

# Adobe® Integrated Runtime (AIR™)

## Developing AIR Applications with Adobe Flex (Beta 1)



© 2007 Adobe Systems Incorporated. All rights reserved.

## Developing AIR Applications Using Flex™

If this guide is distributed with software that includes an end-user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end-user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names in sample templates are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, the Adobe logo, Flex, Flex Builder and Flash Player are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

ActiveX and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and other countries. Macintosh is a trademark of Apple Inc., registered in the United States and other countries. All other trademarks are the property of their respective owners.

Speech compression and decompression technology licensed from Nellymoser, Inc. ([www.nellymoser.com](http://www.nellymoser.com)).



Sorenson™ Spark™ video compression and decompression technology licensed from Sorenson Media, Inc.

Opera® browser Copyright © 1995-2002 Opera Software ASA and its suppliers. All rights reserved.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA

Notice to U.S. government end users. The software and documentation are "Commercial Items," as that term is defined at 48 C.F.R. §2.101, consisting of "Commercial Computer Software" and "Commercial Computer Software Documentation," as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA. For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.



# Chapter 2: Getting started with Adobe AIR

This chapter contains the following topics:

- [About Adobe AIR](#)
- [Installing the runtime and sample applications](#)
- [Setting up your development environment](#)
- [Distributing, installing, and running AIR applications](#)
- [About AIR security](#)
- [AIR file structures](#)

## About Adobe AIR

Adobe® Integrated Runtime (AIR™) is a cross-operating system runtime that allows you to leverage your existing web development skills (Flash, Flex, HTML, JavaScript, Ajax) to build and deploy Rich Internet Applications (RIAs) to the desktop.

AIR enables you to work in familiar environments, to leverage the tools and approaches you find most comfortable, and by supporting Flash, Flex, HTML, JavaScript, and Ajax, to build the best possible experience that meets your needs.

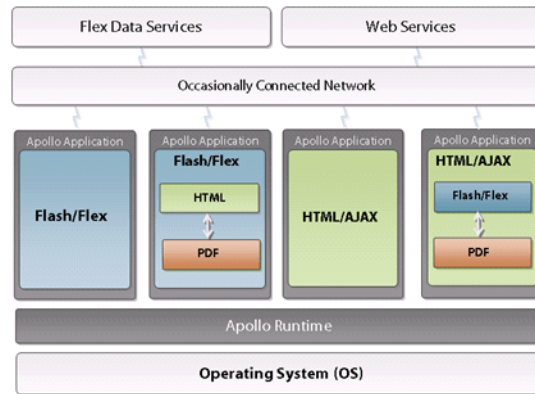
For example, applications can be developed using one or a combination of technologies below:

- Flash / Flex / ActionScript
- HTML / JavaScript / CSS / Ajax
- PDF can be leveraged with any application

As a result, AIR applications can be:

- Based on Flash or Flex: Application whose root content is Flash/Flex (SWF)
- Based on Flash or Flex with HTML or PDF. Applications whose root content is Flash/Flex (SWF) with HTML (HTML, JS, CSS) or PDF content included
- HTML-based. Application whose root content is HTML, JS, CSS
- HTML-based with Flash/Flex or PDF. Applications whose root content is HTML with Flash/Flex (SWF) or PDF content included

Users interact with AIR applications in the same way that they interact with native desktop applications. The runtime is installed once on the user's computer, and then AIR applications are installed and run just like any other desktop application.



Since AIR is an application runtime, it has little or no visible user interface and you have complete control over the application and the experience it provides to users. The runtime provides a consistent cross-operating system platform and framework for deploying applications and therefore eliminates cross-browser testing by ensuring consistent functionality and interactions across desktops. Instead of developing for a specific operating system, you target the runtime. This has a number of benefits:

- Applications developed for AIR run across multiple operating systems without any additional work by you. The runtime ensures consistent and predictable presentation and interactions across all the operating systems supported by AIR.
- Applications are built faster by enabling you to leverage existing web technologies and design patterns. This allows you to extend your web based applications to the desktop without learning traditional desktop development technologies or the complexity of native code. This is easier than using lower level languages such as C and C++, and does away with the need to learn complex low-level APIs specific to each operating system.

When developing applications for AIR, you can leverage a rich set of frameworks and APIs:

- APIs specific to AIR provided by the runtime, and the AIR framework
- ActionScript APIs used in SWF files and Flex framework (as well as other ActionScript based libraries and frameworks)

AIR delivers a new paradigm that dramatically changes how applications can be created, deployed, and experienced. You gain more creative control and can extend your Flash, Flex, HTML, and Ajax-based applications to the desktop, without learning traditional desktop development technologies.

## Installing the runtime and sample applications

AIR allows you to run rich Internet applications on the desktop. To begin, you must install the runtime on your computer. Once you've installed the runtime, download the sample applications to see AIR in action and to evaluate the underlying code.

## Install the runtime

Use the following instructions to download and install the Windows and Mac OS X versions of AIR. You only need to install the runtime once for each computer that will run AIR applications.

### Install the runtime on a Windows computer

- 1 Download the runtime installation file (AIR1\_win\_beta1.exe) from the Adobe Labs site.
- 2 Double-click the AIR1\_win\_beta1.exe file.
- 3 In the installation window, follow the prompts to complete the installation.
- 4 If you have created or modified the mms.cfg file used by Flash Player, remove it while running AIR applications. On Windows, the file is located in the Macromedia Flash Player folder within the system directory (for example, C:\winnt\system32\macromed\flash\mms.cfg on a default Windows XP installation).

### Install the runtime on Mac OS

- 1 Download the runtime installation file (AIR1\_mac\_beta1.dmg) from the Adobe Labs site.
- 2 Double-click the AIR1\_mac\_beta1.dmg file.
- 3 In the installation window, follow the prompts to complete the installation.
- 4 If the Installer displays an Authenticate window, enter your Mac OS user name and password.
- 5 If you have created or modified the mms.cfg file used by Flash Player, remove it while running AIR applications. On Mac OS, the file is located at /Library/Application Support/Macromedia/mms.cfg.

## Install and run the AIR sample applications

The beta 1 release of AIR includes a number of sample applications.

- 1 Download the sample AIR file from the Adobe Labs site.
- 2 Double-click the AIR file.
- 3 In the Installation window, select installation options, and then click Continue.
- 4 After the installation is complete, open the application:
  - On Windows, double-click the application's icon on the desktop or select it from the Windows Start menu.
  - On Mac OS, double-click the application's icon, which is installed in the Applications sub-directory of your user directory (for example, in Macintosh HD/Users/JoeUser/Applications/) by default.

The application opens in the runtime environment.

**Note:** Check the AIR Beta 1 release notes for updates to these instructions.

## Run an AIR application

Once you have installed the runtime and the AIR application you want to run, running the application is as simple as running any other desktop application:

- On a Windows computer, double-click the application's icon (which may be installed on the desktop or in a folder), or select the application from the Start menu.
- On Mac OS, double-click the application in the folder in which it was installed. The default installation directory is the Applications subdirectory of the user directory.

## Setting up your development environment

Before you build your first AIR application, you need to set up your development environment. You can build Flex and ActionScript-based AIR applications in Flex Builder 3, or you can use the command line tools contained in the Flex and AIR SDKs. If you're developing AIR applications in HTML, you'll need the AIR SDK, which contains the tools you need to package your applications.

To locate, download, install and configure the tools you need to develop AIR applications, see these instructions:

- [“Setting up for Flex Builder” on page 12](#)
- [“Setting up for Flex SDK” on page 14](#)

**Note:** Check the AIR Beta 1 release notes for updates to these instructions.

## Distributing, installing, and running AIR applications

AIR applications are easy to install and run. The *seamless install* feature lets users install the latest version of the AIR runtime (if it is not installed) when clicking the link to install a specific AIR application. Once the AIR application is installed, users simply double-click the application icon to run it, just like any other desktop application.

Once you package an AIR application (Flex Builder users see [“Package AIR applications with Flex Builder” on page 26](#), SDK users see [“Packaging an AIR application using the AIR Developer Tool” on page 34](#)), there are a couple of ways to distribute it:

- You can install it by sending the AIR package to the end user just as you would distribute any file. For example, you can send the AIR package as an e-mail attachment or as a link in a web page.
- You can add a *seamless install* installation link in a web page. The seamless install feature lets you provide a link in a web page that lets the user install an AIR application by simply clicking the link. If the AIR runtime is not installed, the user is given the option to install it. The seamless install feature also lets users install the AIR application without downloading the AIR file to their machine.

