted-decimal value.
- IPv4 compatible addresses are output as [::a.b.c.d], where a.b.c.d is a typical IPv4 dotted-decimal value.

Chapter 23: HTTP communications

Flash Player 9 and later, Adobe AIR 1.0 and later

Adobe® AIR® and Adobe® Flash® Player applications can communicate with HTTP-based servers to load data, images, video and to exchange messages.

This topic describes the AIR networking and communication API—functionality uniquely provided to applications running in the runtime. It does not describe all networking and communications functionality inherent to HTML and JavaScript that would function in a web browser (such as the details of using the XMLHttpRequest class).

More Help topics

[flash.net.URLLoader](#)

[flash.net.URLStream](#)

[flash.net.URLRequest](#)

[flash.net.URLRequestDefaults](#)

[flash.net.URLRequestHeader](#)

[flash.net.URLRequestMethod](#)

[flash.net.URLVariables](#)

Loading external data

Flash Player 9 and later, Adobe AIR 1.0 and later

The AIR runtime includes mechanisms for loading data from external sources. Those sources can provide static content such as text files, or dynamic content, such as content generated by a web script. The data can be formatted in various ways, and the runtime provides functionality for decoding and accessing the data. You can also send data to the external server as part of the process of retrieving data.

Using the URLRequest class

Flash Player 9 and later, Adobe AIR 1.0 and later

Many APIs that load external data use the URLRequest class to define the properties of necessary network request.

URLRequest properties

Flash Player 9 and later, Adobe AIR 1.0 and later

You can set the following properties of a URLRequest object in any security sandbox:

Property	Description
contentType	The MIME content type of any data sent with the URL request. If no contentType is set, values are sent as application/x-www-form-urlencoded.
data	An object containing data to be transmitted with the URL request.
digest	A string that uniquely identifies the signed Adobe platform component to be stored to (or retrieved from) the Adobe® Flash® Player cache.
method	The HTTP request method, such as a GET or POST. (Content running in the AIR application security domain can specify strings other than "GET" or "POST" as the method property. Any HTTP verb is allowed and "GET" is the default method. See "AIR security" on page 69.)
requestHeaders	The array of HTTP request headers to be appended to the HTTP request. Note that permission to set some headers is restricted in Flash Player as well as in AIR content running outside the application security sandbox.
url	Specifies the URL to be requested.

The URLRequest class includes the following properties which are available to content only in the AIR application security sandbox:

Property	Description
followRedirects	Specifies whether redirects are to be followed (<code>true</code> , the default value) or not (<code>false</code>). This is only supported in the AIR application sandbox.
manageCookies	Specifies whether the HTTP protocol stack should manage cookies (<code>true</code> , the default value) or not (<code>false</code>) for this request. Setting this property is only supported in the AIR application sandbox.
authenticate	Specifies whether authentication requests should be handled (<code>true</code>) for this request. Setting this property is only supported in the AIR application sandbox. The default is to authenticate requests—which may cause an authentication dialog box to be displayed if the server requires credentials. You can also set the user name and password using the URLRequestDefaults class—see "Setting URLRequest defaults (AIR only)" on page 317.
cacheResponse	Specifies whether response data should be cached for this request. Setting this property is only supported in the AIR application sandbox. The default is to cache the response (<code>true</code>).
useCache	Specifies whether the local cache should be consulted before this URLRequest fetches data. Setting this property is only supported in the AIR application sandbox. The default (<code>true</code>) is to use the local cached version, if available.
userAgent	Specifies the user-agent string to be used in the HTTP request.

Setting URLRequest defaults (AIR only)

Adobe AIR 1.0 and later

The URLRequestDefaults class lets you define application-specific default settings for URLRequest objects. For example, the following code sets the default values for the `manageCookies` and `useCache` properties. All new URLRequest objects will use the specified values for these properties instead of the normal defaults:

```
air.URLRequestDefaults.manageCookies = false;
air.URLRequestDefaults.useCache = false;
```

Note: The URLRequestDefaults class is defined for content running in Adobe AIR only. It is not supported in Flash Player.

The URLRequestDefaults class includes a `setLoginCredentialsForHost()` method that lets you specify a default user name and password to use for a specific host. The host, which is defined in the `hostname` parameter of the method, can be a domain, such as "www.example.com", or a domain and a port number, such as "www.example.com:80". Note that "example.com", "www.example.com", and "sales.example.com" are each considered unique hosts.

These credentials are only used if the server requires them. If the user has already authenticated (for example, by using the authentication dialog box), then calling the `setLoginCredentialsForHost()` method does not change the authenticated user.

The following code sets the default user name and password to use for requests sent to `www.example.com`:

```
air.URLRequestDefaults.setLoginCredentialsForHost("www.example.com", "Ada", "love1816$X");
```

The `URLRequestDefaults` settings only apply to the current application domain, with one exception. The credentials passed to the `setLoginCredentialsForHost()` method are used for requests made in any application domain within the AIR application.

For more information, see the `URLRequestDefaults` class in the [Adobe AIR API Reference for HTML Developers](#).

URI schemes

Flash Player 9 and later, Adobe AIR 1.0 and later

The standard URI schemes, such as the following, can be used in requests made from any security sandbox:

http: and https:

Use these for standard Internet URLs (in the same way that they are used in a web browser).

file:

Use `file:` to specify the URL of a file located on the local file system. For example:

```
file:///c:/AIR Test/test.txt
```

In AIR, you can also use the following schemes when defining a URL for content running in the application security sandbox:

app:

Use `app:` to specify a path relative to the root directory of the installed application. For example, the following path points to a resources subdirectory of the directory of the installed application:

```
app:/resources
```

When an AIR application is launched using the AIR Debug Launcher (ADL), the application directory is the directory that contains the application descriptor file.

The URL (and `url` property) for a `File` object created with `File.applicationDirectory` uses the `app` URI scheme, as in the following:

```
var dir = air.File.applicationDirectory;  
dir = dir.resolvePath("assets");  
air.trace(dir.url); // app:/assets
```

app-storage:

Use `app-storage:` to specify a path relative to the data storage directory of the application. For each installed application (and user), AIR creates a unique application storage directory, which is a useful place to store data specific to that application. For example, the following path points to a `prefs.xml` file in a settings subdirectory of the application store directory:

```
app-storage:/settings/prefs.xml
```

The URL (and `url` property) for a `File` object created with `File.applicationStorageDirectory` uses the `app-storage` URI scheme, as in the following:

```
var prefsFile = air.File.applicationStorageDirectory;  
prefsFile = prefsFile.resolvePath("prefs.xml");  
air.trace(prefsFile.url); // app-storage:/prefs.xml
```

mailto:

You can use the `mailto` scheme in `URLRequest` objects passed to the `navigateToURL()` function. See “[Opening a URL in another application](#)” on page 328.

You can use a `URLRequest` object that uses any of these URI schemes to define the URL request for a number of different objects, such as a `FileStream` or a `Sound` object. You can also use these schemes in HTML content running in AIR; for example, you can use them in the `src` attribute of an `img` tag.

However, you can only use these AIR-specific URI schemes (`app:` and `app-storage:`) in content in the application security sandbox. For more information, see “[AIR security](#)” on page 69.

Setting URL variables

While you can add variables to the URL string directly, it can be easier to use the `URLVariables` class to define any variables needed for a request.

There are three ways in which you can add parameters to a `URLVariables` object:

- Within the `URLVariables` constructor
- With the `URLVariables.decode()` method
- As dynamic properties of the `URLVariables` object itself

The following example illustrates all three methods and also how to assign the variables to a `URLRequest` object:

```
var urlVar = new air.URLVariables( "one=1&two=2" );  
urlVar.decode("amp=" + air.encodeURIComponent( "&" ) );  
urlVar.three = 3;  
urlVar.amp2 = "&&";  
air.trace(urlVar.toString()); //amp=%26&amp2=%26%26&one=1&two=2&three=3  
  
var urlRequest = new air.URLRequest( "http://www.example.com/test.cfm" );  
urlRequest.data = urlVar;
```

When you define variables within the `URLVariables` constructor or within the `URLVariables.decode()` method, make sure that you URL-encode the characters that have a special meaning in a URI string. For example, when you use an ampersand in a parameter name or value, you must encode the ampersand by changing it from `&` to `%26` because the ampersand acts as a delimiter for parameters. The top-level `encodeURIComponent()` function can be used for this purpose.

Using the URLLoader class

Flash Player 9 and later, Adobe AIR 1.0 and later

The `URLLoader` class let you send a request to a server and access the information returned. You can also use the `URLLoader` class to access files on the local file system in contexts where local file access is permitted (such as the Flash Player local-with-filesystem sandbox and the AIR application sandbox). The `URLLoader` class downloads data from a URL as text, binary data, or URL-encoded variables. The `URLLoader` class dispatches events such as `complete`, `httpStatus`, `ioError`, `open`, `progress`, and `securityError`.

The `URLLoader` class provides an alternative to the `XMLHttpRequest` class. You can use either class to download data via an HTTP request.

Downloaded data is not available until the download has completed. You can monitor the progress of the download (bytes loaded and bytes total) by listening for the `progress` event to be dispatched. However, if a file loads quickly enough a `progress` event might not be dispatched. When a file has successfully downloaded, the `complete` event is dispatched. By setting the `URLLoader dataFormat` property, you can receive the data as text, raw binary data, or as a `URLVariables` object.

The `URLLoader.load()` method (and optionally the `URLLoader` class's constructor) takes a single parameter, `request`, which is a `URLRequest` object. A `URLRequest` object contains all of the information for a single HTTP request, such as the target URL, request method (`GET` or `POST`), additional header information, and the MIME type.

For example, to upload an XML packet to a server-side script, you could use the following code:

```
var secondsUTC = new Date().time;
var dataXML = (new DOMParser()).parseFromString( "<time>" + secondsUTC + "</time>",
"application/xml" );
var request = new air.URLRequest("http://www.example.com/time.cfm");
request.contentType = "text/xml";
request.data = dataXML;
request.method = air.URLRequestMethod.POST;
var loader = new air.URLLoader();
loader.load(request);
```

The previous snippet creates an XML document named `dataXML` that contains the XML packet to be sent to the server. The example sets the `URLRequest contentType` property to `"text/xml"` and assigns the XML document to the `URLRequest data` property. Finally, the example creates a `URLLoader` object and sends the request to the remote script by using the `load()` method.

Using the `URLStream` class

Flash Player 9 and later, Adobe AIR 1.0 and later

The `URLStream` class provides access to the downloading data as the data arrives. The `URLStream` class also lets you close a stream before it finishes downloading. The downloaded data is available as raw binary data.

When reading data from a `URLStream` object, use the `bytesAvailable` property to determine whether sufficient data is available before reading it. An `EOFError` exception is thrown if you attempt to read more data than is available.

The `httpResponseStatus` event (AIR)

The `URLStream` class dispatches an `httpResponseStatus` event before any response data is delivered. The `httpResponseStatus` event (represented by the `HTTPStatusEvent` class) includes a `responseURL` property, which is the URL that the response was returned from, and a `responseHeaders` property, which is an array of `URLRequestHeader` objects representing the response headers that the response returned.

Loading data from external documents

Flash Player 9 and later, Adobe AIR 1.0 and later

When you build dynamic applications, it can be useful to load data from external files or from server-side scripts. This lets you build dynamic applications without having to edit or recompile your application. For example, if you build a "tip of the day" application, you can write a server-side script that retrieves a random tip from a database and saves it to a text file once a day. Then your application can load the contents of a static text file instead of querying the database each time.

The following snippet creates a `URLRequest` and `URLLoader` object, which loads the contents of an external text file, `params.txt`:

```
var request = new air.URLRequest("params.txt");
var loader = new air.URLLoader();
loader.load(request);
```


By default, if you do not define a request method, Flash Player and Adobe AIR load the content using the HTTP `GET` method. To send the request using the `POST` method, set the `request.method` property to `POST` using the static constant `URLRequestMethod.POST`, as the following code shows:

```
var request = new air.URLRequest("http://www.example.com/sendfeedback.cfm");
request.method = air.URLRequestMethod.POST;
```

The external document, `params.txt`, that is loaded at run time contains the following data:

```
monthNames=January, February, March, April, May, June, July, August, September, October, November, December&dayNames=Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
```

The file contains two parameters, `monthNames` and `dayNames`. Each parameter contains a comma-separated list that is parsed as strings. You can split this list into an array using the `String.split()` method.

 *Avoid using reserved words or language constructs as variable names in external data files, because doing so makes reading and debugging your code more difficult.*

Once the data has loaded, the `complete` event is dispatched, and the contents of the external document are available to use in the `URLLoader`'s `data` property, as the following code shows:


```
function completeHandler(event)
{
    var loader2 = URLLoader(event.target);
    air.trace(loader2.data);
}
```

If the remote document contains name-value pairs, you can parse the data using the `URLVariables` class by passing in the contents of the loaded file, as follows:

```
function completeHandler(event)
{
    var loader2 = event.target;
    var variables = new air.URLVariables(loader2.data);
    air.trace(variables.dayNames);
}
```

Each name-value pair from the external file is created as a property in the `URLVariables` object. Each property within the variables object in the previous code sample is treated as a string. If the value of the name-value pair is a list of items, you can convert the string into an array by calling the `String.split()` method, as follows:

```
var dayNameArray = variables.dayNames.split(",");
```

 *If you are loading numeric data from external text files, convert the values into numeric values by using a top-level function, such as `parseInt()`, `parseFloat()`, and `Number()`.*

Instead of loading the contents of the remote file as a string and creating a new `URLVariables` object, you could instead set the `URLLoader.dataFormat` property to one of the static properties found in the `URLLoaderDataFormat` class. The three possible values for the `URLLoader.dataFormat` property are as follows:

- `URLLoaderDataFormat.BINARY`—The `URLLoader.data` property will contain binary data stored in a `ByteArray` object.
- `URLLoaderDataFormat.TEXT`—The `URLLoader.data` property will contain text in a `String` object.

- `URLLoaderDataFormat.VARIABLES`—The `URLLoader.data` property will contain URL-encoded variables stored in a `URLVariables` object.

The following code demonstrates how setting the `URLLoader.dataFormat` property to `URLLoaderDataFormat.VARIABLES` allows you to automatically parse loaded data into a `URLVariables` object:

```
var request = new air.URLRequest("http://www.example.com/params.txt");
var variables = new air.URLLoader();
variables.dataFormat = air.URLLoaderDataFormat.VARIABLES;
variables.addEventListener(air.Event.COMPLETE, completeHandler);
try
{
    variables.load(request);
}
catch (error)
{
    air.trace("Unable to load URL: " + error);
}

function completeHandler(event)
{
    var loader = event.target;
    air.trace(loader.data.dayNames);
}
```

Note: The default value for `URLLoader.dataFormat` is `URLLoaderDataFormat.TEXT`.

As the following example shows, loading XML from an external file is the same as loading `URLVariables`. You can create a `URLRequest` instance and a `URLLoader` instance and use them to download a remote XML document. When the file has completely downloaded, the `complete` event is dispatched and the `trace()` function outputs the contents of the file to the command line.

```
var request = new air.URLRequest("http://www.example.com/data.xml");
var loader = new air.URLLoader();
loader.addEventListener(air.Event.COMPLETE, completeHandler);
loader.load(request);

function completeHandler(event)
{
    var dataXML = event.target.data;
    air.trace(dataXML);
}
```

Communicating with external scripts

Flash Player 9 and later, Adobe AIR 1.0 and later

In addition to loading external data files, you can also use the `URLVariables` class to send variables to a server-side script and process the server's response. This is useful, for example, if you are programming a game and want to send the user's score to a server to calculate whether it should be added to the high scores list, or even send a user's login information to a server for validation. A server-side script can process the user name and password, validate it against a database, and return confirmation of whether the user-supplied credentials are valid.