For details, see [“Distributing, installing, and running AIR applications” on page 1](#).

If the user downloads the AIR file, then the user double-clicks the AIR file to launch the AIR application installer. If the user clicks a seamless install link in a web page, a dialog box lets the user confirm that they want to install the application before launching the AIR application installer.

In Windows, with the default settings selected, the AIR application installer does the following:

- Installs the application into the Program Files directory
- Creates a desktop shortcut for application
- Creates a Start Menu shortcut
- Adds an entry for application in the Add / Remove Programs Control Panel

In the Mac OS, by default the AIR application installer adds the application to the Applications subdirectory of the user directory.

If the application is already installed, the installer gives the user the choice of opening the existing version of the application or updating to the version in the downloaded AIR file. The installer identifies the application using the application ID (`appId`) in the AIR file.



An application can also install a new version via ActionScript or JavaScript. For more information, see [“Updating applications programatically” on page 75](#).

Once a user has installed the runtime and the AIR application, running the application is as simple as running any other desktop application:

- On a Windows computer, double-click the application's icon (which may be installed on the desktop or in a folder), or select the application from the Start menu.
- On Mac OS, double-click the application in the folder in which it was installed. The default installation directory is the Applications subdirectory of the user directory.

## About AIR security

The AIR environment gives you access to some of the same operating system resources that you can access from a standard desktop application. For this reason, an AIR application has fewer access restrictions than a SWF or HTML file running in a browser. Since this can present a security risk, it's important to understand the AIR application security model.

### Installer security warnings

During the AIR application installation process, users see a security notice, intended to provide information about the publisher of the application and the type of system access permitted to the application. This information lets users make an informed decision about installing the application.

### Security sandboxes

In the Beta release of AIR, local files that are part of an AIR application are accessed from a special security sandbox, known as the AIR application security sandbox. In future releases of AIR, application resources may have different sandboxes depending on how the AIR application is signed or installed.

You can use the read-only `Security.sandboxType` property to determine the security sandbox for a SWF file. For a SWF file that is bundled with the AIR application, this property is set to the value defined by the `Security.APPLICATION` constant.

All other resources—those that are not installed with the AIR application—are put in the same security sandboxes as they would be placed in if they were running in Flash Player in a web browser. Remote resources are put in sandboxes according to their source domains, and local resources are put in the local-with-networking, local-with-filesystem, or local-trusted sandbox.

### Privileges for resources in the AIR application security sandbox

SWF files in the AIR application sandbox can cross-script any SWF file from any domain. However, by default, SWF files outside of the AIR application security sandbox are restricted from cross-scripting the SWF file in the AIR application security sandbox.

SWF files and HTML content in the AIR application sandbox can load content and data from any domain.

SWF files installed with AIR applications do not need to look for cross-domain policy files. Capabilities that normally require another SWF file to grant access by calling the `Security.allowDomain()` method are not restricted to SWF files installed in AIR applications.

AIR provides enhanced capabilities for SWF files and HTML content in the AIR application security sandbox—capabilities that are not available to SWF files and HTML content running in a web browser (in Flash Player). These include reading and writing to local resources and files.

The settings in the Flash Player Settings UI do not apply to resources installed with AIR applications.

## Best practices for developing secure applications

While building AIR applications, be aware that although you are using web technologies, you 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 intentionally or unintentionally. AIR attempts to minimize this risk, but there are still ways where vulnerabilities can be introduced.

The greatest area of risk for unintentionally introducing a security problem to an AIR application is when the application uses external data or content — so you must take special care when using data from the network or file system. The following are examples of areas of potential risk:

**Importing content into the application** This can lead to script injections:

- If a TextField object loads content that includes a link, the link may run with unintended privilege.
- If an application loads a SWF from an untrusted source, that SWF may run with the unintended privilege.
- If an application loads JSON content from outside the application, that content may access the runtime capabilities.

**Data that influences application behavior** This could lead to a security vulnerability. For example, if an application uses data from a network source to determine a file name or write to a configuration file then that data needs to be validated to make sure that it is safe or that it comes from a trusted source.

This is an beta version of the product, so if you encounter configurations or options that might lead to potential vulnerabilities, please let Adobe know. Also, Adobe is currently working on a broad set of security best practices that will be provided to you prior to the 1.0 release of AIR.

## Security restrictions for HTML content

HTML content operates under the same security model as other types of AIR content. However, there are a few additional considerations. HTML content in an HTMLControl object can only access the security-restricted runtime classes (via the `window.runtime` JavaScript object) if the content is within the application security sandbox. For HTML-based applications, the top-level frame of an HTML-based application always has access to the runtime classes since it is loaded from the app-resource directory. Content loaded from outside the app-resource directory, whether in a sub-frame (or IFRAME), or loaded by changing the page location, is placed in a security sandbox corresponding to its domain of origin and cannot access the security-restricted AIR runtime classes. By default, non-application content is also not allowed to cross-script application content. Thus references to the JavaScript window properties, `nativeWindow` and `htmlControl` will not work outside the application sandbox. To allow safe cross-scripting, you can use the `flash.system.Door` API to create a guarded communication gateway that provides a limited interface between application content and non-application content.

### See also

[“Scripting between content in different domains” on page 67](#)

The AIR Security Whitepaper

## AIR file structures

Aside from all the files and other assets that make up your AIR applications, the following two files are required for AIR applications:

**AIR files** Used to package an AIR application, and it serves as the installer file.

**The application descriptor file** An XML file, included in each AIR application (embedded in the AIR file) defines various properties of the application, such as the application name, appID, and the characteristics of the main application window.

When using Flex Builder and the AIR Extensions, the application.xml file is automatically generated for you when you create a new AIR project. If you're developing AIR applications using the Flex and AIR SDKs (including when developing HTML-based applications), you need to create this file manually. Additionally, when using the AIR Extensions for Flex Builder, you export your application as an AIR file and it is generated for you. When developing with the Flex and AIR SDKs, you use the ADT command-line tool to generate the AIR file.

For more information, see [“Setting application properties” on page 42](#).

# Chapter 3: Setting up for Flex Builder

To develop Adobe® Integrated Runtime (AIR™) applications with Flex, you have the following options:

- You can download and install Adobe Flex Builder 3, which provides integrated tools to create Adobe AIR projects and test, debug, and package your AIR applications.
- You can download the Adobe Flex 3 SDK and develop Flex AIR applications using your favorite text editor and the command-line tools.

This chapter contains the following sections:

[About AIR support in Flex Builder](#)

[Downloading Flex Builder 3](#)

[Moving from Flex Builder 2.0.1 to Flex Builder 3](#)

## See also

[“Setting up for Flex SDK” on page 14](#)

## About AIR support in Flex Builder

AIR support in Flex Builder includes the following:

- A wizard for creating new AIR application projects
- Automatic creation and management of the application.xml file
- Running and debugging of AIR applications from Flex Builder
- Automated exporting of your AIR application project to an AIR file for distribution to users

## See also

[“Creating your first Flex AIR application in Flex Builder” on page 17](#)

[“Developing AIR applications with Flex Builder” on page 25](#)

[“Using the Flex AIR components” on page 27](#)

## Downloading Flex Builder 3

You can download Flex Builder 3 from Adobe Labs. After you’ve downloaded the installation file, begin the Flex Builder 3 installation process and follow the prompts.

**Note:** *If you already have an earlier version of Flex Builder installed, you can either install the Beta 1 version over it (not recommended) or you can install Flex Builder 3 separately and use it to develop AIR applications during this beta phase.*

## Moving from Flex Builder 2.0.1 to Flex Builder 3

AIR applications developed using Flex Builder 2.0.1 and the Alpha version of AIR (referred to by its code name Apollo), are not supported in Flex and Flex Builder 3.

### Use Flex Builder to port AIR applications to the new version

- 1 Create an AIR project in Flex Builder. See [“Developing AIR applications with Flex Builder” on page 25](#).
- 2 Copy and paste the application code from your old Apollo project’s main MXML file into the new AIR project MXML file. Note that the AIR application container (ApolloApplication) has been changed to WindowedApplication.
- 3 Add your project’s assets into the new project using the Flex Builder project navigator.
- 4 Update any remaining code that refers to the ApolloApplication container.