The following snippet creates a `URLVariables` object named `variables`, which creates a new variable called `name`. Next, a `URLRequest` object is created that specifies the URL of the server-side script to send the variables to. Then you set the `method` property of the `URLRequest` object to send the variables as an `HTTP POST` request. To add the `URLVariables` object to the URL request, you set the `data` property of the `URLRequest` object to the `URLVariables` object created earlier. Finally, the `URLLoader` instance is created and the `URLLoader.load()` method is invoked, which initiates the request.

```
var variables = new air.URLVariables("name=Franklin");
var request = new air.URLRequest();
request.url = "http://www.[yourdomain].com/greeting.cfm";
request.method = air.URLRequestMethod.POST;
request.data = variables;
var loader = new air.URLLoader();
loader.dataFormat = URLLoaderDataFormat.VARIABLES;
loader.addEventListener(Event.COMPLETE, completeHandler);
try
{
    loader.load(request);
}
catch (error)
{
    air.trace("Unable to load URL");
}

function completeHandler(event)
{
    air.trace(event.target.data.welcomeMessage);
}
```

The following code contains the contents of the Adobe ColdFusion® `greeting.cfm` document used in the previous example:

```
<cfif NOT IsDefined("Form.name") OR Len(Trim(Form.Name)) EQ 0>
    <cfset Form.Name = "Stranger" />
</cfif>
<cfoutput>welcomeMessage=#UrlEncodedFormat("Welcome, " & Form.name)#
</cfoutput>
```

Web service requests

Flash Player 9 and later, Adobe AIR 1.0 and later

There are a variety of HTTP-based web services. The main types include:

- REST
- XML-RPC
- SOAP

To use a web service in ActionScript 3, you create a `URLRequest` object, construct the web service call using either URL variables or an XML document, and send the call to the service using a `URLLoader` object. The Flex framework contains several classes that make it easier to use web services—especially useful when accessing complex SOAP services. Starting with Flash Professional CS3, you can use the Flex classes in applications developed with Flash Professional as well as in applications developed in Flash Builder.

In HTML-based AIR applications, you can use either the `URLRequest` and `URLLoader` classes or the JavaScript `XMLHttpRequest` class. If desired, you can also create a SWF library that exposes the web service components of the Flex framework to your JavaScript code.

When your application runs in a browser, you can only use web services in the same Internet domain as the calling SWF unless the server hosting the web service also hosts a cross-domain policy file that permits access from other domains. A technique that is often used when a cross-domain policy file is not available is to proxy the requests through your own server. Adobe Blaze DS and Adobe LiveCycle support web service proxying.

In AIR applications, a cross-domain policy file is not required when the web service call originates from the application security sandbox. AIR application content is never served from a remote domain, so it cannot participate in the types of attacks that cross-domain policies prevent. In HTML-based AIR applications, content in the application security sandbox can make cross-domain `XMLHttpRequests`. You can allow content in other security sandboxes to make cross-domain `XMLHttpRequests` as long as that content is loaded into an `iframe`.

More Help topics

[Adobe BlazeDS](#)

[Adobe LiveCycle ES2](#)

[REST architecture](#)

[XML-RPC](#)

[SOAP protocol](#)

REST-style web service requests

Flash Player 9 and later, Adobe AIR 1.0 and later

REST-style web services use HTTP method verbs to designate the basic action and URL variables to specify the action details. For example, a request to get data for an item could use the `GET` verb and URL variables to specify a method name and item ID. The resulting URL string might look like:

```
http://service.example.com/?method=getItem&id=d3452
```

To access a REST-style web service with `ActionScript`, you can use the `URLRequest`, `URLVariables`, and `URLLoader` classes. In JavaScript code within an AIR application, you can also use an `XMLHttpRequest`.

Programming a REST-style web service call in `ActionScript`, typically involves the following steps:

- 1 Create a `URLRequest` object.
- 2 Set the service URL and HTTP method verb on the request object.
- 3 Create a `URLVariables` object.
- 4 Set the service call parameters as dynamic properties of the variables object.
- 5 Assign the variables object to the data property of the request object.
- 6 Send the call to the service with a `URLLoader` object.
- 7 Handle the `complete` event dispatched by the `URLLoader` that indicates that the service call is complete. It is also wise to listen for the various error events that can be dispatched by a `URLLoader` object.

For example, consider a web service that exposes a test method that echoes the call parameters back to the requestor. The following `ActionScript` code could be used to call the service:

```
function restServiceCall()
{
    //Create the HTTP request object
    var request = new air.URLRequest( "http://service.example.com/" );
    request.method = air.URLRequestMethod.GET;

    //Add the URL variables
    var variables = new air.URLVariables();
    variables.method = "test.echo";
    variables.api_key = "123456ABC";
    variables.message = "Able was I, ere I saw Elba.";
    request.data = variables;

    //Initiate the transaction
    window.requestor = new air.URLLoader();
    requestor.addEventListener( air.Event.COMPLETE, httpRequestComplete );
    requestor.addEventListener( air.IOErrorEvent.IOERROR, httpRequestError );
    requestor.addEventListener( air.SecurityErrorEvent.SECURITY_ERROR, httpRequestError );
    requestor.load( request );
}
function httpRequestComplete( event )
{
    air.trace( event.target.data );
}
function httpRequestError( error ){
    air.trace( "An error occured: " + error.message );
}
```

In JavaScript within an AIR application, you can make the same request using the XMLHttpRequest object:

```
<html>
<head><title>RESTful web service request</title>
<script type="text/javascript">

function makeRequest()
{
    var requestDisplay = document.getElementById( "request" );
    var resultDisplay = document.getElementById( "result" );

    //Create a convenience object to hold the call properties
    var request = {};
    request.URL = "http://service.example.com/";
    request.method = "test.echo";
    request.HTTPmethod = "GET";
    request.parameters = {};
    request.parameters.api_key = "ABCDEF123";
    request.parameters.message = "Able was I ere I saw Elba.";
    var requestURL = makeURL( request );
    xmlhttp = new XMLHttpRequest();
    xmlhttp.open( request.HTTPmethod, requestURL, true);
    xmlhttp.onreadystatechange = function() {
        if (xmlhttp.readyState == 4) {
            resultDisplay.innerHTML = xmlhttp.responseText;
        }
    }
    xmlhttp.send(null);
}
```

```
    requestDisplay.innerHTML = requestURL;
}
//Convert the request object into a properly formatted URL
function makeURL( request )
{
    var url = request.URL + "?method=" + escape( request.method );
    for( var property in request.parameters )
    {
        url += "&" + property + "=" + escape( request.parameters[property] );
    }

    return url;
}
</script>
</head>
<body onload="makeRequest()">
<h1>Request:</h1>
<div id="request"></div>
<h1>Result:</h1>
<div id="result"></div>
</body>
</html>
```

XML-RPC web service requests

Flash Player 9 and later, Adobe AIR 1.0 and later

An XML-RPC web service takes its call parameters as an XML document rather than as a set of URL variables. To conduct a transaction with an XML-RPC web service, create a properly formatted XML message and send it to the web service using the HTTP POST method. In addition, you should set the Content-Type header for the request so that the server treats the request data as XML.

The following example uses DOM methods to create an XML-RPC message and an XMLHttpRequest to conduct the web service transaction:

```
<html>
<head>
<title>XML-RPC web service request</title>
<script type="text/javascript">

function makeRequest()
{
    var requestDisplay = document.getElementById( "request" );
    var resultDisplay = document.getElementById( "result" );

    var request = {};
    request.URL = "http://services.example.com/xmlrpc/";
    request.method = "test.echo";
    request.HTTPmethod = "POST";
    request.parameters = {};
    request.parameters.api_key = "123456ABC";
    request.parameters.message = "Able was I ere I saw Elba.";
    var requestMessage = formatXMLRPC( request );

    xmlhttp = new XMLHttpRequest();
```

```
xmlhttp.open( request.HTTPMethod, request.URL, true);
xmlhttp.onreadystatechange = function() {
    if (xmlhttp.readyState == 4) {
        resultDisplay.innerHTML = xmlhttp.responseText;
    }
}
xmlhttp.send( requestMessage );

requestDisplay.innerHTML = xmlToString( requestMessage.documentElement );
}

//Formats a request as XML-RPC document
function formatXMLRPC( request )
{
    var xmldoc = document.implementation.createDocument( "", "", null );
    var root = xmldoc.createElement( "methodCall" );
    xmldoc.appendChild( root );
    var methodName = xmldoc.createElement( "methodName" );
    var methodString = xmldoc.createTextNode( request.method );
    methodName.appendChild( methodString );

    root.appendChild( methodName );

    var params = xmldoc.createElement( "params" );
    root.appendChild( params );

    var param = xmldoc.createElement( "param" );
    params.appendChild( param );
    var value = xmldoc.createElement( "value" );
    param.appendChild( value );
    var struct = xmldoc.createElement( "struct" );
    value.appendChild( struct );

    for( var property in request.parameters )
    {
        var member = xmldoc.createElement( "member" );
        struct.appendChild( member );

        var name = xmldoc.createElement( "name" );
        var paramName = xmldoc.createTextNode( property );
        name.appendChild( paramName );
        member.appendChild( name );

        var value = xmldoc.createElement( "value" );
        var type = xmldoc.createElement( "string" );
        value.appendChild( type );
        var paramValue = xmldoc.createTextNode( request.parameters[property] );
        type.appendChild( paramValue );
        member.appendChild( value );
    }
    return xmldoc;
}

//Returns a string representation of an XML node
function xmlToString( rootNode, indent )
{
    if( indent == null ) indent = "";
```

```
var result = indent + "<" + rootNode.tagName + ">\n";
for( var i = 0; i < rootNode.childNodes.length; i++)
{
    if(rootNode.childNodes.item( i ).nodeType == Node.TEXT_NODE )
    {
        result += indent + "    " + rootNode.childNodes.item( i ).textContent + "\n";
    }
}
if( rootNode.childElementCount > 0 )
{
    result += xmlToString( rootNode.firstChild, indent + "    " );
}
if( rootNode.nextElementSibling )
{
    result += indent + "</" + rootNode.tagName + ">\n";
    result += xmlToString( rootNode.nextElementSibling, indent );
}
else
{
    result += indent + "</" + rootNode.tagName + ">\n";
}
return result;
}

</script>
</head>
<body onload="makeRequest()" >
<h1>Request:</h1>
<pre id="request"></pre>
<h1>Result:</h1>
<pre id="result"></pre>
</body>
</html>
```

Opening a URL in another application

Flash Player 9 and later, Adobe AIR 1.0 and later

You can use the `navigateToURL()` function to open a URL in the default system web browser.

For the `URLRequest` object you pass as the `request` parameter of this function, only the `url` property is used.

The first parameter of the `navigateToURL()` function, the `navigate` parameter, is a `URLRequest` object (see [“Using the URLRequest class”](#) on page 316). The second is an optional `window` parameter, in which you can specify the window name. For example, the following code opens the `www.adobe.com` web page:

```
var url = "http://www.adobe.com";
var urlReq = new air.URLRequest(url);
air.navigateToURL(urlReq);
```

Note: When using the `navigateToURL()` function, the runtime treats a `URLRequest` object that uses the `POST` method (one that has its `method` property set to `URLRequestMethod.POST`) as using the `GET` method.

When using the `navigateToURL()` function, URI schemes are permitted based on the security sandbox of the code calling the `navigateToURL()` function.

Some APIs allow you to launch content in a web browser. For security reasons, some URI schemes are prohibited when using these APIs in AIR. The list of prohibited schemes depends on the security sandbox of the code using the API. (For details on security sandboxes, see “[AIR security](#)” on page 69.)

Application sandbox (AIR only)

Any URI scheme can be used in URL launched by content running in the AIR application sandbox. An application must be registered to handle the URI scheme or the request does nothing. The following schemes are supported on many computers and devices:

- `http:`
- `https:`
- `file:`
- `mailto:` — AIR directs these requests to the registered system mail application
- `sms:` — AIR directs `sms:` requests to the default text message app. The URL format must conform to the system conventions under which the app is running. For example, on Android, the URI scheme must be lowercase.

```
navigateToURL( new URLRequest( "sms:+15555550101" ) );
```
- `tel:` — AIR directs `tel:` requests to the default telephone dialing app. The URL format must conform to the system conventions under which the app is running. For example, on Android, the URI scheme must be lowercase.

```
navigateToURL( new URLRequest( "tel:5555555555" ) );
```
- `market:` — AIR directs `market:` requests to the Market app typically supported on Android devices.

```
navigateToURL( new URLRequest( "market://search?q=Adobe Flash" ) );  
navigateToURL( new URLRequest( "market://search?q=pname:com.adobe.flashplayer" ) );
```

Where allowed by the operating system, applications can define and register custom URI schemes. You can create a URL using the scheme to launch the application from AIR.

Remote sandboxes

The following schemes are allowed. Use these schemes as you would use them in a web browser.

- `http:`
- `https:`
- `mailto:` — AIR directs these requests to the registered system mail application

All other URI schemes are prohibited.

Local-with-file sandbox

The following schemes are allowed. Use these schemes as you would use them in a web browser.

- `file:`
- `mailto:` — AIR directs these requests to the registered system mail application

All other URI schemes are prohibited.

Local-with-networking sandbox

The following schemes are allowed. Use these schemes as you would use them in a web browser.

- `http:`
- `https:`

- `mailto:` — AIR directs these requests to the registered system mail application

All other URI schemes are prohibited.

Local-trusted sandbox

The following schemes are allowed. Use these schemes as you would use them in a web browser.

- `file:`
- `http:`
- `https:`
- `mailto:` — AIR directs these requests to the registered system mail application

All other URI schemes are prohibited.

Sending a URL to a server

Flash Player 9 and later, Adobe AIR 1.0 and later

You can use the `sendToURL()` function to send a URL request to a server. This function ignores any server response. The `sendToURL()` function takes one argument, `request`, which is a `URLRequest` object (see “[Using the URLRequest class](#)” on page 316). Here is an example:

```
var url = "http://www.example.com/application.jsp";
var variables = new air.URLVariables();
variables.sessionId = new Date().getTime();
variables.userLabel = "Your Name";
var request = new air.URLRequest(url);
request.data = variables;
air.sendToURL(request);
```

This example uses the `URLVariables` class to include variable data in the `URLRequest` object. For more information, see “[Using the URLLoader class](#)” on page 319.

Chapter 24: Communicating with other Flash Player and AIR instances

Flash Player 9 and later, Adobe AIR 1.0 and later

The `LocalConnection` class enables communications between Adobe® AIR® applications, as well as between SWF content running in the browser. You can also use the `LocalConnection` class to communicate between an AIR application and SWF content running in the browser. The `LocalConnection` class allows you to build versatile applications that can share data between Flash Player and AIR instances.

About the `LocalConnection` class

Flash Player 9 and later, Adobe AIR 1.0 and later

`LocalConnection` objects can communicate only among AIR applications and SWF files that are running on the same client computer. However, the applications can run in different applications. For example, two AIR applications can communicate using the `LocalConnection` class, as can an AIR application and a SWF file running in a browser.

The simplest way to use a `LocalConnection` object is to allow communication only between `LocalConnection` objects located in the same domain or the same AIR application. That way, you do not have to worry about security issues. However, if you need to allow communication between domains, you have several ways to implement security measures. For more information, see the discussion of the `connectionName` parameter of the `send()` method and the `allowDomain()` and `domain` entries in the `LocalConnection` class listing in the [ActionScript 3.0 Reference for the Adobe Flash Platform](#).

To add callback methods to your `LocalConnection` objects, set the `LocalConnection.client` property to an object that has member methods, as the following code shows:

```
var lc = new air.LocalConnection();
var clientObject = new Object();
clientObject.doMethod1 = function() {
    air.trace("doMethod1 called.");
}
clientObject.doMethod2 = function(param1) {
    air.trace("doMethod2 called with one parameter: " + param1);
    air.trace("The square of the parameter is: " + param1 * param1);
}
lc.client = clientObject;
```

The `LocalConnection.client` property includes all callback methods that can be invoked.

isPerUser property

The `isPerUser` property was added to Flash Player (10.0.32) and AIR (1.5.2) to resolve a conflict that occurs when more than one user is logged into a Mac computer. On other operating systems, the property is ignored since the local connection has always been scoped to individual users. The `isPerUser` property should be set to `true` in new code. However, the default value is currently `false` for backward compatibility. The default may be changed in future versions of the runtimes.

Sending messages between two applications

Flash Player 9 and later, Adobe AIR 1.0 and later

You use the `LocalConnection` class to communicate between different AIR applications and between different Adobe® Flash® Player (SWF) applications running in a browser. You can also use the `LocalConnection` class to communicate between an AIR application and a SWF application running in a browser.

The following code defines a `LocalConnection` object that acts as a server and accepts incoming `LocalConnection` calls from other applications:

```
var lc = new air.LocalConnection();
lc.connect("connectionName");
var clientObject = new Object();
clientObject.echoMsg = function(msg) {
    air.trace("This message was received: " + msg);
}
lc.client = clientObject;
```

This code first creates a `LocalConnection` object named `lc` and sets the `client` property to an object, `clientObject`. When another application calls a method in this `LocalConnection` instance, the runtime looks for that method in the `clientObject` object.

If you already have a connection with the specified name, an `ArgumentError` exception is thrown, indicating that the connection attempt failed because the object is already connected.

The following snippet demonstrates how to create a `LocalConnection` with the name `conn1`:

```
connection.connect("conn1");
```

Connecting to the primary application from a secondary application requires that you first create a `LocalConnection` object in the sending `LocalConnection` object; then call the `LocalConnection.send()` method with the name of the connection and the name of the method to execute. For example, to send the `doQuit` method to the `LocalConnection` object that you created earlier, use the following code:

```
sendingConnection.send("conn1", "doQuit");
```

This code connects to an existing `LocalConnection` object with the connection name `conn1` and invokes the `doMessage()` method in the remote application. If you want to send parameters to the remote application, you specify additional arguments after the method name in the `send()` method, as the following snippet shows:

```
sendingConnection.send("conn1", "doMessage", "Hello world");
```

Connecting to content in different domains and to AIR applications

Flash Player 9 and later, Adobe AIR 1.0 and later

To allow communications only from specific domains, you call the `allowDomain()` or `allowInsecureDomain()` method of the `LocalConnection` class and pass a list of one or more domains that are allowed to access this `LocalConnection` object, passing one or more names of domains to be allowed.

There are two special values that you can pass to the `LocalConnection.allowDomain()` and `LocalConnection.allowInsecureDomain()` methods: `*` and `localhost`. The asterisk value (`*`) allows access from all domains. The string `localhost` allows calls to the application from content locally installed, but outside of the application resource directory.

If the `LocalConnection.send()` method attempts to communicate with an application from a security sandbox to which the calling code does not have access, a `securityError` event (`SecurityErrorEvent.SECURITY_ERROR`) is dispatched. To work around this error, you can specify the caller's domain in the receiver's `LocalConnection.allowDomain()` method.

If you implement communication only between content in the same domain, you can specify a `connectionName` parameter that does not begin with an underscore (`_`) and does not specify a domain name (for example, `myDomain:connectionName`). Use the same string in the `LocalConnection.connect(connectionName)` command.

If you implement communication between content in different domains, you specify a `connectionName` parameter that begins with an underscore. Specifying the underscore makes the content with the receiving `LocalConnection` object more portable between domains. Here are the two possible cases:

- If the string for `connectionName` does not begin with an underscore, the runtime adds a prefix with the superdomain name and a colon (for example, `myDomain:connectionName`). Although this ensures that your connection does not conflict with connections of the same name from other domains, any sending `LocalConnection` objects must specify this superdomain (for example, `myDomain:connectionName`). If you move the HTML or SWF file with the receiving `LocalConnection` object to another domain, the runtime changes the prefix to reflect the new superdomain (for example, `anotherDomain:connectionName`). All sending `LocalConnection` objects have to be manually edited to point to the new superdomain.
- If the string for `connectionName` begins with an underscore (for example, `_connectionName`), the runtime does not add a prefix to the string. This means the receiving and sending `LocalConnection` objects use identical strings for `connectionName`. If the receiving object uses `LocalConnection.allowDomain()` to specify that connections from any domain will be accepted, you can move the HTML or SWF file with the receiving `LocalConnection` object to another domain without altering any sending `LocalConnection` objects.

A downside to using underscore names in `connectionName` is the potential for collisions, such as when two applications both try to connect using the same `connectionName`. A second, related downside is security-related. Connection names that use underscore syntax do not identify the domain of the listening application. For these reasons, domain-qualified names are preferred.

Adobe AIR

To communicate with content running in the AIR application security sandbox (content installed with the AIR application), you must prefix the connection name with a superdomain identifying the AIR application. The superdomain string starts with `app#` followed by the application ID followed by a dot (`.`) character, followed by the publisher ID (if defined). For example, the proper superdomain to use in the `connectionName` parameter for an application with the application ID, `com.example.air.MyApp`, and no publisher ID is:
`"app#com.example.air.MyApp"`. Thus, if the base connection name is "appConnection," then the entire string to use in the `connectionName` parameter is: `"app#com.example.air.MyApp:appConnection"`. If the application has the publisher ID, then the that ID must also be included in the superdomain string:
`"app#com.example.air.MyApp.B146A943FBD637B68C334022D304CEA226D129B4.1"`.

When you allow another AIR application to communicate with your application through the local connection, you must call the `allowDomain()` of the `LocalConnection` object, passing in the local connection domain name. For an AIR application, this domain name is formed from the application and publisher IDs in the same fashion as the connection string. For example, if the sending AIR application has an application ID of `com.example.air.FriendlyApp` and a publisher ID of `214649436BD677B62C33D02233043EA236D13934.1`, then the domain string that you would use to allow this application to connect is:
`app#com.example.air.FriendlyApp.214649436BD677B62C33D02233043EA236D13934.1`. (As of AIR 1.5.3, not all AIR applications have publisher IDs.)

Chapter 25: ActionScript basics for JavaScript developers

Adobe® ActionScript® 3.0 is a programming language like JavaScript—both are based on ECMAScript. ActionScript 3.0 was released with Adobe® Flash® Player 9 and you can therefore develop rich Internet applications with it in Adobe® Flash® CS3 Professional, Adobe® Flash® CS4 Professional, and Adobe® Flex™ 3.

The current version of ActionScript 3.0 was available only when developing SWF content for Flash Player 9 in the browser. It is now also available for developing SWF content running in Adobe® AIR®.

The [Adobe AIR API Reference for HTML Developers](#) includes documentation for those classes that are useful in JavaScript code in an HTML-based application. It's a subset of the entire set of classes in the runtime. Other classes in the runtime are useful in developing SWF-based applications (the `DisplayObject` class for example, which defines the structure of visual content). If you need to use these classes in JavaScript, refer to the following ActionScript documentation:

- The [Adobe ActionScript 3.0 Developer's Guide](#)
- The [Adobe ActionScript 3.0 Reference for the Adobe Flash Platform](#). (Only the top-level classes and functions in the flash package are available to HTML content running in AIR. The classes in the mx package are available only in Flex-based SWF applications.)

Differences between ActionScript and JavaScript: an overview

ActionScript, like JavaScript, is based on the ECMAScript language specification; therefore, the two languages have a common core syntax. For example, the following code works the same in JavaScript and in ActionScript:

```
var str1 = "hello";
var str2 = " world.";
var str = reverseString(str1 + str2);

function reverseString(s) {
    var newString = "";
    var i;
    for (i = s.length - 1; i >= 0; i--) {
        newString += s.charAt(i);
    }
    return newString;
}
```

However, there are differences in the syntax and workings of the two languages. For example, the preceding code example can be written as the following in ActionScript 3.0 (in a SWF file):

```
function reverseString(s:String):String {
    var newString:String = "";
    for (var i:int = s.length - 1; i >= 0; i--) {
        newString += s.charAt(i);
    }
    return newString;
}
```

The version of JavaScript supported in HTML content in Adobe AIR is JavaScript 1.7. The differences between JavaScript 1.7 and ActionScript 3.0 are described throughout this topic.

The runtime includes some built-in classes that provide advanced capabilities. At runtime, JavaScript in an HTML page can access those classes. The same runtime classes are available both to ActionScript (in a SWF file) and JavaScript (in an HTML file running in a browser). However, the current API documentation for these classes (which are not included in the [Adobe AIR API Reference for HTML Developers](#)) describes them using ActionScript syntax. In other words, for some of the advanced capabilities of the runtime, refer to The [Adobe ActionScript 3.0 Reference for the Adobe Flash Platform](#). Understanding the basics of ActionScript helps you understand how to use these runtime classes in JavaScript.

For example, the following JavaScript code plays sound from an MP3 file:

```
var file = air.File.userDirectory.resolve("My Music/test.mp3");
var sound = air.Sound(file);
sound.play();
```

Each of these lines of code calls runtime functionality from JavaScript.

In a SWF file, ActionScript code can access these runtime capabilities as in the following code:

```
var file:File = File.userDirectory.resolve("My Music/test.mp3");
var sound = new Sound(file);
sound.play();
```

ActionScript 3.0 data types

ActionScript 3.0 is a *strongly typed* language. That means that you can assign a data type to a variable. For example, the first line of the previous example could be written as the following:

```
var str1:String = "hello";
```

Here, the `str1` variable is declared to be of type `String`. All subsequent assignments to the `str1` variable assign `String` values to the variable.

You can assign types to variables, parameters of functions, and return types of functions. Therefore, the function declaration in the previous example looks like the following in ActionScript:

```
function reverseString(s:String):String {
    var newString:String = "";
    for (var i:int = s.length - 1; i >= 0; i--) {
        newString += s.charAt(i);
    }
    return newString;
}
```

Note: The `s` parameter and the return value of the function are both assigned the type `String`.

Although assigning types is optional in ActionScript, there are advantages to declaring types for objects:

- Typed objects allow for type checking of data not only at run-time, but also at compile time if you use strict mode. Type checking at compile time helps identify errors. (Strict mode is a compiler option.)
- Using typed objects creates applications that are more efficient.

For this reason, the examples in the ActionScript documentation use data types. Often, you can convert sample ActionScript code to JavaScript by simply removing the type declarations (such as `:String`).

Data types corresponding to custom classes

An ActionScript 3.0 object can have a data type that corresponds to the top-level classes, such as `String`, `Number`, or `Date`.

In ActionScript 3.0, you can define custom classes. Each custom class also defines a data type. This means that an ActionScript variable, function parameter, or function return can have a type annotation defined by that class. For more information, see [“Custom ActionScript 3.0 classes”](#) on page 338.

The void data type

The `void` data type is used as the return value for a function that, in fact, returns no value. (A function that does not include a `return` statement returns no value.)

The * data type

Use of the asterisk character (`*`) as a data type is the same as not assigning a data type. For example, the following function includes a parameter, `n`, and a return value that are both not given a data type:

```
function exampleFunction(n:*):* {  
    trace("hi, " + n);  
}
```

Use of the `*` as a data type is not defining a data type at all. You use the asterisk in ActionScript 3.0 code to be explicit that no data type is defined.

ActionScript 3.0 classes, packages, and namespaces

ActionScript 3.0 includes capabilities related to classes that are not found in JavaScript 1.7.

Runtime classes

The runtime includes built-in classes, many of which are also included in standard JavaScript, such as the `Array`, `Date`, `Math`, and `String` classes (and others). However, the runtime also includes classes that are not found in standard JavaScript. These additional classes have various uses, from playing rich media (such as sounds) to interacting with sockets.

Most runtime classes are in the `flash` package, or one of the packages contained by the `flash` package. Packages are a means to organize ActionScript 3.0 classes (see [“ActionScript 3.0 packages”](#) on page 338).

Custom ActionScript 3.0 classes

ActionScript 3.0 allows developers to create their own custom classes. For example, the following code defines a custom class named `ExampleClass`:

```
public class ExampleClass {
    public var x:Number;
    public function ExampleClass(input:Number):void {
        x = input;
    }
    public function greet():void {
        trace("The value of x is: ", x);
    }
}
```

This class has the following members:

- A constructor method, `ExampleClass()`, which lets you instantiate new objects of the `ExampleClass` type.
- A public property, `x` (of type `Number`), which you can get and set for objects of type `ExampleClass`.
- A public method, `greet()`, which you can call on objects of type `ExampleClass`.

In this example, the `x` property and the `greet()` method are in the `public` namespace. The `public` namespace makes methods and properties accessible from objects and classes outside the class.

ActionScript 3.0 packages

Packages provide the means to arrange ActionScript 3.0 classes. For example, many classes related to working with files and directories on the computer are included in the `flash.filesystem` package. In this case, `flash` is one package that contains another package, `filesystem`. And that package may contain other classes or packages. In fact, the `flash.filesystem` package contains the following classes: `File`, `FileMode`, and `FileStream`. To reference the `File` class in ActionScript, you can write the following:

```
flash.filesystem.File
```

Both built-in and custom classes can be arranged in packages.

When referencing an ActionScript package from JavaScript, use the special `runtime` object. For example, the following code instantiates a new ActionScript `File` object in JavaScript:

```
var myFile = new air.flash.filesystem.File();
```

Here, the `File()` method is the constructor function corresponding to the class of the same name (`File`).

ActionScript 3.0 namespaces

In ActionScript 3.0, namespaces define the scope for which properties and functions in classes can be accessed.

Only those properties and methods in the `public` namespace are available in JavaScript.

For example, the `File` class (in the `flash.filesystem` package) includes `public` properties and methods, such as `userDirectory` and `resolve()`. Both are available as properties of a JavaScript variable that instantiates a `File` object (via the `runtime.flash.filesystem.File()` constructor method).

There are four predefined namespaces:

Namespace	Description
public	Any code that instantiates an object of a certain type can access the public properties and methods in the class that defines that type. Also, any code can access the public static properties and methods of a public class.
private	Properties and methods designated as private are only available to code within the class. They cannot be accessed as properties or methods of an object defined by that class. Properties and methods in the private namespace are not available in JavaScript.
protected	Properties and methods designated as protected are only available to code in the class definition and to classes that inherit that class. Properties and methods in the protected namespace are not available in JavaScript.
internal	Properties and methods designated as internal are available to any caller within the same package. Classes, properties, and methods belong to the internal namespace by default.

Additionally, custom classes can use other namespaces that are not available to JavaScript code.

Required parameters and default values in ActionScript 3.0 functions

In both ActionScript 3.0 and JavaScript, functions can include parameters. In ActionScript 3.0, parameters can be required or optional; whereas in JavaScript, parameters are always optional.