### CLOSE PROCEDURE

### Use the Flex SDK to port AIR applications to the new version

- 1 Recreate the application.xml file. For details of this file format, see [“Setting application properties” on page 42](#).
- 2 Update your code with the new name for the AIR application container (WindowedApplication).

### CLOSE PROCEDURE

# Chapter 4: Setting up for Flex SDK

Most of the command-line tools you use to create Adobe® Integrated Runtime (AIR™) applications are the same tools used to build Flex applications. The Flex SDK tools and their command-line options are fully described in [Building and Deploying Flex 3 Applications](http://www.adobe.com/go/learn_flex3_building) ([http://www.adobe.com/go/learn\\_flex3\\_building](http://www.adobe.com/go/learn_flex3_building)) in the Flex 3 documentation library. The Flex SDK tools are described here at a basic level to help you get started and to point out the differences between building Flex and building AIR applications.

This chapter contains the following sections:

[Install and configure the Flex 3 SDK in Windows](#)

[Compiler setup](#)

[Debugger setup](#)

[Application packager setup](#)

[Disabling mms.cfg settings](#)

## See also

[“Creating an AIR application using the command line tools” on page 30](#)

## Install and configure the Flex 3 SDK in Windows

This section contains the following topics:

[“Compiler setup” on page 15](#)

[“Compiler configuration files” on page 15](#)

[“Debugger setup” on page 15](#)

[“Application packager setup” on page 15](#)

The Adobe AIR command-line tools require that Java be installed on your computer. You can use the Java virtual machine from either the JRE or the JDK (version 1.4.2 or newer). This is installed with Flex Builder on Mac OS X. The Java JRE is available at <http://java.sun.com/j2se/1.4.2/download.html>. The Java JDK is available at <http://java.sun.com/javase/downloads/index.jsp>.

**Note:** *Java is not required for end users to run AIR applications.*

The Flex SDK provides you with the AIR API and command-line tools that you use to package, compile, and debug your AIR applications.

- 1 If you haven't already done so, download the Flex 3 SDK, which is available on Adobe Labs.
- 2 Unzip the SDK into a folder (for example, Flex 3 SDK).
- 3 The command lines are located in the bin folder.

## CLOSE PROCEDURE

## Compiler setup

Two compilers are included with the Flex SDK. The mxmcl compiler creates SWF files from MXML and ActionScript code. The compc compiler creates SWC files for components and libraries. Both compilers can be run from the command line as either native or Java programs. (The native executable files call the Java programs.) If you plan to use the native versions, you might want to add the Flex 3 SDK\bin directory to your system path. If you plan to use the Java versions directly, you might want to add mxmcl.jar and compc.jar to your environment variable.

## Compiler configuration files

You typically specify compilation options both on the command line and with one or more configuration files. The global Flex SDK configuration file contains default values that are used whenever the compilers are run. You can edit this file to suit your own development environment. The air\_config.xml global configuration file is located in the frameworks directory of your Flex 3 SDK installation.

**Note:** The air\_config.xml is used instead of flex\_config.xml when the amxmcl command is used to start the compiler.

The default configuration values are suitable for discovering how Flex and AIR work, but you should examine the available options more closely when you embark on a full-scale project. You can supply project-specific values for the compiler options in a local configuration file that will take precedence over the global values for a given project. For a full list of the compilation options and for the syntax of the configuration files, see [About configuration files](http://livedocs.macromedia.com/flex/2/docs/00001490.html) (<http://livedocs.macromedia.com/flex/2/docs/00001490.html>) in the Flex 2 documentation library.

**Note:** No compilation options are used specifically for AIR applications, but you do need to reference the AIR libraries when compiling an AIR application. Typically, these libraries are referenced in a project-level configuration file, in a tool for a build tool such as Ant, or directly on the command line.

## See also

[“Creating an AIR application using the command line tools” on page 30](#)

## Debugger setup

AIR supports debugging directly, so you do not need a debug version of the runtime (as you would with Flash Player). To conduct command-line debugging, you use the Flash Debugger and, optionally, the AIR Debug Launcher. The only setup required is to set your path or environment variables so that you can conveniently launch these programs.

The Flash Debugger is distributed in the Flex 3 SDK directory. The native versions, for example fdb.exe on Windows, are in the bin subdirectory. The Java version is in the lib subdirectory. The AIR Debug Launcher, adl.exe or ADL, is in the bin directory of your Flex SDK installation (there is no separate Java version).

**Note:** You cannot start an AIR application directly with FDB, because FDB attempts to launch it with Flash Player. Instead, you must let the AIR application connect to a running FDB session.

## See also

[“Debugging using the AIR Debug Launcher” on page 33](#)

## Application packager setup

The AIR Developer Tool (ADT), which packages your application into an installable AIR file, is a Java program. To run ADT, Java must be installed and your system path set so that you can run Java from the command line.

The SDK includes a script file for executing ADT. To run the ADT script from the command-line, add the bin directory of the Flex SDK to your system path.

You can also run ADT as a Java program, which may be convenient when using build tools such as Apache Ant.

**See also**

[“Packaging an AIR application using the AIR Developer Tool” on page 34](#)

## Disabling mms.cfg settings

If you modified the mms.cfg file (a security settings file that Flash Player uses), remove it while you test AIR. In the M2 release of AIR, some of the settings in this configuration file might restrict AIR functionality.

- On Mac OS, the file is located at /Library/Application Support/Macromedia/mms.cfg.
- On Microsoft Windows, the file is located in the Macromedia Flash Player folder within the system directory (for example, C:\winnt\system32\macromed\flash\mms.cfg on a default Windows XP installation).



# Chapter 5: Creating your first Flex AIR application in Flex Builder

For a quick, hands-on illustration of how Adobe® Integrated Runtime (AIR™) works, use these instructions to create and package a simple SWF file-based AIR “Hello World” application using Adobe Flex Builder 3.

This chapter contains the following sections:

[Create an AIR project](#)

[Write the application code](#)

[Test the application](#)

[Package and run your application](#)

## See also

[“Installing the runtime and sample applications” on page 6](#)

[“Setting up for Flex Builder” on page 12](#)

## Create an AIR project

To begin, you must have installed Adobe AIR and set up your development environment. See the following sections:

### Create the project in Flex Builder

- 1 Open Flex Builder 3.
- 2 Select File | New | AIR Project (if you are using the Eclipse plug-in configuration, select File | New | Other, expand the Flex node, and select AIR Project).
- 3 Accept the default project location and Flex SDK version, and then click Next.
- 4 In the next page of the dialog box (for setting build paths), make no changes, and click Next.
- 5 Specify the following settings for the AIR application, and then click Finish:

**ID** AIRHelloWorld

**Name** Hello World

**Description** A test AIR application

**Copyright** 2007

The Description and Copyright fields are optional for this Hello World example. However, set the other fields with the values shown.

### CLOSE PROCEDURE

**Set the application's window transparency**

- 1 Open the AIRHelloWorld-app.xml file from the Project Navigator. The file is opened in the Flex Builder text editor.
- 2 In the `rootContent` node of the application, set the `systemChrome` and `transparent` attributes to the following:  
`systemChrome="standard" transparent="false"`
- 3 Save your changes, and then close the AIRHelloWorld-app.xml file.

**CLOSE PROCEDURE**

## Write the application code

To write the “Hello World” application’s code, you edit the application’s MXML file (AIRHelloWorld.mxml), which should be open in the editor. If it isn’t, use the Project Navigator to open the file.

All Flex AIR applications are contained within the MXML `WindowedApplication` tag, which creates a simple window that includes basic window controls such as a title bar and close button.