The following ActionScript 3.0 code defines a function for which the one parameter, *n*, is required:

```
function cube(n:Number):Number {  
    return n*n*n;  
}
```

The following ActionScript 3.0 code defines a function for which the *n* parameter is required. It also includes the *p* parameter, which is optional, with a default value of 1:

```
function root(n:Number, p:Number = 1):Number {  
    return Math.pow(n, 1/p);  
}
```

An ActionScript 3.0 function can also receive any number of arguments, represented by `...rest` syntax at the end of a list of parameters, as in the following:

```
function average(... args) : Number{  
    var sum:Number = 0;  
    for (var i:int = 0; i < args.length; i++) {  
        sum += args[i];  
    }  
    return (sum / args.length);  
}
```

ActionScript 3.0 event listeners

In ActionScript 3.0 programming, all events are handled using *event listeners*. An event listener is a function. When an object dispatches an event, the event listener responds to the event. The event, which is an ActionScript object, is passed to the event listener as a parameter of the function. This use of event object differs from the DOM event model used in JavaScript.

For example, when you call the `load()` method of a `Sound` object (to load an mp3 file), the `Sound` object attempts to load the sound. Then the `Sound` object dispatches any of the following events:

Event	Description
<code>complete</code>	When the data has loaded successfully.
<code>id3</code>	When mp3 ID3 data is available.
<code>ioError</code>	When an input/output error occurs that causes a load operation to fail.
<code>open</code>	When the load operation starts.
<code>progress</code>	When data is received as a load operation progresses.

Any class that can dispatch events either extends the `EventDispatcher` class or implements the `IEventDispatcher` interface. (An ActionScript 3.0 interface is a data type used to define a set of methods that a class can implement.) In each class listing for these classes in the ActionScript Language Reference, there is a list of events that the class can dispatch.

You can register an event listener function to handle any of these events, using the `addEventListener()` method of the object that dispatches the event. For example, in the case of a `Sound` object, you can register for the `progress` and `complete` events, as shown in the following ActionScript code:

```
var sound:Sound = new Sound();
var urlReq:URLRequest = new URLRequest("test.mp3");
sound.load(urlReq);
sound.addEventListener(ProgressEvent.PROGRESS, progressHandler);
sound.addEventListener(Event.COMPLETE, completeHandler);

function progressHandler(progressEvent):void {
    trace("Progress " + progressEvent.bytesTotal + " bytes out of " + progressEvent.bytesTotal);
}

function completeHandler(completeEvent):void {
    trace("Sound loaded.");
}
```

In HTML content running in AIR, you can register a JavaScript function as the event listener. The following code illustrates this. (This code assumes that the HTML document includes a `TextArea` object named `progressTextArea`.)

```
var sound = new runtime.flash.media.Sound();
var urlReq = new runtime.flash.net.URLRequest("test.mp3");
sound.load(urlReq);
sound.addEventListener(runtime.flash.events.ProgressEvent.PROGRESS, progressHandler);
sound.addEventListener(runtime.flash.events.Event.COMPLETE, completeHandler);

function progressHandler(progressEvent) {
    document.progressTextArea.value += "Progress " + progressEvent.bytesTotal + " bytes out
of " + progressEvent.bytesTotal;
}

function completeHandler(completeEvent) {
    document.progressTextArea.value += "Sound loaded.";
}
```

Chapter 26: SQL support in local databases

Adobe AIR includes a SQL database engine with support for local SQL databases with many standard SQL features, using the open source [SQLite](#) database system. The runtime does not specify how or where database data is stored on the file system. Each database is stored completely within a single file. A developer can specify the location in the file system where the database file is stored, and a single AIR application can access one or many separate databases (i.e. separate database files). This document outlines the SQL syntax and data type support for Adobe AIR local SQL databases. This document is not intended to serve as a comprehensive SQL reference. Rather, it describes specific details of the SQL dialect that Adobe AIR supports. The runtime supports most of the SQL-92 standard SQL dialect. Because there are numerous references, web sites, books, and training materials for learning SQL, this document is not intended to be a comprehensive SQL reference or tutorial. Instead, this document particularly focuses on the AIR-supported SQL syntax, and the differences between SQL-92 and the supported SQL dialect.

SQL statement definition conventions

Within statement definitions in this document, the following conventions are used:

- Text case
 - UPPER CASE - literal SQL keywords are written in all upper case.
 - lower case - placeholder terms or clause names are written in all lower case.
- Definition characters
 - ::= Indicates a clause or statement definition.
- Grouping and alternating characters
 - | The pipe character is used between alternative options, and can be read as "or".
 - [] Items in square brackets are optional items; the brackets can contain a single item or a set of alternative items.
 - () Parentheses surrounding a set of alternatives (a set of items separated by pipe characters), designates a required group of items, that is, a set of items that are the possible values for a single required item.
- Quantifiers
 - + A plus character following an item in parentheses indicates that the preceding item can occur 1 or more times.
 - * An asterisk character following an item in square brackets indicates that the preceding (bracketed) item can occur 0 or more times
- Literal characters
 - * An asterisk character used in a column name or between the parentheses following a function name signifies a literal asterisk character rather than the "0 or more" quantifier.
 - . A period character represents a literal period.
 - , A comma character represents a literal comma.
 - () A pair of parentheses surrounding a single clause or item indicates that the parentheses are required, literal parentheses characters.
 - Other characters, unless otherwise indicated, represent those literal characters.

Supported SQL syntax

The following SQL syntax listings are supported by the Adobe AIR SQL database engine. The listings are divided into explanations of different statement and clause types, expressions, built-in functions, and operators. The following topics are covered:

- General SQL syntax
- Data manipulation statements (SELECT, INSERT, UPDATE, and DELETE)
- Data definition statements (CREATE, ALTER, and DROP statements for tables, indices, views, and triggers)
- Special statements and clauses
- Built-in functions (Aggregate, scalar, and date/time formatting functions)
- Operators
- Parameters
- Unsupported SQL features
- Additional SQL features

General SQL syntax

In addition to the specific syntax for various statements and expressions, the following are general rules of SQL syntax:

Case sensitivity SQL statements, including object names, are not case sensitive. Nevertheless, SQL statements are frequently written with SQL keywords written in uppercase, and this document uses that convention. While SQL syntax is not case sensitive, literal text values in SQL are case sensitive, and comparison and sorting operations can be case sensitive, as specified by the collation sequence defined for a column or operation. For more information see COLLATE.

White space A white-space character (such as space, tab, new line, and so forth) must be used to separate individual words in an SQL statement. However, white space is optional between words and symbols. The type and quantity of white-space characters in a SQL statement is not significant. You can use white space, such as indenting and line breaks, to format your SQL statements for easy readability, without affecting the meaning of the statement.

Data manipulation statements

Data manipulation statements are the most commonly used SQL statements. These statements are used to retrieve, add, modify, and remove data from database tables. The following data manipulation statements are supported: SELECT, INSERT, UPDATE, and DELETE.

SELECT

The SELECT statement is used to query the database. The result of a SELECT is zero or more rows of data where each row has a fixed number of columns. The number of columns in the result is specified by the result column name or expression list between the SELECT and optional FROM keywords.

```

sql-statement ::= SELECT [ALL | DISTINCT] result
               [FROM table-list]
               [WHERE expr]
               [GROUP BY expr-list]
               [HAVING expr]
               [compound-op select-statement]*
               [ORDER BY sort-expr-list]
               [LIMIT integer [( OFFSET | , ) integer]]

result        ::= result-column [, result-column]*
result-column ::= * | table-name . * | expr [[AS] string]
table-list    ::= table [ join-op table join-args ]*
table        ::= table-name [AS alias] |
               ( select ) [AS alias]

join-op       ::= , | [NATURAL] [LEFT | RIGHT | FULL] [OUTER | INNER | CROSS] JOIN
join-args     ::= [ON expr] [USING ( id-list )]
compound-op   ::= UNION | UNION ALL | INTERSECT | EXCEPT
sort-expr-list ::= expr [sort-order] [, expr [sort-order]]*
sort-order    ::= [COLLATE collation-name] [ASC | DESC]
collation-name ::= BINARY | NOCASE

```

Any arbitrary expression can be used as a result. If a result expression is `*` then all columns of all tables are substituted for that one expression. If the expression is the name of a table followed by `.*` then the result is all columns in that one table.

The `DISTINCT` keyword causes a subset of result rows to be returned, in which each result row is different. `NULL` values are not treated as distinct from each other. The default behavior is that all result rows are returned, which can be made explicit with the keyword `ALL`.

The query is executed against one or more tables specified after the `FROM` keyword. If multiple table names are separated by commas, then the query uses the cross join of the various tables. The `JOIN` syntax can also be used to specify how tables are joined. The only type of outer join that is supported is `LEFT OUTER JOIN`. The `ON` clause expression in `join-args` must resolve to a boolean value. A subquery in parentheses may be used as a table in the `FROM` clause. The entire `FROM` clause may be omitted, in which case the result is a single row consisting of the values of the result expression list.

The `WHERE` clause is used to limit the number of rows the query retrieves. `WHERE` clause expressions must resolve to a boolean value. `WHERE` clause filtering is performed before any grouping, so `WHERE` clause expressions may not include aggregate functions.

The `GROUP BY` clause causes one or more rows of the result to be combined into a single row of output. A `GROUP BY` clause is especially useful when the result contains aggregate functions. The expressions in the `GROUP BY` clause do not have to be expressions that appear in the `SELECT` expression list.

The `HAVING` clause is like `WHERE` in that it limits the rows returned by the statement. However, the `HAVING` clause applies after any grouping specified by a `GROUP BY` clause has occurred. Consequently, the `HAVING` expression may refer to values that include aggregate functions. A `HAVING` clause expression is not required to appear in the `SELECT` list. Like a `WHERE` expression, a `HAVING` expression must resolve to a boolean value.

The `ORDER BY` clause causes the output rows to be sorted. The `sort-expr-list` argument to the `ORDER BY` clause is a list of expressions that are used as the key for the sort. The expressions do not have to be part of the result for a simple `SELECT`, but in a compound `SELECT` (a `SELECT` using one of the `compound-op` operators) each sort expression must exactly match one of the result columns. Each sort expression may be optionally followed by a `sort-order` clause consisting of the `COLLATE` keyword and the name of a collation function used for ordering text and/or the keyword `ASC` or `DESC` to specify the sort order (ascending or descending). The `sort-order` can be omitted and the default (ascending order) is used. For a definition of the `COLLATE` clause and collation functions, see `COLLATE`.

The LIMIT clause places an upper bound on the number of rows returned in the result. A negative LIMIT indicates no upper bound. The optional OFFSET following LIMIT specifies how many rows to skip at the beginning of the result set. In a compound SELECT query, the LIMIT clause may only appear after the final SELECT statement, and the limit is applied to the entire query. Note that if the OFFSET keyword is used in the LIMIT clause, then the limit is the first integer and the offset is the second integer. If a comma is used instead of the OFFSET keyword, then the offset is the first number and the limit is the second number. This seeming contradiction is intentional — it maximizes compatibility with legacy SQL database systems.

A compound SELECT is formed from two or more simple SELECT statements connected by one of the operators UNION, UNION ALL, INTERSECT, or EXCEPT. In a compound SELECT, all the constituent SELECT statements must specify the same number of result columns. There can only be a single ORDER BY clause after the final SELECT statement (and before the single LIMIT clause, if one is specified). The UNION and UNION ALL operators combine the results of the preceding and following SELECT statements into a single table. The difference is that in UNION, all result rows are distinct, but in UNION ALL, there may be duplicates. The INTERSECT operator takes the intersection of the results of the preceding and following SELECT statements. EXCEPT takes the result of preceding SELECT after removing the results of the following SELECT. When three or more SELECT statements are connected into a compound, they group from first to last.

For a definition of permitted expressions, see Expressions.

Starting with AIR 2.5, the SQL CAST operator is supported when reading to convert BLOB data to ActionScript ByteArray objects. For example, the following code reads raw data that is not stored in the AMF format and stores it in a ByteArray object:

```
stmt.text = "SELECT CAST(data AS ByteArray) AS data FROM pictures;";
stmt.execute();
var result:SQLResult = stmt.getResult();
var bytes:ByteArray = result.data[0].data;
```

INSERT

The INSERT statement comes in two basic forms and is used to populate tables with data.

```
sql-statement ::= INSERT [OR conflict-algorithm] INTO [database-name.] table-name [(column-
list)] VALUES (value-list) |
                INSERT [OR conflict-algorithm] INTO [database-name.] table-name [(column-
list)] select-statement
                REPLACE INTO [database-name.] table-name [(column-list)] VALUES (value-list) |
                REPLACE INTO [database-name.] table-name [(column-list)] select-statement
```

The first form (with the VALUES keyword) creates a single new row in an existing table. If no column-list is specified then the number of values must be the same as the number of columns in the table. If a column-list is specified, then the number of values must match the number of specified columns. Columns of the table that do not appear in the column list are filled with the default value defined when the table is created, or with NULL if no default value is defined.

The second form of the INSERT statement takes its data from a SELECT statement. The number of columns in the result of the SELECT must exactly match the number of columns in the table if column-list is not specified, or it must match the number of columns named in the column-list. A new entry is made in the table for every row of the SELECT result. The SELECT may be simple or compound. For a definition of allowable SELECT statements, see SELECT.

The optional conflict-algorithm allows the specification of an alternative constraint conflict resolution algorithm to use during this one command. For an explanation and definition of conflict algorithms, see [“Special statements and clauses”](#) on page 351.

The two REPLACE INTO forms of the statement are equivalent to using the standard INSERT [OR conflict-algorithm] form with the REPLACE conflict algorithm (i.e. the INSERT OR REPLACE... form).

The two REPLACE INTO forms of the statement are equivalent to using the standard INSERT [OR conflict-algorithm] form with the REPLACE conflict algorithm (i.e. the INSERT OR REPLACE... form).

UPDATE

The update command changes the existing records in a table.

```
sql-statement ::= UPDATE [database-name.] table-name SET column1=value1, column2=value2,...  
[WHERE expr]
```

The command consists of the UPDATE keyword followed by the name of the table in which you want to update the records. After the SET keyword, provide the name of the column and the value to which the column to be changed as a comma-separated list. The WHERE clause expression provides the row or rows in which the records are updated.

DELETE

The delete command is used to remove records from a table.

```
sql-statement ::= DELETE FROM [database-name.] table-name [WHERE expr]
```

The command consists of the DELETE FROM keywords followed by the name of the table from which records are to be removed.

Without a WHERE clause, all rows of the table are removed. If a WHERE clause is supplied, then only those rows that match the expression are removed. The WHERE clause expression must resolve to a boolean value. For a definition of permitted expressions, see Expressions.

Data definition statements

Data definition statements are used to create, modify, and remove database objects such as tables, views, indices, and triggers. The following data definition statements are supported:

- Tables:
 - CREATE TABLE
 - ALTER TABLE
 - DROP TABLE
- Indices:
 - CREATE INDEX
 - DROP INDEX
- Views:
 - CREATE VIEWS
 - DROP VIEWS
- Triggers:
 - CREATE TRIGGERS
 - DROP TRIGGERS

CREATE TABLE

A CREATE TABLE statement consists of the keywords CREATE TABLE followed by the name of the new table, then (in parentheses) a list of column definitions and constraints. The table name can be either an identifier or a string.

```

sql-statement      ::= CREATE [TEMP | TEMPORARY] TABLE [IF NOT EXISTS] [database-name.] table-
name
                    ( column-def [, column-def]* [, constraint]* )
sql-statement      ::= CREATE [TEMP | TEMPORARY] TABLE [database-name.] table-name AS select-
statement
column-def         ::= name [type] [[CONSTRAINT name] column-constraint]*
type               ::= typename | typename ( number ) | typename ( number , number )
column-constraint ::= NOT NULL [ conflict-clause ] |
                    PRIMARY KEY [sort-order] [ conflict-clause ] [AUTOINCREMENT] |
                    UNIQUE [conflict-clause] |
                    CHECK ( expr ) |
                    DEFAULT default-value |
                    COLLATE collation-name
constraint         ::= PRIMARY KEY ( column-list ) [conflict-clause] |
                    UNIQUE ( column-list ) [conflict-clause] |
                    CHECK ( expr )
conflict-clause   ::= ON CONFLICT conflict-algorithm
conflict-algorithm ::= ROLLBACK | ABORT | FAIL | IGNORE | REPLACE
default-value     ::= NULL | string | number | CURRENT_TIME | CURRENT_DATE | CURRENT_TIMESTAMP
sort-order        ::= ASC | DESC
collation-name    ::= BINARY | NOCASE
column-list       ::= column-name [, column-name]*

```

Each column definition is the name of the column followed by the data type for that column, then one or more optional column constraints. The data type for the column restricts what data may be stored in that column. If an attempt is made to store a value in a column with a different data type, the runtime converts the value to the appropriate type if possible, or raises an error. See the Data type support section for additional information.

The NOT NULL column constraint indicates that the column cannot contain NULL values.

A UNIQUE constraint causes an index to be created on the specified column or columns. This index must contain unique keys—no two rows may contain duplicate values or combinations of values for the specified column or columns. A CREATE TABLE statement can have multiple UNIQUE constraints, including multiple columns with a UNIQUE constraint in the column's definition and/or multiple table-level UNIQUE constraints.

A CHECK constraint defines an expression that is evaluated and must be true in order for a row's data to be inserted or updated. The CHECK expression must resolve to a boolean value.

A COLLATE clause in a column definition specifies what text collation function to use when comparing text entries for the column. The BINARY collating function is used by default. For details on the COLLATE clause and collation functions, see COLLATE.

The DEFAULT constraint specifies a default value to use when doing an INSERT. The value may be NULL, a string constant, or a number. The default value may also be one of the special case-independent keywords CURRENT_TIME, CURRENT_DATE or CURRENT_TIMESTAMP. If the value is NULL, a string constant, or a number, it is literally inserted into the column whenever an INSERT statement does not specify a value for the column. If the value is CURRENT_TIME, CURRENT_DATE or CURRENT_TIMESTAMP, then the current UTC date and/or time is inserted into the column. For CURRENT_TIME, the format is HH:MM:SS. For CURRENT_DATE, the format is YYYY-MM-DD. The format for CURRENT_TIMESTAMP is YYYY-MM-DD HH:MM:SS.

Specifying a PRIMARY KEY normally just creates a UNIQUE index on the corresponding column or columns. However, if the PRIMARY KEY constraint is on a single column that has the data type INTEGER (or one of its synonyms such as int) then that column is used by the database as the actual primary key for the table. This means that the column may only hold unique integer values. (Note that in many SQLite implementations, only the column type INTEGER causes the column to serve as the internal primary key, but in Adobe AIR synonyms for INTEGER such as int also specify that behavior.)

If a table does not have an INTEGER PRIMARY KEY column, an integer key is automatically generated when a row is inserted. The primary key for a row can always be accessed using one of the special names ROWID, OID, or _ROWID_. These names can be used regardless of whether it is an explicitly declared INTEGER PRIMARY KEY or an internal generated value. However, if the table has an explicit INTEGER PRIMARY KEY, the name of the column in the result data is the actual column name rather than the special name.

An INTEGER PRIMARY KEY column can also include the keyword AUTOINCREMENT. When the AUTOINCREMENT keyword is used, the database automatically generates and inserts a sequentially incremented integer key in the INTEGER PRIMARY KEY column when it executes an INSERT statement that doesn't specify an explicit value for the column.

There can only be one PRIMARY KEY constraint in a CREATE TABLE statement. It can either be part of one column's definition or one single table-level PRIMARY KEY constraint. A primary key column is implicitly NOT NULL.

The optional conflict-clause following many constraints allows the specification of an alternative default constraint conflict resolution algorithm for that constraint. The default is ABORT. Different constraints within the same table may have different default conflict resolution algorithms. If an INSERT or UPDATE statement specifies a different conflict resolution algorithm, that algorithm is used in place of the algorithm specified in the CREATE TABLE statement. See the ON CONFLICT section of [“Special statements and clauses”](#) on page 351 for additional information.

Additional constraints, such as FOREIGN KEY constraints, do not result in an error but the runtime ignores them.

If the TEMP or TEMPORARY keyword occurs between CREATE and TABLE then the table that is created is only visible within the same database connection (SQLConnection instance). It is automatically deleted when the database connection is closed. Any indices created on a temporary table are also temporary. Temporary tables and indices are stored in a separate file distinct from the main database file.

If the optional database-name prefix is specified, then the table is created in a named database (a database that was connected to the SQLConnection instance by calling the attach() method with the specified database name). It is an error to specify both a database-name prefix and the TEMP keyword, unless the database-name prefix is temp. If no database name is specified, and the TEMP keyword is not present, the table is created in the main database (the database that was connected to the SQLConnection instance using the open() or openAsync() method).

There are no arbitrary limits on the number of columns or on the number of constraints in a table. There is also no arbitrary limit on the amount of data in a row.

The CREATE TABLE AS form defines the table as the result set of a query. The names of the table columns are the names of the columns in the result.

If the optional IF NOT EXISTS clause is present and another table with the same name already exists, then the database ignores the CREATE TABLE command.

A table can be removed using the DROP TABLE statement, and limited changes can be made using the ALTER TABLE statement.

ALTER TABLE

The ALTER TABLE command allows the user to rename or add a new column to an existing table. It is not possible to remove a column from a table.

```
sql-statement ::= ALTER TABLE [database-name.] table-name alteration
alteration    ::= RENAME TO new-table-name
alteration    ::= ADD [COLUMN] column-def
```

The RENAME TO syntax is used to rename the table identified by [database-name.] table-name to new-table-name. This command cannot be used to move a table between attached databases, only to rename a table within the same database.

If the table being renamed has triggers or indices, then they remain attached to the table after it has been renamed. However, if there are any view definitions or statements executed by triggers that refer to the table being renamed, they are not automatically modified to use the new table name. If a renamed table has associated views or triggers, you must manually drop and recreate the triggers or view definitions using the new table name.

The ADD [COLUMN] syntax is used to add a new column to an existing table. The new column is always appended to the end of the list of existing columns. The column-def clause may take any of the forms permissible in a CREATE TABLE statement, with the following restrictions:

- The column may not have a PRIMARY KEY or UNIQUE constraint.
- The column may not have a default value of CURRENT_TIME, CURRENT_DATE or CURRENT_TIMESTAMP.
- If a NOT NULL constraint is specified, the column must have a default value other than NULL.

The execution time of the ALTER TABLE statement is not affected by the amount of data in the table.

DROP TABLE

The DROP TABLE statement removes a table added with a CREATE TABLE statement. The table with the specified table-name is the table that's dropped. It is completely removed from the database and the disk file. The table cannot be recovered. All indices associated with the table are also deleted.

```
sql-statement ::= DROP TABLE [IF EXISTS] [database-name.] table-name
```

By default the DROP TABLE statement does not reduce the size of the database file. Empty space in the database is retained and used in subsequent INSERT operations. To remove free space in the database use the SQLConnection.clean() method. If the autoClean parameter is set to true when the database is initially created, the space is freed automatically.

The optional IF EXISTS clause suppresses the error that would normally result if the table does not exist.

CREATE INDEX

The CREATE INDEX command consists of the keywords CREATE INDEX followed by the name of the new index, the keyword ON, the name of a previously created table that is to be indexed, and a parenthesized list of names of columns in the table whose values are used for the index key.

```
sql-statement ::= CREATE [UNIQUE] INDEX [IF NOT EXISTS] [database-name.] index-name
                ON table-name ( column-name [, column-name]* )
column-name   ::= name [COLLATE collation-name] [ASC | DESC]
```

Each column name can be followed by ASC or DESC keywords to indicate sort order, but the sort order designation is ignored by the runtime. Sorting is always done in ascending order.

The COLLATE clause following each column name defines a collating sequence used for text values in that column. The default collation sequence is the collation sequence defined for that column in the CREATE TABLE statement. If no collation sequence is specified, the BINARY collation sequence is used. For a definition of the COLLATE clause and collation functions see COLLATE.

There are no arbitrary limits on the number of indices that can be attached to a single table. There are also no limits on the number of columns in an index.

DROP INDEX

The drop index statement removes an index added with the CREATE INDEX statement. The specified index is completely removed from the database file. The only way to recover the index is to reenter the appropriate CREATE INDEX command.

```
sql-statement ::= DROP INDEX [IF EXISTS] [database-name.] index-name
```

By default the DROP INDEX statement does not reduce the size of the database file. Empty space in the database is retained and used in subsequent INSERT operations. To remove free space in the database use the `SQLConnection.clean()` method. If the `autoClean` parameter is set to true when the database is initially created, the space is freed automatically.

CREATE VIEW

The CREATE VIEW command assigns a name to a pre-defined SELECT statement. This new name can then be used in a FROM clause of another SELECT statement in place of a table name. Views are commonly used to simplify queries by combining a complex (and frequently used) set of data into a structure that can be used in other operations.

```
sql-statement ::= CREATE [TEMP | TEMPORARY] VIEW [IF NOT EXISTS] [database-name.] view-name AS  
select-statement
```

If the TEMP or TEMPORARY keyword occurs in between CREATE and VIEW then the view that is created is only visible to the SQLConnection instance that opened the database and is automatically deleted when the database is closed.

If a [database-name] is specified the view is created in the named database (a database that was connected to the SQLConnection instance using the `attach()` method, with the specified name argument. It is an error to specify both a [database-name] and the TEMP keyword unless the [database-name] is temp. If no database name is specified, and the TEMP keyword is not present, the view is created in the main database (the database that was connected to the SQLConnection instance using the `open()` or `openAsync()` method).

Views are read only. A DELETE, INSERT, or UPDATE statement cannot be used on a view, unless at least one trigger of the associated type (INSTEAD OF DELETE, INSTEAD OF INSERT, INSTEAD OF UPDATE) is defined. For information on creating a trigger for a view, see CREATE TRIGGER.

A view is removed from a database using the DROP VIEW statement.

DROP VIEW

The DROP VIEW statement removes a view created by a CREATE VIEW statement.

```
sql-statement ::= DROP VIEW [IF EXISTS] view-name
```

The specified view-name is the name of the view to drop. It is removed from the database, but no data in the underlying tables is modified.

CREATE TRIGGER

The create trigger statement is used to add triggers to the database schema. A trigger is a database operation (the trigger-action) that is automatically performed when a specified database event (the database-event) occurs.

```
sql-statement ::= CREATE [TEMP | TEMPORARY] TRIGGER [IF NOT EXISTS] [database-name.] trigger-
name
                [BEFORE | AFTER] database-event
                ON table-name
                trigger-action
sql-statement ::= CREATE [TEMP | TEMPORARY] TRIGGER [IF NOT EXISTS] [database-name.] trigger-
name
                INSTEAD OF database-event
                ON view-name
                trigger-action
database-event ::= DELETE |
                INSERT |
                UPDATE |
                UPDATE OF column-list
trigger-action ::= [FOR EACH ROW] [WHEN expr]
                BEGIN
                    trigger-step ;
                    [ trigger-step ; ]*
                END
trigger-step  ::= update-statement |
                insert-statement |
                delete-statement |
                select-statement
column-list   ::= column-name [, column-name]*
```

A trigger is specified to fire whenever a DELETE, INSERT, or UPDATE of a particular database table occurs, or whenever an UPDATE of one or more specified columns of a table are updated. Triggers are permanent unless the TEMP or TEMPORARY keyword is used. In that case the trigger is removed when the SQLConnection instance's main database connection is closed. If no timing is specified (BEFORE or AFTER) the trigger defaults to BEFORE.

Only FOR EACH ROW triggers are supported, so the FOR EACH ROW text is optional. With a FOR EACH ROW trigger, the trigger-step statements are executed for each database row being inserted, updated or deleted by the statement causing the trigger to fire, if the WHEN clause expression evaluates to true.

If a WHEN clause is supplied, the SQL statements specified as trigger-steps are only executed for rows for which the WHEN clause is true. If no WHEN clause is supplied, the SQL statements are executed for all rows.

Within the body of a trigger, (the trigger-action clause) the pre-change and post-change values of the affected table are available using the special table names OLD and NEW. The structure of the OLD and NEW tables matches the structure of the table on which the trigger is created. The OLD table contains any rows that are modified or deleted by the triggering statement, in their state before the triggering statement's operations. The NEW table contains any rows that are modified or created by the triggering statement, in their state after the triggering statement's operations. Both the WHEN clause and the trigger-step statements can access values from the row being inserted, deleted or updated using references of the form NEW.column-name and OLD.column-name, where column-name is the name of a column from the table with which the trigger is associated. The availability of the OLD and NEW table references depends on the type of database-event the trigger handles:

- INSERT – NEW references are valid
- UPDATE – NEW and OLD references are valid
- DELETE – OLD references are valid

The specified timing (BEFORE, AFTER, or INSTEAD OF) determines when the trigger-step statements are executed relative to the insertion, modification or removal of the associated row. An ON CONFLICT clause may be specified as part of an UPDATE or INSERT statement in a trigger-step. However, if an ON CONFLICT clause is specified as part of the statement causing the trigger to fire, then that conflict handling policy is used instead.

In addition to table triggers, an INSTEAD OF trigger can be created on a view. If one or more INSTEAD OF INSERT, INSTEAD OF DELETE, or INSTEAD OF UPDATE triggers are defined on a view, it is not considered an error to execute the associated type of statement (INSERT, DELETE, or UPDATE) on the view. In that case, executing an INSERT, DELETE or UPDATE on the view causes the associated triggers to fire. Because the trigger is an INSTEAD OF trigger, the tables underlying the view are not modified by the statement that causes the trigger to fire. However, the triggers can be used to perform modifying operations on the underlying tables.

There is an important issue to keep in mind when creating a trigger on a table with an INTEGER PRIMARY KEY column. If a BEFORE trigger modifies the INTEGER PRIMARY KEY column of a row that is to be updated by the statement that causes the trigger to fire, the update doesn't occur. A workaround is to create the table with a PRIMARY KEY column instead of an INTEGER PRIMARY KEY column.

A trigger can be removed using the DROP TRIGGER statement. When a table or view is dropped, all triggers associated with that table or view are automatically dropped as well.

RAISE () function

A special SQL function RAISE() can be used in a trigger-step statement of a trigger. This function has the following syntax:

```
raise-function ::= RAISE ( ABORT, error-message ) |  
                  RAISE ( FAIL, error-message ) |  
                  RAISE ( ROLLBACK, error-message ) |  
                  RAISE ( IGNORE )
```

When one of the first three forms is called during trigger execution, the specified ON CONFLICT processing action (ABORT, FAIL, or ROLLBACK) is performed and the current statement's execution ends. The ROLLBACK is considered a statement execution failure, so the SQLStatement instance whose execute() method was being carried out dispatches an error (SQLErrorEvent.ERROR) event. The SQLError object in the dispatched event object's error property has its details property set to the error-message specified in the RAISE() function.

When RAISE(IGNORE) is called, the remainder of the current trigger, the statement that caused the trigger to execute, and any subsequent triggers that would have been executed are abandoned. No database changes are rolled back. If the statement that caused the trigger to execute is itself part of a trigger, that trigger program resumes execution at the beginning of the next step. For more information about the conflict resolution algorithms, see the section ON CONFLICT (conflict algorithms).