**Add the code**

- 1 Add a `title` attribute to the `WindowedApplication` component, and assign it the value “Hello World”:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
title="Hello World">

</mx:WindowedApplication>
```

- 2 Add a `Label` component to the application (place it inside the `WindowedApplication` tag), set the `text` property of the `Label` component to “Hello AIR”, and set the layout constraints to always keep it centered, as shown here:

```
<?xml version="1.0" encoding="utf-8"?><?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
title="Hello World">
    <mx:Label text="Hello AIR" horizontalCenter="0" verticalCenter="0"/>
</mx:WindowedApplication>
```

- 3 Add the following style block:

```
<mx:Style>
    Application
    {
        background-image:"";
        background-color:"";
        background-alpha:"0.5";
    }
</mx:Style>
```


These style settings apply to the entire application and render the window background a slightly transparent gray.

The entire application code should now look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
title="Hello World">
  <mx:Style>
    Application
    {
      background-image: "";
      background-color: "";
      background-alpha: "0.5";
    }
  </mx:Style>
  <mx:Label text="Hello AIR" horizontalCenter="0" verticalCenter="0"/>
</mx:WindowedApplication>
```

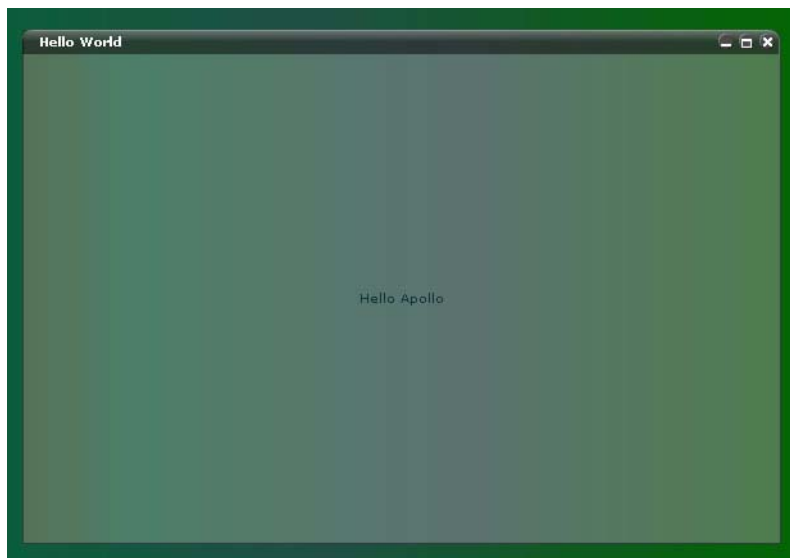
## CLOSE PROCEDURE

# Test the application

1 Click the Debug button  in the main Flex Builder toolbar.

You can also select the Run | Debug | AIRHelloWorld command.

The resulting AIR application should look something like the following example (the green background is the user's desktop):



2 Using the horizontalCenter and verticalCenter properties of the Label control, the text is placed in the center of the window. Move or resize the window as you would any other desktop application.

***Note:** If the application does not compile, fix any syntax or spelling errors that you may have inadvertently entered into the code. Errors and warnings are displayed in the Problems view in Flex Builder.*

#### **CLOSE PROCEDURE**

## **Package and run your application**

You are now ready use Flex Builder to package the "Hello World" application into an AIR file for distribution. An AIR file is an archive file that contains the application files (all of the files contained in the project's bin folder). You distribute the AIR package to users who then use it to install the application.

- 1** Ensure that your application has no compile errors and runs as expected.
- 2** Select **File | Export**.
- 3** Select the AIR Package option from the AIR folder, and then click **Next**.
- 4** By default, the project name and application MXML file are selected, as are the files to be included in the package. Also by default, the package is created in the project's folder in your Flex Builder workspace. You can change this. Click **Finish** to create the AIR package.

You can now run the application from the desktop as your users would by double-clicking the AIR file.

#### **CLOSE PROCEDURE**

# Chapter 6: Creating your first AIR application with the Flex SDK

For a quick, hands-on illustration of how AIR works, use these instructions to create a simple SWF-based AIR "Hello World" application using the Flex SDK. You will learn how to compile, test, and package an AIR application with the command-line tools provided with the SDK.

To begin, you must have installed the runtime and set up your development environment.

This chapter contains the following sections:

- [Create the application XML file](#)
- [Write the application code](#)
- [Compile the application](#)
- [Test the application](#)
- [Package the application](#)

## See also

"Installing the runtime and sample applications" on page 6

"Setting up for Flex SDK" on page 14

"Creating an AIR application using the command line tools" on page 30

## Create the application XML file

Each AIR application requires an application descriptor file. This XML file defines various properties of the application, and is embedded in the AIR package that is distributed to users.

❖ To create the application descriptor file, use a text editor to create an XML file named `AIRHelloWorld-app.xml` and then add the following text:

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

<application xmlns="http://ns.adobe.com/air/application/1.0.M4"
    appId="com.adobe.air.example.AIRHelloWorld" version="1.0">
    <name>Hello World</name>
    <description>A test AIR application.</description>
    <copyright>2007</copyright>

    <rootContent systemChrome="none" transparent="true"
        visible="true">AIRHelloWorld.swf</rootContent>
</application>
```

## Write the application code

**Note:** SWF-based AIR applications can use a main class defined either with MXML or with ActionScript. This example uses an MXML file to define its main class. The process for creating an AIR application with a main ActionScript class is similar. Instead of compiling an MXML file into the SWF, you compile the ActionScript class file. The main ActionScript class must extend the `flash.display.Sprite` class.

Like all Flex applications, AIR applications built with the Flex framework contain a main MXML file. AIR applications, however, can use the `WindowedApplication` component as the root element instead of the `Application` component. The `WindowedApplication` component provides the window and basic window control chrome needed to run your application on the desktop. The following procedure creates the Hello World application.

- 1 Using a text editor, create a file named `AIRHelloWorld.mxml` and add the following MXML code:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute" title="Hello World">
</mx:WindowedApplication>
```

- 2 Next, add a `Label` component to the application (place it inside the `WindowedApplication` tag) and set the `text` property of the `Label` component to `"Hello AIR"` and set the layout constraints to always keep it centered, as shown here:

```
<?xml version="1.0" encoding="utf-8"?><?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
    title="Hello World">
    <mx:Label text="Hello AIR" horizontalCenter="0" verticalCenter="0"/>
</mx:WindowedApplication>
```

- 3 Add the following style block:

```
<mx:Style>
    Application
    {
        background-image:"";
        background-color:"";
        background-alpha:"0.5";
    }
</mx:Style>
```

These style settings apply to the entire application and render the window background a slightly transparent gray.

The entire application code now looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
    title="Hello World">
    <mx:Style>
```

```
        Application
        {
            background-image:"";
            background-color:"";
            background-alpha:"0.5";
        }
    </mx:Style>
    <mx:Label text="Hello AIR" horizontalCenter="0" verticalCenter="0"/>
</mx:WindowedApplication>
```

## Compile the application

Before you can run and debug the application, you must compile the MXML into a SWF file. (Make sure you have already added the AIR command-line tools to your class path.)

**1** Open a command prompt in Windows or a terminal in the Mac OS and navigate to the source location of your AIR application.

**2** Enter the following command:

```
amxmlc AIRHelloWorld.mxml
```

**Note:** If the application does not compile, fix syntax or spelling errors. Errors and warnings are displayed in the console window used to run the amxmlc compiler.

### See also

“Compiling an Apollo application with the amxmlc compiler” on page 4

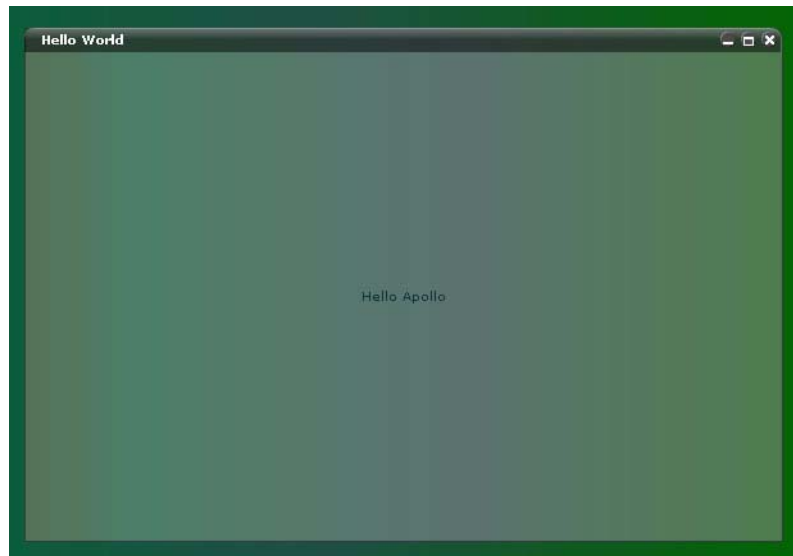
## Test the application

To run and test the application from the command line, use the AIR Debug Launcher (ADL) to launch the application using its application descriptor file.

❖ From the command prompt, enter the following command:

```
adl AIRHelloWorld-app.xml
```

The resulting AIR application looks something like this (the green background is the user's desktop):



Using the `horizontalCenter` and `verticalCenter` properties of the `Label` control, the text is placed in the center of the window. Move or resize the window as you would any other desktop application.

**See also**

“Debugging using the Apollo Debug Launcher” on page 7

## Package the application

You are now ready to package the “Hello World” application into an AIR file for distribution or testing from the desktop. An AIR package is an archive file that contains the application files. You distribute the AIR package to users who then use it to install the application.

- 1 Ensure that your application has no compile errors and runs as expected.
- 2 From the command prompt, enter the following command:

```
adt -package AIRHelloWorld.air AIRHelloWorld-app.xml AIRHelloWorld.swf
```

The first argument after the `-package` command is the name of the AIR file that ADT produces. The second argument is the name of the application descriptor file. The subsequent arguments are the files for the modules, scripts, and assets used by your application. This example only uses a single file, but you can include any number of files and directories.

After the AIR package is created, you can double-click, Ctrl-Click, or type the AIR filename as a command in the shell to install and run the application.

**See also**

“Packaging an Apollo application using the Apollo Developer Tool” on page 9



# Chapter 7: Developing AIR applications with Flex Builder

Adobe Flex Builder 3 provides you with the tools you need to create Adobe Integrated Runtime (AIR) projects, work with the Flex AIR components, and debug and package your Adobe AIR applications.

This chapter contains the following sections:

[“Create AIR projects with Flex Builder” on page 25](#)

[“Debug AIR applications with Flex Builder ” on page 25](#)

[“Package AIR applications with Flex Builder” on page 26](#)

[“Create an AIR Library project ” on page 26](#)

## See also

[“Setting up for Flex Builder” on page 12](#)

## Create AIR projects with Flex Builder

If you have not already done so, install AIR and Flex Builder 3.

- 1 Open Flex Builder 3.
- 2 Select File > New > AIR Project (if you are using the Eclipse plug-in configuration, select File > New > Other, expand the Flex node, and select AIR Project).
- 3 Leave the Basic option selected, and then click Next.
- 4 Type the Project Name, modify the default location (optional), and then click Next.
- 5 To use ActionScript to define the main class of the application, change the setting for the main application file to use the .as file extension. (The default file name uses the Flex MXML .mxml file extension.)
- 6 Click Next again.
- 7 Specify settings for the AIR application, and then click Finish.

## CLOSE PROCEDURE


## See also

[“Setting up for Flex Builder” on page 12](#)

[“The application descriptor file structure” on page 42](#)

## Debug AIR applications with Flex Builder

Flex Builder provides full debugging support for AIR applications. For more information about the debugging capabilities of Flex Builder, refer to Flex Builder Help.

- 1 Open a source file for the application (such as an MXML file) in Flex Builder.
- 2 Click the Debug button on the main toolbar  .

You can also select Run > Debug.

The application launches and runs in the ADL application (the AIR Debugger Launcher). The Flex Builder debugger catches any breakpoints or runtime errors and you can debug the application like any other Flex application.

You can also debug an application from the command line, using the AIR Debug Launcher command-line tool.

#### **CLOSE PROCEDURE**

#### **See also**

“Debugging using the AIR Debug Launcher” on page 33

### **Package AIR applications with Flex Builder**

When your application is complete and ready to be distributed (or tested running from the desktop), you package it into an AIR file.

- 1 Open the project and ensure that the application has no compile errors and runs as expected.
- 2 Select File > Export.
- 3 Select Deployable AIR File from the AIR folder, and then click Next.
- 4 Select any files (such as loaded media or SWF files) to include in the AIR file. The application.xml file and the main SWF file for the project are included by default.
- 5 Specify the destination for the AIR file, including the filename, and then click Finish.

You can also package an AIR application using the ADT command-line tool.

#### **CLOSE PROCEDURE**

#### **See also**

“Packaging an AIR application using the AIR Developer Tool” on page 34

### **Create an AIR Library project**

The beta release of Flex Builder 3 does not contain a new project wizard to generate an AIR Library project. Therefore, to create a custom AIR component in Flex Builder, you need to create a standard Library project and then manually edit it to create a library that references AIR classes.

- 1 Select File > New > Flex Library Project.
- 2 Specify a project name and click Next.
- 3 Click the Library Path tab.
- 4 Remove \${FRAMEWORKS}/libs/playerglobal.swc.
- 5 Add airglobal.swc and set the link type to External.
- 6 Click Finish.

#### **CLOSE PROCEDURE**

# Chapter 8: Using the Flex AIR components

You use the Flex AIR components in an Adobe® Integrated Runtime (AIR™) application to display HTML web pages, and file information for the system on which the Adobe AIR application runs. You can display file system information in a list, tree, or data-grid format. For example, you can use the `FileSystemDataGrid` control to display file information in a data-grid format that includes the file name, creation date, modification date, type, and size.

This chapter contains the following sections:

[About the Flex AIR components](#)

[Using the WindowedApplication component](#)

## About the Flex AIR components

Flex provides the following AIR components:

**WindowedApplication container** Defines the application container that you use to create Flex applications for AIR. The `WindowedApplication` container adds window-related functionality to the `Flex Application` container when you build AIR applications.

**HTML control** Displays HTML web pages in your application.

**FileSystemComboBox control** Defines a combo box control for selecting a location in a file system.

**FileSystemDataGrid control** Displays file information in a data-grid format. The file information displayed can include the file name, creation date, modification date, type, and size.

**FileSystemHistoryButton control** Use with a `FileSystemList` or `FileSystemDataGrid` control to let the user move backwards or forwards through the navigation history of the control.

**FileSystemList control** Displays the contents of a file system directory.

**FileSystemTree control** Displays the contents of a file system directory as a tree.

For more information on these components, see the *Flex 3 Language Reference*.

### Example: Displaying a directory structure with Flex AIR

The following example uses the `WindowedApplication` container and the `FileSystemTree` and `FileSystemDataGrid` controls. In this example, the `FileSystemTree` control displays a directory structure. Clicking a directory name in the `FileSystemTree` control causes the `FileSystemDataGrid` control to display information about the files in the selected directory:

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

<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml">
```

```

<mx:HDividedBox>
  <mx:FileSystemTree id="tree"
    width="200" height="100%"
    directory="{new File('C:\\')} "
    enumerationMode="directoriesOnly"
    change="dataGrid.directory = File(tree.selectedItem);"/>
  <mx:FileSystemDataGrid id="dataGrid"
    width="100%" height="100%"
    directory="{new File('C:\\')}"/>
</mx:HDividedBox>
</mx:WindowedApplication>

```

**See also**

## Using the WindowedApplication component

The `mx:WindowedApplication` container component defines an AIR application object that includes its own window controls. In an MXML AIR application, a `<WindowedApplication>` tag replaces the `<Application>` tag.

A `WindowedApplication` component provides the following controls:

- A title bar
- A minimize button
- A maximize button
- A close button

The window that the `WindowedApplication` component defines conforms to the standard behavior of the operating system. The user can move the window by dragging the title bar and resize the window by dragging on any side or corner of the window.

By default, the `WindowedApplication` component creates an application window for which `windowMode` is set to `systemChrome`, and `visibility` is set to `true`. These settings are made in the `application.xml` file that Flex Builder generates for an AIR application. To eliminate the system chrome and window controls that the `WindowedApplication` component creates by default, edit the opening and closing `WindowedApplication` tags in the MXML file so that they are `Application` tags and, in the `application.xml` file, set `systemChrome` to `none`.

The following simple application shows how the `WindowedApplication` component works:

```

<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication
  xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute"
  <mx:Text text="Hello World" />

```

```
</mx:WindowedApplication>
```

For more information about the WindowedApplication container and other Flex AIR components, see the *Flex 3 Language Reference*.

# Chapter 9: Creating an AIR application using the command line tools

The Adobe Integrated Runtime (AIR) command-line tools provide an alternative to Flex Builder for compiling, debugging, and packaging Adobe AIR applications. You can also use these tools in automated build processes. The command-line tools are included in both the Flex and AIR SDKs.