DROP TRIGGER

The DROP TRIGGER statement removes a trigger created by the CREATE TRIGGER statement.

```
sql-statement ::= DROP TRIGGER [IF EXISTS] [database-name.] trigger-name
```

The trigger is deleted from the database. Note that triggers are automatically dropped when their associated table is dropped.

Special statements and clauses

This section describes several clauses that are extensions to SQL provided by the runtime, as well as two language elements that can be used in many statements, comments and expressions.

COLLATE

The COLLATE clause is used in SELECT, CREATE TABLE, and CREATE INDEX statements to specify the comparison algorithm that is used when comparing or sorting values.

```
sql-statement ::= COLLATE collation-name  
collation-name ::= BINARY | NOCASE
```

The default collation type for columns is BINARY. When BINARY collation is used with values of the TEXT storage class, binary collation is performed by comparing the bytes in memory that represent the value regardless of the text encoding.

The NOCASE collation sequence is only applied for values of the TEXT storage class. When used, the NOCASE collation performs a case-insensitive comparison.

No collation sequence is used for storage classes of type NULL, BLOB, INTEGER, or REAL.

To use a collation type other than BINARY with a column, a COLLATE clause must be specified as part of the column definition in the CREATE TABLE statement. Whenever two TEXT values are compared, a collation sequence is used to determine the results of the comparison according to the following rules:

- For binary comparison operators, if either operand is a column, then the default collation type of the column determines the collation sequence that is used for the comparison. If both operands are columns, then the collation type for the left operand determines the collation sequence used. If neither operand is a column, then the BINARY collation sequence is used.
- The BETWEEN...AND operator is equivalent to using two expressions with the >= and <= operators. For example, the expression x BETWEEN y AND z is equivalent to x >= y AND x <= z. Consequently, the BETWEEN...AND operator follows the preceding rule to determine the collation sequence.
- The IN operator behaves like the =operator for the purposes of determining the collation sequence to use. For example, the collation sequence used for the expression x IN (y, z) is the default collation type of x if x is a column. Otherwise, BINARY collation is used.
- An ORDER BY clause that is part of a SELECT statement may be explicitly assigned a collation sequence to be used for the sort operation. In that case the explicit collation sequence is always used. Otherwise, if the expression sorted by an ORDER BY clause is a column, the default collation type of the column is used to determine sort order. If the expression is not a column, the BINARY collation sequence is used.

EXPLAIN

The EXPLAIN command modifier is a non-standard extension to SQL.

```
sql-statement ::= EXPLAIN sql-statement
```

If the EXPLAIN keyword appears before any other SQL statement, then instead of actually executing the command, the result reports the sequence of virtual machine instructions it would have used to execute the command, had the EXPLAIN keyword not been present. The EXPLAIN feature is an advanced feature and allows developers to change SQL statement text in an attempt to optimize performance or debug a statement that doesn't appear to be working properly.

ON CONFLICT (conflict algorithms)

The ON CONFLICT clause is not a separate SQL command. It is a non-standard clause that can appear in many other SQL commands.

```
conflict-clause ::= ON CONFLICT conflict-algorithm  
conflict-clause ::= OR conflict-algorithm  
conflict-algorithm ::= ROLLBACK |  
ABORT |  
FAIL |  
IGNORE |  
REPLACE
```

The first form of the ON CONFLICT clause, using the keywords ON CONFLICT, is used in a CREATE TABLE statement. For an INSERT or UPDATE statement, the second form is used, with ON CONFLICT replaced by OR to make the syntax seem more natural. For example, instead of INSERT ON CONFLICT IGNORE, the statement becomes INSERT OR IGNORE. Although the keywords are different, the meaning of the clause is the same in either form.

The ON CONFLICT clause specifies the algorithm that is used to resolve constraint conflicts. The five algorithms are ROLLBACK, ABORT, FAIL, IGNORE, and REPLACE. The default algorithm is ABORT. The following is an explanation of the five conflict algorithms:

ROLLBACK When a constraint violation occurs, an immediate ROLLBACK occurs, ending the current transaction. The command aborts and the SQLStatement instance dispatches an error event. If no transaction is active (other than the implied transaction that is created on every command) then this algorithm works the same as ABORT.

ABORT When a constraint violation occurs, the command backs out any prior changes it might have made and the SQLStatement instance dispatches an error event. No ROLLBACK is executed, so changes from prior commands within a transaction are preserved. ABORT is the default behavior.

FAIL When a constraint violation occurs, the command aborts and the SQLStatement dispatches an error event. However, any changes to the database that the statement made before encountering the constraint violation are preserved and are not backed out. For example, if an UPDATE statement encounters a constraint violation on the 100th row that it attempts to update, then the first 99 row changes are preserved but changes to rows 100 and beyond don't occur.

IGNORE When a constraint violation occurs, the one row that contains the constraint violation is not inserted or changed. Aside from this row being ignored, the command continues executing normally. Other rows before and after the row that contained the constraint violation continue to be inserted or updated normally. No error is returned.

REPLACE When a UNIQUE constraint violation occurs, the pre-existing rows that are causing the constraint violation are removed before inserting or updating the current row. Consequently, the insert or update always occurs, and the command continues executing normally. No error is returned. If a NOT NULL constraint violation occurs, the NULL value is replaced by the default value for that column. If the column has no default value, then the ABORT algorithm is used. If a CHECK constraint violation occurs then the IGNORE algorithm is used. When this conflict resolution strategy deletes rows in order to satisfy a constraint, it does not invoke delete triggers on those rows.

The algorithm specified in the OR clause of an INSERT or UPDATE statement overrides any algorithm specified in a CREATE TABLE statement. If no algorithm is specified in the CREATE TABLE statement or the executing INSERT or UPDATE statement, the ABORT algorithm is used.

REINDEX

The REINDEX command is used to delete and re-create one or more indices. This command is useful when the definition of a collation sequence has changed.

```
sql-statement ::= REINDEX collation-name  
sql-statement ::= REINDEX [database-name .] ( table-name | index-name )
```

In the first form, all indices in all attached databases that use the named collation sequence are recreated. In the second form, when a table-name is specified, all indices associated with the table are rebuilt. If an index-name is given, only the specified index is deleted and recreated.

COMMENTS

Comments aren't SQL commands, but they can occur in SQL queries. They are treated as white space by the runtime. They can begin anywhere white space can be found, including inside expressions that span multiple lines.

```
comment          ::= single-line-comment |
                  block-comment
single-line-comment ::= -- single-line
block-comment    ::= /* multiple-lines or block [*/]
```

A single-line comment is indicated by two dashes. A single line comment only extends to the end of the current line.

Block comments can span any number of lines, or be embedded within a single line. If there is no terminating delimiter, a block comment extends to the end of the input. This situation is not treated as an error. A new SQL statement can begin on a line after a block comment ends. Block comments can be embedded anywhere white space can occur, including inside expressions, and in the middle of other SQL statements. Block comments do not nest. Single-line comments inside a block comment are ignored.

EXPRESSIONS

Expressions are subcommands within other SQL blocks. The following describes the valid syntax for an expression within a SQL statement:

```
expr             ::= expr binary-op expr |
                  expr [NOT] like-op expr [ESCAPE expr] |
                  unary-op expr |
                  ( expr ) |
                  column-name |
                  table-name.column-name |
                  database-name.table-name.column-name |
                  literal-value |
                  parameter |
                  function-name( expr-list | * ) |
                  expr ISNULL |
                  expr NOTNULL |
                  expr [NOT] BETWEEN expr AND expr |
                  expr [NOT] IN ( value-list ) |
                  expr [NOT] IN ( select-statement ) |
                  expr [NOT] IN [database-name.] table-name |
                  [EXISTS] ( select-statement ) |
                  CASE [expr] ( WHEN expr THEN expr )+ [ELSE expr] END |
                  CAST ( expr AS type ) |
                  expr COLLATE collation-name

like-op          ::= LIKE | GLOB
binary-op        ::= see Operators
unary-op         ::= see Operators
parameter        ::= :param-name | @param-name | ?
value-list       ::= literal-value [, literal-value]*
literal-value    ::= literal-string | literal-number | literal-boolean | literal-blob |
literal-null
literal-string   ::= 'string value'
literal-number   ::= integer | number
literal-boolean  ::= true | false
literal-blob     ::= X'string of hexadecimal data'
literal-null     ::= NULL
```

An expression is any combination of values and operators that can be resolved to a single value. Expressions can be divided into two general types, according to whether they resolve to a boolean (true or false) value or whether they resolve to a non-boolean value.

In several common situations, including in a WHERE clause, a HAVING clause, the ON expression in a JOIN clause, and a CHECK expression, the expression must resolve to a boolean value. The following types of expressions meet this condition:

- ISNULL
- NOTNULL
- IN ()
- EXISTS ()
- LIKE
- GLOB
- Certain functions
- Certain operators (specifically comparison operators)

Literal values

A literal numeric value is written as an integer number or a floating point number. Scientific notation is supported. The . (period) character is always used as the decimal point.

A string literal is indicated by enclosing the string in single quotes '. To include a single quote within a string, put two single quotes in a row like this example: ''.

A boolean literal is indicated by the value true or false. Literal boolean values are used with the Boolean column data type.

A BLOB literal is a string literal containing hexadecimal data and preceded by a single x or X character, such as X'53514697465'.

A literal value can also be the token NULL.

Column name

A column name can be any of the names defined in the CREATE TABLE statement or one of the following special identifiers: ROWID, OID, or _ROWID_. These special identifiers all describe the unique random integer key (the "row key") associated with every row of every table. The special identifiers only refer to the row key if the CREATE TABLE statement does not define a real column with the same name. Row keys behave as read-only columns. A row key can be used anywhere a regular column can be used, except that you cannot change the value of a row key in an UPDATE or INSERT statement. The SELECT * FROM table statement does not include the row key in its result set.

SELECT statement

A SELECT statement can appear in an expression as either the right-hand operand of the IN operator, as a scalar quantity (a single result value), or as the operand of an EXISTS operator. When used as a scalar quantity or the operand of an IN operator, the SELECT can only have a single column in its result. A compound SELECT statement (connected with keywords like UNION or EXCEPT) is allowed. With the EXISTS operator, the columns in the result set of the SELECT are ignored and the expression returns TRUE if one or more rows exist and FALSE if the result set is empty. If no terms in the SELECT expression refer to the value in the containing query, then the expression is evaluated once before any other processing and the result is reused as necessary. If the SELECT expression does contain variables from the outer query, known as a correlated subquery, then the SELECT is re-evaluated every time it is needed.

When a SELECT is the right operand of the IN operator, the IN operator returns TRUE if the result of the left operand is equal to any of the values in the SELECT statement's result set. The IN operator may be preceded by the NOT keyword to invert the sense of the test.

When a SELECT appears within an expression but is not the right operand of an IN operator, then the first row of the result of the SELECT becomes the value used in the expression. If the SELECT yields more than one result row, all rows after the first are ignored. If the SELECT yields no rows, then the value of the SELECT is NULL.

CAST expression

A CAST expression changes the data type of the value specified to the one given. The type specified can be any non-empty type name that is valid for the type in a column definition of a CREATE TABLE statement. See Data type support for details.

Additional expression elements

The following SQL elements can also be used in expressions:

- Built-in functions: Aggregate functions, Scalar functions, and Date and time formatting functions
- Operators
- Parameters

Built-in functions

The built-in functions fall into three main categories:

- Aggregate functions
- Scalar functions
- Date and time functions

In addition to these functions, there is a special function RAISE() that is used to provide notification of an error in the execution of a trigger. This function can only be used within the body of a CREATE TRIGGER statement. For information on the RAISE() function, see CREATE TRIGGER > RAISE().

Like all keywords in SQL, function names are not case sensitive.

Aggregate functions

Aggregate functions perform operations on values from multiple rows. These functions are primarily used in SELECT statements in conjunction with the GROUP BY clause.

AVG(X)	Returns the average value of all non-NULL X within a group. String and BLOB values that do not look like numbers are interpreted as 0. The result of AVG() is always a floating point value even if all inputs are integers.
COUNT(X) COUNT(*)	The first form returns a count of the number of times that X is not NULL in a group. The second form (with the * argument) returns the total number of rows in the group.
MAX(X)	Returns the maximum value of all values in the group. The usual sort order is used to determine the maximum.
MIN(X)	Returns the minimum non-NULL value of all values in the group. The usual sort order is used to determine the minimum. If all values in the group are NULL, NULL is returned.
SUM(X)	Returns the numeric sum of all non-NULL values in the group. If all of the values are NULL then SUM() returns NULL, and TOTAL() returns 0.0. The result of TOTAL() is always a floating point value. The result of SUM() is an integer value if all non-NULL inputs are integers. If any input to SUM() is not an integer and not NULL then SUM() returns a floating point value. This value might be an approximation to the true sum.
TOTAL(X)	

In any of the preceding aggregate functions that take a single argument, that argument can be preceded by the keyword DISTINCT. In that case, duplicate elements are filtered before being passed into the aggregate function. For example, the function call COUNT(DISTINCT x) returns the number of distinct values of column X instead of the total number of non-NULL values in column x.

Scalar functions

Scalar functions operate on values one row at a time.

ABS(X)	Returns the absolute value of argument X.
COALESCE(X, Y, ...)	Returns a copy of the first non-NULL argument. If all arguments are NULL then NULL is returned. There must be at least two arguments.
GLOB(X, Y)	This function is used to implement the X GLOB Y syntax.
IFNULL(X, Y)	Returns a copy of the first non-NULL argument. If both arguments are NULL then NULL is returned. This function behaves the same as COALESCE().
HEX(X)	The argument is interpreted as a value of the BLOB storage type. The result is a hexadecimal rendering of the content of that value.
LAST_INSERT_ROWID()	Returns the row identifier (generated primary key) of the last row inserted to the database through the current SQLConnection. This value is the same as the value returned by the SQLConnection.lastInsertRowID property.
LENGTH(X)	Returns the string length of X in characters.
LIKE(X, Y [, Z])	This function is used to implement the X LIKE Y [ESCAPE Z] syntax of SQL. If the optional ESCAPE clause is present, then the function is invoked with three arguments. Otherwise, it is invoked with two arguments only.
LOWER(X)	Returns a copy of string X with all characters converted to lower case.
LTRIM(X) LTRIM(X, Y)	Returns a string formed by removing spaces from the left side of X. If a Y argument is specified, the function removes any of the characters in Y from the left side of X.
MAX(X, Y, ...)	Returns the argument with the maximum value. Arguments may be strings in addition to numbers. The maximum value is determined by the defined sort order. Note that MAX() is a simple function when it has 2 or more arguments but is an aggregate function when it has a single argument.
MIN(X, Y, ...)	Returns the argument with the minimum value. Arguments may be strings in addition to numbers. The minimum value is determined by the defined sort order. Note that MIN() is a simple function when it has 2 or more arguments but is an aggregate function when it has a single argument.
NULLIF(X, Y)	Returns the first argument if the arguments are different, otherwise returns NULL.
QUOTE(X)	This routine returns a string which is the value of its argument suitable for inclusion into another SQL statement. Strings are surrounded by single-quotes with escapes on interior quotes as needed. BLOB storage classes are encoded as hexadecimal literals. The function is useful when writing triggers to implement undo/redo functionality.
RANDOM(*)	Returns a pseudo-random integer between -9223372036854775808 and 9223372036854775807. This random value is not crypto-strong.
RANDBLOB(N)	Returns an N-byte BLOB containing pseudo-random bytes. N should be a positive integer. This random value is not crypto-strong. If the value of N is negative a single byte is returned.
ROUND(X) ROUND(X, Y)	Rounds off the number X to Y digits to the right of the decimal point. If the Y argument is omitted, 0 is used.
RTRIM(X) RTRIM(X, Y)	Returns a string formed by removing spaces from the right side of X. If a Y argument is specified, the function removes any of the characters in Y from the right side of X.
SUBSTR(X, Y, Z)	Returns a substring of input string X that begins with the Y-th character and which is Z characters long. The left-most character of X is index position 1. If Y is negative the first character of the substring is found by counting from the right rather than the left.
TRIM(X) TRIM(X, Y)	Returns a string formed by removing spaces from the right side of X. If a Y argument is specified, the function removes any of the characters in Y from the right side of X.
typeof(X)	Returns the type of the expression X. The possible return values are 'null', 'integer', 'real', 'text', and 'blob'. For more information on data types see Data type support .
UPPER(X)	Returns a copy of input string X converted to all upper-case letters.
ZEROBLOB(N)	Returns a BLOB containing N bytes of 0x00.