This chapter contains the following sections:

- [Compiling an AIR application with the amxmlc compiler](#)
- [Compiling an AIR component or library with the acompc compiler](#)
- [Debugging using the AIR Debug Launcher](#)
- [Packaging an AIR application using the AIR Developer Tool](#)
- [Using Apache Ant with the AIR SDK tools](#)

## Compiling an AIR application with the amxmlc compiler

Compile the ActionScript and MXML assets of your AIR application with the command-line MXML compiler (amxmlc):

```
amxmlc [compiler options] -- MyAIRApp.xml
```

where *[compiler options]* specifies the command-line options used to compile your AIR application.

The amxmlc command invokes mxmmlc with an additional parameter, `+configname=air`, which instructs the compiler to use the `air-config.xml` file instead of the `flex_config.xml` file. Using amxmlc is otherwise identical to using mxmmlc. The mxmmlc compiler and the configuration file format are described in [Building and Deploying Flex 2 Applications](#) in the Flex 2 documentation library.

The compiler loads the `air-config.xml` configuration file specifying the AIR and Flex libraries typically required to compile an AIR application. You can also use a local, project-level configuration file to override or add additional options to the global configuration. Typically the easiest way to create a local configuration file is to edit a copy of the global version. You can load the local file with the `-load-config` option:

`-load-config=project-config.xml` Overrides global options.

`-load-config+=project-config.xml` Adds additional values to those global options that take more than value, such as the `-library-path` option. Global options that only take a single value are overridden.

You can also use a special naming convention for the file and the amxmlc compiler will load the local configuration file automatically. For example, if your application's main MXML file is `RunningMan.xml`, then name the configuration file: `RunningMan-config.xml`. To compile the application you only need to type:

```
amxmlc RunningMan.xml
```

**Note:** The compiler commands use relative paths to find configuration files and executables. See “Setting up for Flex SDK” on page 14

## Examples

The following examples demonstrate use of the amxmlc compiler. (Only the ActionScript and MXML assets of your application need to be compiled.)

Compile an AIR MXML file:

```
amxmlc myApp.mxml
```

Compile and set the output name:

```
amxmlc -output anApp.swf -- myApp.mxml
```

Compile an AIR ActionScript file:

```
amxmlc myApp.as
```

Specify a compiler configuration file:

```
amxmlc -load-config config.xml -- myApp.mxml
```

Add additional options from another configuration file:

```
amxmlc -load-config+=moreConfig.xml -- myApp.mxml
```

Add an additional libraries on the command line (to those already in the configuration file):

```
amxmlc -library-path+=/libs/libOne.swc,/libs/libTwo.swc -- myApp.mxml
```

Compile an AIR MXML file without using a configuration file (Win):

```
mxmmlc -library-path [AIR SDK]/frameworks/libs/air/airframework.swc, ^  
[AIR SDK]/frameworks/libs/air/airframework.swc, ^  
-library-path [Flex 2 SDK]/frameworks/libs/framework.swc ^  
-- myApp.mxml
```

Compile an AIR MXML file without using a configuration file (Mac OS X):

```
mxmmlc -library-path [AIR SDK]/frameworks/libs/air/airframework.swc, \  
[AIR SDK]/frameworks/libs/air/airframework.swc, \  
-library-path [Flex 2 SDK]/frameworks/libs/framework.swc \  
-- myApp.mxml
```

Compile an AIR MXML file to use a runtime-shared library

```
amxmlc -external-library-path+=../lib/myLib.swc -runtime-shared-libraries=myrsl.swf --  
myApp.mxml
```

Compiling from Java (with the class path set):

```
java flex2.tools.Compiler +flexlib [Flex SDK 2]/frameworks +configname=air [additional  
compiler options] -- myApp.mxml
```

The flexlib option identifies the location of your Flex SDK frameworks directory, enabling the compiler to locate the flex\_config.xml file.

Compiling from Java (without the class path set):

```
java -jar [Flex SDK 2]/lib/mxmlc.jar +flexlib [Flex SDK 2]/frameworks +configname=air  
[additional compiler options] -- myApp.mxml
```

## Compiling an AIR component or library with the `acompc` compiler

Use the component compiler, `acompc`, to compile AIR libraries and independent components. The component compiler behaves like the `amxmlc` compiler, with the following exceptions:

- You must specify which classes within the code base to include in the library or component.
- `acompc` does not look for a local configuration file automatically. To use a project configuration file, you must use the `-load-config` option.

`acompc` is identical to `compc`, except that it loads configuration options from the `air-config.xml` file instead of the `flex_config.xml` file.

### Component Compiler Configuration File

You can use a compiler configuration file with the `acompc` compiler by specifying the path to a local config file with the `-load-config` option. By adding the path to the source files and specify the classes to include within the configuration file you avoid having to type them on the command line.

The following example illustrates a configuration for building a library with two classes, `ParticleManager` and `Particle`, both in the ActionScript package, `com.adobe.samples.particles`, and located in the `source/com/adobe/samples/particles` folder (relative to the working directory when the compiler is run).

```
<flex-config>
  <compiler>
    <source-path>
      <path-element>source</path-element>
    </source-path>
  </compiler>
  <include-classes>
    <class>com.adobe.samples.particles.ParticleManager</class>
    <class>com.adobe.samples.particles.Particle</class>
  </include-classes>
</flex-config>
```

To enter the same command entirely on the command line, type:

```
compc -source-path source -include-classes com.adobe.samples.particles.Particle
com.adobe.samples.particles.ParticleManager
```

(Type the entire command on one line, or use the line continuation character for your command shell.)

### Examples

These examples assume you are using a configuration file.

Compile an AIR component or library:

```
acompc -load-config myLib-config.xml -output lib/myLib.swc
```



Compile a runtime-shared library:

```
acompc -load-config myLib-config.xml -directory -output lib
```

(Note, the folder lib must exist and be empty before running the command.)

Reference a runtime-shared library:

```
acompc -load-config myLib-config.xml -output lib/myLib.swc
```

## Debugging using the AIR Debug Launcher

Use the AIR Debug Launcher (ADL) to run both Flex-based and HTML-based applications during development. Using ADL, you can run an application without first packaging and installing it. The runtime does not need to be installed to use ADL (just the AIR SDK).

The debugging support provided by ADL is limited to the printing of trace statements. If you're developing a Flex-based application, use the Flash Debugger (or Flex Builder) for complex debugging issues in SWF-based applications.

### Launch an application with ADL

❖ Use the following syntax:

```
adl [-runtime <runtime-directory>] <application.xml> [<root-directory>] [-- arguments]
```

**-runtime <runtime-directory>** Specify the directory containing the runtime to use. If not specified, the runtime directory in the same SDK as the ADL program will be used. (If ADL is moved out of its SDK folder, then the runtime directory must be specified.)

**<application.xml>** The application descriptor file.

**<root-directory>** The root directory of the application to run. If not specified, the directory containing the application descriptor file is used.

**-- arguments** Any character strings appearing after "--" are passed to the application as command-line arguments.

**Note:** When you launch an AIR application that is already running, a new instance of that application is not started. Instead, an invoke event is dispatched to the running instance. In this case, the ADL root-directory and runtime parameters are not significant.

### Print trace statements

❖ To print trace statements to the console used to run ADL, use the `trace()` function:

```
trace("debug message");
```

### ADL Examples

Run an application in the current directory:

```
adl myApp-app.xml
```

Run an application in a subdirectory of the current directory:

```
adl source/myApp-app.xml release
```

Run an application and pass in two command-line arguments, "foo" and "bar":

```
adl myApp-app.xml -- foo bar
```

Run an application using a specific runtime:

```
adl -runtime /AIR/XYZ/AIRSDK/bin myApp-app.xml
```

### Set breakpoints with the Flash Debugger

To debug a SWF-based AIR application with the Flash Debugger, you must start an FDB session and then launch a debug version of your application. The debug version of a SWF file will automatically connect to a "listening" FDB session.

- 1 Start FDB. The FDB program can be found in the `bin` directory of your Flex 2 SDK folder.

The console displays the FDB prompt: `<fdb>`

- 2 Execute the Run command: `<fdb>run [Enter]`

- 3 In a different command or shell console, start a debug version of your application:

```
adl myApp-debug.xml
```

- 4 Using the FDB commands, set breakpoints as desired.

- 5 Type: `continue [Enter]`

- 6 Set any additional breakpoints.

- 7 Type: `continue [Enter]`

## Packaging an AIR application using the AIR Developer Tool

Package your application as an AIR file for distribution with the AIR Developer Tool (ADT). ADT creates installation packages for both HTML-based and SWF-based AIR applications. (If you are using Flex Builder to create your application, you can use the Flex Builder Export wizard to build the AIR file package.)

ADT is a Java program that you can run from the command line or a build tool such as Ant. The SDK includes command-line scripts that execute the Java program for you. See [“Setting up for Flex SDK” on page 14](#) for information on configuring your system to run the ADT tool.

Every AIR application must, at a minimum, have an application descriptor file and a main SWF or HTML file. Any other application assets to be installed with the application must be packaged in the AIR file as well.

**Note:** The settings in the application descriptor file determine the identity of an AIR application and its default installation path. See “The application descriptor file structure” on page 42 for more information.

### Use ADT

- ❖ Use the following syntax:

```
adt -package air_file app_xml [ file_or_dir | -C dir file_or_dir ... ] ...
```

**air\_file** The name of the AIR file to be created.

**app\_xml** The path to the application descriptor file. No matter what name is assigned to the application descriptor file, it will be renamed to “application.xml” in the package. The path can be specified relative to the current directory or as an absolute path.

**file\_or\_dir** The files and directories to package in the AIR file. Any number of files and directories can be specified, delimited by whitespace. If you list a directory, all files and subdirectories within, except hidden files, are added to the package. (In addition, if the application descriptor file is specified, either directly, or through wildcard or directory expansion, it is ignored and not added to the package a second time.) The files and directories specified must be in the current directory or one of its subdirectories. Use the `-C` option to change the current directory.

**Important:** Wildcards cannot be used in the `file_or_dir` arguments following the `-C` option. (The command shell expands wildcards before passing the arguments to ADT, which results in ADT looking for files in the original working directory instead of the directory specified by the `-C` option.)

**-C dir** Changes the working directory to `dir` before processing subsequent files and directories added to the application package. The files or directories are added to the root of the application package. The `-C` option can be used any number of times to include files from multiple points in the file system. If a relative path is specified for `dir`, the path is always resolved from the original working directory.

As ADT process the files and directories to be included in the package, the relative paths between the current directory and the target files are stored. These paths are expanded into the application directory structure when the package is installed. Thus specifying `"-C release/bin lib/feature.swf"` would place the file `"release/bin/lib/feature.swf"` in the `"lib"` subdirectory of the root application folder.

**Note:** You must make sure that the `rootContent` element of the application descriptor file specifies the final path to the main application file relative to the root application directory. This may be an issue if the main application file is not located in the current directory when you run ADT.

## ADT Examples

Package specific application files in the current directory:

```
adt -package myApp.air myApp.xml myApp.swf components.swc
```

Package all files and subdirectories in the current working directory:

```
adt -package myApp.air myApp.xml .
```

Package only the main files and an images subdirectory:

```
adt -package myApp.air myApp.xml myApp.swf images
```

Package the application.xml file and main SWF located in a working directory (release\bin):

```
adt -package myApp.air release\bin\myApp.xml -C release\bin myApp.swf
```

The following example shows how to package assets from more than one place in your build file system. Suppose the application assets are located in the following folders prior to packaging:

```
/devRoot
  /myApp
    /release
      /bin
        myApp.xml
        myApp.swf
  /artwork
    /myApp
```

```

        /images
            image-1.png
            ...
            image-n.png
    /libraries
        /release
            /libs
                lib-1.swf
                ...
                lib-n.swf

```

The following ADT command is run from the /devRoot/myApp directory:

```
adt -package myApp.air release/bin/myApp.xml -C release/bin myApp.swf
    -C ../artwork/myApp images -C ../audio
```

That command results in this package structure:

```

/myAppRoot
    /META-INF
        /AIR
            application.xml
            hash
myApp.swf
mimetype
/images
    image-1.png
    ...
    image-n.png
/lib
    lib-1.swf
    ...
    lib-n.swfAIRAlias.js

```

Run ADT as a Java program (without setting the classpath):

```
java -jar {AIRSDK}\lib\ADT.jar -package myApp.air myApp.xml myApp.swf
```

Run ADT as a Java program (with the Java classpath set to include the ADT.jar package):

```
java com.adobe.air.ADT -package -package myApp.air myApp.xml myApp.swf
```

## Using Apache Ant with the AIR SDK tools

This section provides examples of using the Apache Ant build tool to compile, test, and package AIR applications.

**Note:** this discussion does not attempt to provide a comprehensive outline of Apache Ant. For Ant documentation, see <http://Ant.Apache.org>.

### Using Ant for simple projects

This example illustrates building an AIR application using Ant and the AIR command-line tools. A very simple project structure is used with all files stored in a single directory.

**Note:** this example assumes that you are using the separate AIR SDK rather than Flex Builder. The tools and configuration files included with Flex Builder are stored in a different directory structure.

To make it easier to reuse the build script, these examples use several defined properties. One set of properties identifies the installed locations of the command-line tools:

```
<property name="SDK_HOME" value="C:/FlexSDK"/>
<property name="MXMLC.JAR" value="${SDK_HOME}/lib/mxmlc.jar"/>
<property name="ADL" value="${SDK_HOME}/bin/adl.exe"/>
<property name="ADT.JAR" value="${SDK_HOME}/lib/adt.jar"/>
```

The second set of properties are project specific. These properties assume the naming convention that application descriptor and AIR files are named based on the root source file, but other conventions are easily supported. The properties also define the MXMLC debug parameter as true (by default).

```
<property name="APP_NAME" value="ExampleApplication"/>
<property name="APP_ROOT" value="."/>
<property name="MAIN_SOURCE_FILE" value="${APP_ROOT}/${APP_NAME}.mxml"/>
<property name="APP_DESCRIPTOR" value="${APP_ROOT}/${APP_NAME}-app.xml"/>
<property name="AIR_NAME" value="${APP_NAME}.air"/>
<property name="DEBUG" value="true"/>
```

### Invoking the compiler

To invoke the compiler, the example uses a Java task to run mxmlc.jar:

```
<target name="compile">
  <java jar="${MXMLC.JAR}" fork="true" failonerror="true">
    <arg value="-debug=${DEBUG}"/>
    <arg value="+flexlib=${SDK_HOME}/frameworks"/>
    <arg value="+configname=air"/>
    <arg value="-file-specs=${MAIN_SOURCE_FILE}"/>
  </java>
</target>
```

When invoking mxmlc using Java, you must specify the +flexlib parameter. The +configname=air parameter instructs mxmlc to load the supplied AIR configuration file along with the normal Flex config file.

### Invoking ADL to test an application

To run the application with ADL, use an exec task:

```

<target name="test" depends="compile">
    <exec executable="${ADL}">
        <arg value="${APP_DESCRIPTOR}" />
    </exec>
</target>

```

### Invoking ADT to package an application

To package the application use a Java task to run the adt.jar tool:

```

<target name="package" depends="compile">
    <java jar="${ADT.JAR}" fork="true" failonerror="true">
        <arg value="-package" />
        <arg value="${AIR_NAME}" />
        <arg value="${APP_DESCRIPTOR}" />
        <arg value="${APP_NAME}.swf" />
        <arg value="*.png" />
    </java>
</target>

```

If your application has more files to package, you could add additional <arg> elements.

## Using Ant for complex projects

Because few applications would keep all files in a single directory, the following example illustrates a build file used to compile, test, and package an AIR application which has a more practical project directory structure.

This sample project stores the application source files and other assets like icon files within a src directory. The build script creates the following working directories:

**build** Stores the release (non-debug) versions of compiled SWF files.

**debug** Stores an unpackaged debug version of the application, including any compiled SWFs and asset files. The ADL utility runs the application from this directory.

**release** Stores the final AIR package

The AIR tools require the use of some additional options when operating on files outside the current working directory:

**Compiling** The -output option of the mxmcl compiler allows you to specify where to put the compiled file, in this case, in the build or debug subdirectories.