Date and time formatting functions

The date and time formatting functions are a group of scalar functions that are used to create formatted date and time data. Note that these functions operate on and return string and number values. These functions are not intended to be used with the DATE data type. If you use these functions on data in a column whose declared data type is DATE, they do not behave as expected.

DATE(T, ...)	The DATE() function returns a string containing the date in this format: YYYY-MM-DD. The first parameter (T) specifies a time string of the format found under Time formats. Any number of modifiers can be specified after the time string. The modifiers can be found under Modifiers.
TIME(T, ...)	The TIME() function returns a string containing the time as HH:MM:SS. The first parameter (T) specifies a time string of the format found under Time formats. Any number of modifiers can be specified after the time string. The modifiers can be found under Modifiers.
DATETIME(T, ...)	The DATETIME() function returns a string containing the date and time in YYYY-MM-DD HH:MM:SS format. The first parameter (T) specifies a time string of the format found under Time formats. Any number of modifiers can be specified after the time string. The modifiers can be found under Modifiers.
JULIANDAY(T, ...)	The JULIANDAY() function returns a number indicating the number of days since noon in Greenwich on November 24, 4714 B.C. and the provided date. The first parameter (T) specifies a time string of the format found under Time formats. Any number of modifiers can be specified after the time string. The modifiers can be found under Modifiers.
STRFTIME(F, T, ...)	<p>The STRFTIME() routine returns the date formatted according to the format string specified as the first argument F. The format string supports the following substitutions:</p> <p>%d - day of month</p> <p>%f - fractional seconds SS.SSS</p> <p>%H - hour 00-24</p> <p>%j - day of year 001-366</p> <p>%J - Julian day number</p> <p>%m -month 01-12</p> <p>%M - minute 00-59</p> <p>%s - seconds since 1970-01-01</p> <p>%S - seconds 00-59</p> <p>%w - day of week 0-6 (sunday = 0)</p> <p>%W - week of year 00-53</p> <p>%Y - year 0000-9999</p> <p>%% - %</p> <p>The second parameter (T) specifies a time string of the format found under Time formats. Any number of modifiers can be specified after the time string. The modifiers can be found under Modifiers.</p>

Time formats

A time string can be in any of the following formats:

YYYY-MM-DD	2007-06-15
YYYY-MM-DD HH:MM	2007-06-15 07:30
YYYY-MM-DD HH:MM:SS	2007-06-15 07:30:59
YYYY-MM-DD HH:MM:SS.SSS	2007-06-15 07:30:59.152
YYYY-MM-DDTHH:MM	2007-06-15T07:30
YYYY-MM-DDTHH:MM:SS	2007-06-15T07:30:59
YYYY-MM-DDTHH:MM:SS.SSS	2007-06-15T07:30:59.152
HH:MM	07:30 (date is 2000-01-01)
HH:MM:SS	07:30:59 (date is 2000-01-01)
HH:MM:SS.SSS	07:30:59.152 (date is 2000-01-01)
now	Current date and time in Universal Coordinated Time.
DDDD.DDDD	Julian day number as a floating-point number.

The character T in these formats is a literal character "T" separating the date and the time. Formats that only include a time assume the date 2001-01-01.

Modifiers

The time string can be followed by zero or more modifiers that alter the date or alter the interpretation of the date. The available modifiers are as follows:

NNN days	Number of days to add to the time.
NNN hours	Number of hours to add to the time.
NNN minutes	Number of minutes to add to the time.
NNN.NNNN seconds	Number of seconds and milliseconds to add to the time.
NNN months	Number of months to add to the time.
NNN years	Number of years to add to the time.
start of month	Shift time backwards to the start of the month.
start of year	Shift time backwards to the start of the year.
start of day	Shift time backwards to the start of the day.
weekday N	Forwards the time to the specified weekday. (0 = Sunday, 1 = Monday, and so forth).
localtime	Converts the date to local time.
utc	Converts the date to Universal Coordinated Time.

Operators

SQL supports a large selection of operators, including common operators that exist in most programming languages, as well as several operators that are unique to SQL.

Common operators

The following binary operators are allowed in a SQL block and are listed in order from highest to lowest precedence:

```
*      /      %  
+      -  
<< >> & |  
< >= > >=  
=      ==     !=     <> IN  
AND  
OR
```

Supported unary prefix operators are:

```
!      ~      NOT
```

The COLLATE operator can be thought of as a unary postfix operator. The COLLATE operator has the highest precedence. It always binds more tightly than any prefix unary operator or any binary operator.

Note that there are two variations of the equals and not equals operators. Equals can be either = or ==. The not-equals operator can be either != or <>.

The || operator is the string concatenation operator—it joins together the two strings of its operands.

The operator % outputs the remainder of its left operand modulo its right operand.

The result of any binary operator is a numeric value, except for the || concatenation operator which gives a string result.

SQL operators

LIKE

The LIKE operator does a pattern matching comparison.

```
expr      ::= (column-name | expr) LIKE pattern  
pattern   ::= '[ string | % | _ ]'
```

The operand to the right of the LIKE operator contains the pattern, and the left-hand operand contains the string to match against the pattern. A percent symbol (%) in the pattern is a wildcard character—it matches any sequence of zero or more characters in the string. An underscore (_) in the pattern matches any single character in the string. Any other character matches itself or its lower/upper case equivalent, that is, matches are performed in a case-insensitive manner. (Note: the database engine only understands upper/lower case for 7-bit Latin characters. Consequently, the LIKE operator is case sensitive for 8-bit iso8859 characters or UTF-8 characters. For example, the expression 'a' LIKE 'A' is TRUE but 'æ' LIKE 'Æ' is FALSE). Case sensitivity for Latin characters can be changed using the `SQLConnection.caseSensitiveLike` property.

If the optional ESCAPE clause is present, then the expression following the ESCAPE keyword must evaluate to a string consisting of a single character. This character may be used in the LIKE pattern to match literal percent or underscore characters. The escape character followed by a percent symbol, underscore or itself matches a literal percent symbol, underscore or escape character in the string, respectively.

GLOB

The GLOB operator is similar to LIKE but uses the Unix file globbing syntax for its wildcards. Unlike LIKE, GLOB is case sensitive.

IN

The IN operator calculates whether its left operand is equal to one of the values in its right operand (a set of values in parentheses).

```
in-expr      ::=  expr [NOT] IN ( value-list ) |  
              expr [NOT] IN ( select-statement ) |  
              expr [NOT] IN [database-name.] table-name  
value-list   ::=  literal-value [, literal-value]*
```

The right operand can be a set of comma-separated literal values, or it can be the result of a SELECT statement. See SELECT statements in expressions for an explanation and limitations on using a SELECT statement as the right-hand operand of the IN operator.

BETWEEN...AND

The BETWEEN...AND operator is equivalent to using two expressions with the >= and <= operators. For example, the expression `x BETWEEN y AND z` is equivalent to `x >= y AND x <= z`.

NOT

The NOT operator is a negation operator. The GLOB, LIKE, and IN operators may be preceded by the NOT keyword to invert the sense of the test (in other words, to check that a value does not match the indicated pattern).

Parameters

A parameter specifies a placeholder in the expression for a literal value that is filled in at runtime by assigning a value to the `SQLStatement.parameters` associative array. Parameters can take three forms:

	A question mark indicates an indexed parameter. Parameters are assigned numerical (zero-based) index values according to their order in the statement.
:AAAA	A colon followed by an identifier name holds a spot for a named parameter with the name AAAA. Named parameters are also numbered according to their order in the SQL statement. To avoid confusion, it is best to avoid mixing named and numbered parameters.
@AAAA	An "at sign" is equivalent to a colon.

Unsupported SQL features

The following is a list of the standard SQL elements that are not supported in Adobe AIR:

FOREIGN KEY constraints FOREIGN KEY constraints are parsed but are not enforced.

Triggers FOR EACH STATEMENT triggers are not supported (all triggers must be FOR EACH ROW). INSTEAD OF triggers are not supported on tables (INSTEAD OF triggers are only allowed on views). Recursive triggers—triggers that trigger themselves—are not supported.

ALTER TABLE Only the RENAME TABLE and ADD COLUMN variants of the ALTER TABLE command are supported. Other kinds of ALTER TABLE operations such as DROP COLUMN, ALTER COLUMN, ADD CONSTRAINT, and so forth are ignored.

Nested transactions Only a single active transaction is allowed.

RIGHT and FULL OUTER JOIN RIGHT OUTER JOIN or FULL OUTER JOIN are not supported.

Updateable VIEW A view is read only. You may not execute a DELETE, INSERT, or UPDATE statement on a view. An INSTEAD OF trigger that fires on an attempt to DELETE, INSERT, or UPDATE a view is supported and can be used to update supporting tables in the body of the trigger.

GRANT and REVOKE A database is an ordinary disk file; the only access permissions that can be applied are the normal file access permissions of the underlying operating system. The GRANT and REVOKE commands commonly found on client/server RDBMSes are not implemented.

The following SQL elements and SQLite features are supported in some SQLite implementations, but are not supported in Adobe AIR. Most of this functionality is available through methods of the SQLConnection class:

Transaction-related SQL elements (BEGIN, END, COMMIT, ROLLBACK) This functionality is available through the transaction-related methods of the SQLConnection class: SQLConnection.begin(), SQLConnection.commit(), and SQLConnection.rollback().

ANALYZE This functionality is available through the SQLConnection.analyze() method.

ATTACH This functionality is available through the SQLConnection.attach() method.

COPY This statement is not supported.

CREATE VIRTUAL TABLE This statement is not supported.

DETACH This functionality is available through the SQLConnection.detach() method.

PRAGMA This statement is not supported.

VACUUM This functionality is available through the SQLConnection.compact() method.

System table access is not available The system tables including sqlite_master and other tables with the "sqlite_" prefix are not available in SQL statements. The runtime includes a schema API that provides an object-oriented way to access schema data. For more information see the SQLConnection.loadSchema() method.

Regular-expression functions (MATCH() and REGEX()) These functions are not available in SQL statements.

The following functionality differs between many SQLite implementations and Adobe AIR:

Indexed statement parameters In many implementations indexed statement parameters are one-based. However, in Adobe AIR indexed statement parameters are zero-based (that is, the first parameter is given the index 0, the second parameter is given the index 1, and so forth).

INTEGER PRIMARY KEY column definitions In many implementations, only columns that are defined exactly as INTEGER PRIMARY KEY are used as the actual primary key column for a table. In those implementations, using another data type that is usually a synonym for INTEGER (such as int) does not cause the column to be used as the

internal primary key. However, in Adobe AIR, the `int` data type (and other `INTEGER` synonyms) are considered exactly equivalent to `INTEGER`. Consequently, a column defined as `int PRIMARY KEY` is used as the internal primary key for a table. For more information, see the sections `CREATE TABLE` and `Column affinity`.

Additional SQL features

The following column affinity types are not supported by default in SQLite, but are supported in Adobe AIR (Note that, like all keywords in SQL, these data type names are not case-sensitive):

Boolean corresponding to the Boolean class.

Date corresponding to the Date class.

int corresponding to the `int` class (equivalent to the `INTEGER` column affinity).

Number corresponding to the Number class (equivalent to the `REAL` column affinity).

Object corresponding to the Object class or any subclass that can be serialized and deserialized using AMF3. (This includes most classes including custom classes, but excludes some classes including display objects and objects that include display objects as properties.)

String corresponding to the String class (equivalent to the `TEXT` column affinity).

XML corresponding to the ActionScript (E4X) XML class.

XMLList corresponding to the ActionScript (E4X) XMLList class.

The following literal values are not supported by default in SQLite, but are supported in Adobe AIR:

true used to represent the literal boolean value `true`, for working with `BOOLEAN` columns.

false used to represent the literal boolean value `false`, for working with `BOOLEAN` columns.

Data type support

Unlike most SQL databases, the Adobe AIR SQL database engine does not require or enforce that table columns contain values of a certain type. Instead, the runtime uses two concepts, storage classes and column affinity, to control data types. This section describes storage classes and column affinity, as well as how data type differences are resolved under various conditions:

- [“Storage classes”](#) on page 363
- [“Column affinity”](#) on page 363
- [“Data types and comparison operators”](#) on page 366
- [“Data types and mathematical operators”](#) on page 366
- [“Data types and sorting”](#) on page 366
- [“Data types and grouping”](#) on page 366
- [“Data types and compound SELECT statements”](#) on page 367

Storage classes

Storage classes represent the actual data types that are used to store values in a database. The following storage classes are used by the database:

NULL The value is a NULL value.

INTEGER The value is a signed integer.

REAL The value is a floating-point number value.

TEXT The value is a text string (limited to 256 MB).

BLOB The value is a Binary Large Object (BLOB); in other words, raw binary data (limited to 256 MB).

All values supplied to the database as literals embedded in a SQL statement or values bound using parameters to a prepared SQL statement are assigned a storage class before the SQL statement is executed.

Literals that are part of a SQL statement are assigned storage class TEXT if they are enclosed by single or double quotes, INTEGER if the literal is specified as an unquoted number with no decimal point or exponent, REAL if the literal is an unquoted number with a decimal point or exponent and NULL if the value is a NULL. Literals with storage class BLOB are specified using the X'ABCD' notation. For more information, see *Literal values in expressions*.

Values supplied as parameters using the `SQLStatement.parameters` associative array are assigned the storage class that most closely matches the native data type bound. For example, `int` values are bound as INTEGER storage class, Number values are given the REAL storage class, String values are given the TEXT storage class, and `ByteArray` objects are given the BLOB storage class.

Column affinity

The *affinity* of a column is the recommended type for data stored in that column. When a value is stored in a column (through an INSERT or UPDATE statement), the runtime attempts to convert that value from its data type to the specified affinity. For example, if a Date value (an ActionScript or JavaScript Date instance) is inserted into a column whose affinity is TEXT, the Date value is converted to the String representation (equivalent to calling the object's `toString()` method) before being stored in the database. If the value cannot be converted to the specified affinity an error occurs and the operation is not performed. When a value is retrieved from the database using a SELECT statement, it is returned as an instance of the class corresponding to the affinity, regardless of whether it was converted from a different data type when it was stored.

If a column accepts NULL values, the ActionScript or JavaScript value `null` can be used as a parameter value to store NULL in the column. When a NULL storage class value is retrieved in a SELECT statement, it is always returned as the ActionScript or JavaScript value `null`, regardless of the column's affinity. If a column accepts NULL values, always check values retrieved from that column to determine if they're null before attempting to cast the values to a non-nullable type (such as Number or Boolean).

Each column in the database is assigned one of the following type affinities:

- TEXT (or String)
- NUMERIC
- INTEGER (or int)
- REAL (or Number)
- Boolean
- Date
- XML

- XMLLIST
- Object
- NONE

TEXT (or String)

A column with TEXT or String affinity stores all data using storage classes NULL, TEXT, or BLOB. If numerical data is inserted into a column with TEXT affinity it is converted to text form before being stored.

NUMERIC

A column with NUMERIC affinity contains values using storage classes NULL, REAL, or INTEGER. When text data is inserted into a NUMERIC column, an attempt is made to convert it to an integer or real number before it is stored. If the conversion is successful, then the value is stored using the INTEGER or REAL storage class (for example, a value of '10.05' is converted to REAL storage class before being stored). If the conversion cannot be performed an error occurs. No attempt is made to convert a NULL value. A value that's retrieved from a NUMERIC column is returned as an instance of the most specific numeric type into which the value fits. In other words, if the value is a positive integer or 0, it's returned as a uint instance. If it's a negative integer, it's returned as an int instance. Finally, if it has a floating-point component (it's not an integer) it's returned as a Number instance.

INTEGER (or int)

A column that uses INTEGER affinity behaves in the same way as a column with NUMERIC affinity, with one exception. If the value to be stored is a real value (such as a Number instance) with no floating point component or if the value is a text value that can be converted to a real value with no floating point component, it is converted to an integer and stored using the INTEGER storage class. If an attempt is made to store a real value with a floating point component an error occurs.

REAL (or Number)

A column with REAL or NUMBER affinity behaves like a column with NUMERIC affinity except that it forces integer values into floating point representation. A value in a REAL column is always returned from the database as a Number instance.

Boolean

A column with Boolean affinity stores true or false values. A Boolean column accepts a value that is an ActionScript or JavaScript Boolean instance. If code attempts to store a String value, a String with a length greater than zero is considered true, and an empty String is false. If code attempts to store numeric data, any non-zero value is stored as true and 0 is stored as false. When a Boolean value is retrieved using a SELECT statement, it is returned as a Boolean instance. Non-NULL values are stored using the INTEGER storage class (0 for false and 1 for true) and are converted to Boolean objects when data is retrieved.

Date

A column with Date affinity stores date and time values. A Date column is designed to accept values that are ActionScript or JavaScript Date instances. If an attempt is made to store a String value in a Date column, the runtime attempts to convert it to a Julian date. If the conversion fails an error occurs. If code attempts to store a Number, int, or uint value, no attempt is made to validate the data and it is assumed to be a valid Julian date value. A Date value that's retrieved using a SELECT statement is automatically converted to a Date instance. Date values are stored as Julian date values using the REAL storage class, so sorting and comparing operations work as you would expect them to.

XML or XMLList

A column that uses XML or XMMList affinity stores XML structures. When code attempts to store data in an XML column using a SQLStatement parameter the runtime attempts to convert and validate the value using the ActionScript XML() or XMMList() function. If the value cannot be converted to valid XML an error occurs. If the attempt to store the data uses a literal SQL text value (for example INSERT INTO (col1) VALUES ('Invalid XML (no closing tag)'), the value is not parsed or validated — it is assumed to be well-formed. If an invalid value is stored, when it is retrieved it is returned as an empty XML object. XML and XMMList Data is stored using the TEXT storage class or the NULL storage class.

Object

A column with Object affinity stores ActionScript or JavaScript complex objects, including Object class instances as well as instances of Object subclasses such as Array instances and even custom class instances. Object column data is serialized in AMF3 format and stored using the BLOB storage class. When a value is retrieved, it is deserialized from AMF3 and returned as an instance of the class as it was stored. Note that some ActionScript classes, notably display objects, cannot be deserialized as instances of their original data type. Before storing a custom class instance, you must register an alias for the class using the flash.net.registerClassAlias() method (or in Flex by adding [RemoteObject] metadata to the class declaration). Also, before retrieving that data you must register the same alias for the class. Any data that can't be deserialized properly, either because the class inherently can't be deserialized or because of a missing or mismatched class alias, is returned as an anonymous object (an Object class instance) with properties and values corresponding to the original instance as stored.

NONE

A column with affinity NONE does not prefer one storage class over another. It makes no attempt to convert data before it is inserted.

Determining affinity

The type affinity of a column is determined by the declared type of the column in the CREATE TABLE statement. When determining the type the following rules (not case-sensitive) are applied:

- If the data type of the column contains any of the strings "CHAR", "CLOB", "STRI", or "TEXT" then that column has TEXT/String affinity. Notice that the type VARCHAR contains the string "CHAR" and is thus assigned TEXT affinity.
- If the data type for the column contains the string "BLOB" or if no data type is specified then the column has affinity NONE.
- If the data type for column contains the string "XMML" then the column has XMMList affinity.
- If the data type is the string "XML" then the column has XML affinity.
- If the data type contains the string "OBJE" then the column has Object affinity.
- If the data type contains the string "BOOL" then the column has Boolean affinity.
- If the data type contains the string "DATE" then the column has Date affinity.
- If the data type contains the string "INT" (including "UINT") then it is assigned INTEGER/int affinity.
- If the data type for a column contains any of the strings "REAL", "NUMB", "FLOA", or "DOUB" then the column has REAL/Number affinity.
- Otherwise, the affinity is NUMERIC.
- If a table is created using a CREATE TABLE t AS SELECT... statement then all columns have no data type specified and they are given the affinity NONE.

Data types and comparison operators

The following binary comparison operators =, <, <=, >= and != are supported, along with an operation to test for set membership, IN, and the ternary comparison operator BETWEEN. For details about these operators see Operators.

The results of a comparison depend on the storage classes of the two values being compared. When comparing two values the following rules are applied:

- A value with storage class NULL is considered less than any other value (including another value with storage class NULL).
- An INTEGER or REAL value is less than any TEXT or BLOB value. When an INTEGER or REAL is compared to another INTEGER or REAL, a numerical comparison is performed.
- A TEXT value is less than a BLOB value. When two TEXT values are compared, a binary comparison is performed.
- When two BLOB values are compared, the result is always determined using a binary comparison.

The ternary operator BETWEEN is always recast as the equivalent binary expression. For example, a BETWEEN b AND c is recast to a >= b AND a <= c, even if this means that different affinities are applied to a in each of the comparisons required to evaluate the expression.

Expressions of the type a IN (SELECT b ...) are handled by the three rules enumerated previously for binary comparisons, that is, in a similar manner to a = b. For example, if b is a column value and a is an expression, then the affinity of b is applied to a before any comparisons take place. The expression a IN (x, y, z) is recast as a = +x OR a = +y OR a = +z. The values to the right of the IN operator (the x, y, and z values in this example) are considered to be expressions, even if they happen to be column values. If the value of the left of the IN operator is a column, then the affinity of that column is used. If the value is an expression then no conversions occur.

How comparisons are performed can also be affected by the use of a COLLATE clause. For more information, see COLLATE.

Data types and mathematical operators

For each of the supported mathematical operators, *, /, %, +, and -, numeric affinity is applied to each operand before evaluating the expression. If any operand cannot be converted to the NUMERIC storage class successfully the expression evaluates to NULL.

When the concatenation operator || is used each operand is converted to the TEXT storage class before the expression is evaluated. If any operand cannot be converted to the TEXT storage class then the result of the expression is NULL. This inability to convert the value can happen in two situations, if the value of the operand is NULL, or if it's a BLOB containing a non-TEXT storage class.

Data types and sorting

When values are sorted by an ORDER BY clause, values with storage class NULL come first. These are followed by INTEGER and REAL values interspersed in numeric order, followed by TEXT values in binary order or based on the specified collation (BINARY or NOCASE). Finally come BLOB values in binary order. No storage class conversions occur before the sort.

Data types and grouping

When grouping values with the GROUP BY clause, values with different storage classes are considered distinct. An exception is INTEGER and REAL values which are considered equal if they are numerically equivalent. No affinities are applied to any values as the result of a GROUP BY clause.

Data types and compound SELECT statements

The compound SELECT operators UNION, INTERSECT, and EXCEPT perform implicit comparisons between values. Before these comparisons are performed an affinity may be applied to each value. The same affinity, if any, is applied to all values that may be returned in a single column of the compound SELECT result set. The affinity that is applied is the affinity of the column returned by the first component SELECT statement that has a column value (and not some other kind of expression) in that position. If for a given compound SELECT column none of the component SELECT statements return a column value, no affinity is applied to the values from that column before they are compared.

Chapter 27: SQL error detail messages, ids, and arguments

The `SQLException` class represents various errors that can occur while working with an Adobe AIR local SQL database. For any given exception, the `SQLException` instance has a `details` property containing an English error message. In addition, each error message has an associated unique identifier that is available in the `SQLException` object's `detailID` property. Using the `detailID` property, an application can identify the specific details error message. The application can provide alternate text for the end user in the language of his or her locale. The argument values in the `detailArguments` array can be substituted in the appropriate position in the error message string. This is useful for applications that display the `details` property error message for an error directly to end users in a specific locale.

The following table contains a list of the `detailID` values and the associated English error message text. Placeholder text in the messages indicates where `detailArguments` values are substituted in by the runtime. This list can be used as a source for localizing the error messages that can occur in SQL database operations.

SQLException detailID	English error detail message and parameters
1001	Connection closed.
1102	Database must be open to perform this operation.
1003	%s [,and %s] parameter name(s) found in parameters property but not in the SQL specified.
1004	Mismatch in parameter count. Found %d in SQL specified and %d value(s) set in parameters property. Expecting values for %s [,and %s].
1005	Auto compact could not be turned on.
1006	The pageSize value could not be set.
1007	The schema object with name '%s' of type '%s' in database '%s' was not found.
1008	The schema object with name '%s' in database '%s' was not found.
1009	No schema objects with type '%s' in database '%s' were found.
1010	No schema objects in database '%s' were found.
2001	Parser stack overflow.
2002	Too many arguments on function '%s'
2003	near '%s': syntax error
2004	there is already another table or index with this name: '%s'
2005	PRAGMA is not allowed in SQL.
2006	Not a writable directory.
2007	Unknown or unsupported join type: '%s %s %s'
2008	RIGHT and FULL OUTER JOINS are not currently supported.
2009	A NATURAL join may not have an ON or USING clause.
2010	Cannot have both ON and USING clauses in the same join.
2011	Cannot join using column '%s' - column not present in both tables.
2012	Only a single result allowed for a SELECT that is part of an expression.
2013	No such table: '[%s.]%s'
2014	No tables specified.
2015	Too many columns in result set too many columns on '%s'.
2016	%s ORDER GROUP BY term out of range - should be between 1 and %d
2017	Too many terms in ORDER BY clause.
2018	%s ORDER BY term out of range - should be between 1 and %d.
2019	%r ORDER BY term does not match any column in the result set.
2020	ORDER BY clause should come after '%s' not before.
2021	LIMIT clause should come after '%s' not before.
2022	SELECTs to the left and right of '%s' do not have the same number of result columns.
2023	A GROUP BY clause is required before HAVING.
2024	Aggregate functions are not allowed in the GROUP BY clause.
2025	DISTINCT in aggregate must be followed by an expression.

2026	Too many terms in compound SELECT.
2027	Too many terms in ORDER GROUP BY clause
2028	Temporary trigger may not have qualified name
2030	Trigger '%s' already exists
2032	Cannot create BEFORE AFTER trigger on view: '%s'.
2033	Cannot create INSTEAD OF trigger on table: '%s'.
2034	No such trigger: '%s'
2035	Recursive triggers not supported ('%s').
2036	No such column: %s[.%s[.%s]]
2037	VACUUM is not allowed from SQL.
2043	Table '%s': indexing function returned an invalid plan.
2044	At most %d tables in a join.
2046	Cannot add a PRIMARY KEY column.
2047	Cannot add a UNIQUE column.
2048	Cannot add a NOT NULL column with default value NULL.
2049	Cannot add a column with non-constant default.
2050	Cannot add a column to a view.
2051	ANALYZE is not allowed in SQL.
2052	Invalid name: '%s'
2053	ATTACH is not allowed from SQL.
2054	%s '%s' cannot reference objects in database '%s'
2055	Access to '[%s.]%s.%s' is prohibited.
2056	Not authorized.
2058	No such view: '[%s.]%s'
2060	Temporary table name must be unqualified.
2061	Table '%s' already exists.
2062	There is already an index named: '%s'
2064	Duplicate column name: '%s'
2065	Table '%s' has more than one primary key.
2066	AUTOINCREMENT is only allowed on an INTEGER PRIMARY KEY
2067	No such collation sequence: '%s'
2068	Parameters are not allowed in views.
2069	View '%s' is circularly defined.
2070	Table '%s' may not be dropped.
2071	Use DROP VIEW to delete view '%s'
2072	Use DROP TABLE to delete table '%s'
2073	Foreign key on '%s' should reference only one column of table '%s'
2074	Number of columns in foreign key does not match the number of columns in the referenced table.
2075	Unknown column '%s' in foreign key definition.
2076	Table '%s' may not be indexed.
2077	Views may not be indexed.
2080	Conflicting ON CONFLICT clauses specified.
2081	No such index: '%s'
2082	Index associated with UNIQUE or PRIMARY KEY constraint cannot be dropped.
2083	BEGIN is not allowed in SQL.
2084	COMMIT is not allowed in SQL.
2085	ROLLBACK is not allowed in SQL.
2086	Unable to open a temporary database file for storing temporary tables.
2087	Unable to identify the object to be reindexed.
2088	Table '%s' may not be modified.
2089	Cannot modify '%s' because it is a view.
2090	Variable number must be between ?0 and ?%d<
2092	Misuse of aliased aggregate '%s'
2093	Ambiguous column name: [%s.[%s.]]%s'
2094	No such function: '%s'
2095	Wrong number of arguments to function '%s'
2096	Subqueries prohibited in CHECK constraints.
2097	Parameters prohibited in CHECK constraints.

2098	Expression tree is too large (maximum depth %d)
2099	RAISE() may only be used within a trigger-program
2100	Table '%s' has %d columns but %d values were supplied
2101	Database schema is locked: '%s'
2102	Statement too long.
2103	Unable to delete/modify collation sequence due to active statements
2104	Too many attached databases - max %d
2105	Cannot ATTACH database within transaction.
2106	Database '%s' is already in use.
2108	Attached databases must use the same text encoding as main database.
2200	Out of memory.
2201	Unable to open database.
2202	Cannot DETACH database within transaction.
2203	Cannot detach database: '%s'
2204	Database '%s' is locked.
2205	Unable to acquire a read lock on the database.
2206	[column columns] '%s', '%s' are not [unique is] not unique.
2207	Malformed database schema.
2208	Unsupported file format.
2209	Unrecognized token: '%s'
2300	Could not convert text value to numeric value.
2301	Could not convert string value to date.
2302	Could not convert floating point value to integer without loss of data.
2303	Cannot rollback transaction - SQL statements in progress.
2304	Cannot commit transaction - SQL statements in progress.
2305	Database table is locked: '%s'
2306	Read-only table.
2307	String or blob too big.
2309	Cannot open indexed column for writing.
2400	Cannot open value of type %s.
2401	No such rowid: %s<
2402	Object name reserved for internal use: '%s'
2403	View '%s' may not be altered.
2404	Default value of column '%s' is not constant.
2405	Not authorized to use function '%s'
2406	Misuse of aggregate function '%s'
2407	Misuse of aggregate: '%s'
2408	No such database: '%s'
2409	Table '%s' has no column named '%s'
2501	No such module: '%s'
2508	No such savepoint: '%s'
2510	Cannot rollback - no transaction is active.
2511	Cannot commit - no transaction is active.