**Testing** The second argument passed to adl specifies the root directory of the AIR application.

**Packaging** Packaging files from subdirectories that should not be part of the final package structure (such as the src directory) requires using the -C directive to change the adt working directory. When you use the -C directive, files in the new working directory are packaged at the root level of the air file. Thus "-C build file.png" places file.png at the root, and "-C assets icons" places the icon folder at the root level, and packages all the files within the icons folder as well.

```
<?xml version="1.0" ?>

<project>

  <!-- SDK properties -->

  <property name="SDK_HOME" value="C:/FlexSDK"/>

  <property name="MXMLC.JAR" value="${SDK_HOME}/lib/mxmlc.jar"/>

  <property name="ADL" value="${SDK_HOME}/bin/adl.exe"/>

  <property name="ADT.JAR" value="${SDK_HOME}/lib/adt.jar"/>


  <!-- Project properties -->

  <property name="APP_NAME" value="ExampleApplication"/>

  <property name="APP_ROOT_DIR" value="."/>

  <property name="MAIN_SOURCE_FILE" value="${APP_ROOT_DIR}/src/${APP_NAME}.mxm1"/>

  <property name="APP_ROOT_FILE" value="${APP_NAME}.swf"/>

  <property name="APP_DESCRIPTOR" value="${APP_ROOT_DIR}/${APP_NAME}-app.xml"/>

  <property name="AIR_NAME" value="${APP_NAME}.air"/>

  <property name="build" location="${APP_ROOT}/build"/>

  <property name="debug" location="${APP_ROOT_DIR}/debug"/>

  <property name="release" location="${APP_ROOT_DIR}/release"/>

  <property name="assets" location="${APP_ROOT_DIR}/src/assets"/>


  <target name="init" depends="clean">

    <tstamp/>

    <mkdir dir="${build}"/>

    <mkdir dir="${debug}"/>

    <mkdir dir="${release}"/>

  </target>


  <target name="debugcompile" depends="init">

    <java jar="${MXMLC.JAR}" fork="true" failonerror="true">

      <arg value="-debug=true"/>

      <arg value="+flexlib=${SDK_HOME}/frameworks"/>

      <arg value="+configname=air"/>

    </java>

  </target>

</project>
```

```
<arg value="-file-specs=${MAIN_SOURCE}"/>
<arg value="-output=${debug}/${APP_ROOT_FILE}"/>
</java>
<copy todir="${debug}">
    <fileset dir="${assets}"/>
</copy>
</target>

<target name="releasecompile" depends="init">
    <java jar="${MXMLC.JAR}" fork="true" failonerror="true">
        <arg value="-debug=false"/>
        <arg value="+flexlib=${SDK_HOME}/frameworks"/>
        <arg value="+configname=air"/>
        <arg value="-file-specs=${MAIN_SOURCE_FILE}"/>
        <arg value="-output=${build}/${APP_ROOT_FILE}"/>
    </java>
</target>

<target name="test" depends="debugcompile">
    <exec executable="${ADL}">
        <arg value="${APP_DESCRIPTOR}"/>
        <arg value="${debug}"/>
    </exec>
</target>

<target name="package" depends="releasecompile">
    <java jar="${ADT.JAR}" fork="true" failonerror="true">
        <arg value="-package"/>
        <arg value="${release}/${AIR_NAME}"/>
        <arg value="${APP_DESCRIPTOR}"/>
        <arg value="-C"/>
        <arg value="${build}"/>
        <arg value="${APP_ROOT_FILE}"/>
        <arg value="-C"/>
        <arg value="${assets}"/>
        <arg value="icons"/>
    </java>
</target>
```



```
        </java>
    </target>

    <target name="clean" description="clean up">
        <delete dir="${build}"/>
        <delete dir="${debug}"/>
        <delete dir="${release}"/>
    </target>
</project>
```

# Chapter 10: Setting application properties

Aside from all the files and other assets that make up your Adobe Integrated Runtime (AIR) applications, an AIR application requires an application descriptor, an XML file which defines the basic properties of the application.

When you use Flex Builder 3, the application descriptor file is automatically generated when you create a new AIR project. If you're developing AIR applications using the Flex and AIR SDKs, you need to create this file manually.

This chapter contains the following sections:

[The application descriptor file structure](#)

[Defining properties in the application descriptor file](#)

## The application descriptor file structure

The application descriptor file, `application.xml`, contains the properties of the entire application, such as its name, the version, copyright, and so on. Any file name can be used for the application descriptor file. Flex Builder automatically creates a descriptor file when you create a new project. When you package the application, with either Flex Builder or ADT, the application descriptor file is renamed `application.xml` and placed within a special directory inside the package.

Here's the content of a sample application descriptor file:

```
<?xml version="1.0" encoding="utf-8" ?>
<application appId="com.adobe.air.examples.HelloWorld" version="2.0"
  xmlns="http://ns.adobe.com/air/application/1.0.M4">
  <name>AIR Hello World</name>
  <description>
    This is the Hello World sample file from the Adobe AIR documentation.
  </description>
  <title>HelloWorld -- AIR Example</title>
  <copyright>Copyright © 2006</copyright>
  <rootContent systemChrome="none"
    transparent="true"
    visible="true"
    width="640"
    height="480">
    HelloWorld-debug.swf
  </rootContent>
  <installFolder>Adobe/Examples</installFolder>
  <icon>
    <image16x16>icons/smallIcon.png</image16x16>
    <image32x32>icons/mediumIcon.jpg</image32x32>
    <image48x48>icons/bigIcon.gif</image48x48>
    <image128x128>icons/biggestIcon.png</image128x128>
  </icon>
  <handleUpdates/>
  <fileTypes>
    <fileType>
      <name>adobe.VideoFile</name>
      <extension>avf</extension>
      <description>Adobe Video File</description>
      <contentType>application/vnd.adobe.video-file</contentType>
    </fileType>
  </fileTypes>
</application>
```

## Defining properties in the application descriptor file

At its root, the application descriptor file contains an `application` property that has several attributes:

```
<application appId="com.adobe.air.HelloWorld" version="1.0"
  xmlns="http://ns.adobe.com/air/application/1.0.M4">
```

**appId** A unique application identifier. The attribute value is restricted to the following characters:

- 0–9
- a–z
- A–Z
- . (dot)
- - (hyphen)

The value must contain 17 to 255 characters.

The `appId` string typically uses a dot-separated hierarchy, in alignment with a reversed DNS domain address, a Java package or class name, or an OS X Universal Type Identifier. The DNS-like form is not enforced, and AIR does not create any association between the name and actual DNS domains.

**version** Specifies the version information for the application. (It has no relation to the version of the runtime). The version string is an application-defined designator. AIR does not interpret the version string in any way. Thus version "3.0" is not assumed to be more current than version "2.0." Examples: "1.0", ".4", "0.5", "4.9", "1.3.4a".

**xmlns** The AIR namespace, which you must define as the default XML namespace. The namespace will change with each release of AIR.

## Defining the application's name, title, description, copyright, and installation folder

**name** The name of the application. You must define this element.

```
<name>TestApp</name>
```

In Windows, this is displayed in the application's title bar and in the Windows Start menu. On Mac OS, it is displayed in the menu bar when the application is running.

**title** (Optional) Displayed in the AIR application installer.

```
<title>TestApp from Adobe Systems Inc.</title>
```

**description** (Optional) Displayed in the AIR application installer.

```
<description>An MP3 player.</description>
```

**copyright** (Optional) The copyright information for the AIR application.

```
<copyright>Copyright © 2006 [YourCompany, Inc.]</copyright>
```

**installFolder** (Optional) Identifies the subdirectory of the default installation directory.

```
<installFolder>Acme</installFolder>
```

On Windows, the default installation subdirectory is the Program Files directory. On Mac OS, it is the `./Applications` directory. For example, if the `installFolder` property is set to "Acme" and an application is named "ExampleApp", then the application will be installed in `C:\Program Files\Acme\Example` on Windows and in `./Applications/Acme/Example.app` on MacOS.

Use the forward-slash (/) character as the directory separator character if you want to specify a nested subdirectory, as in the following:

```
<installFolder>Acme/Power Tools</installFolder>
```

The `installFolder` property can contain any Unicode (UTF-8) character except the following, which are prohibited from use as folder names various file systems.

Character	Hex Code
various	0x00 – x1F
*	x2A
"	x22

Character	Hex Code
:	x3A
>	x3C
<	x3E
?	x3F
\	x5C
	x7C

The `installFolder` property is optional. If you specify no `installFolder` property, the application is installed in a subdirectory of the default installation directory, based on the `name` property.

### Defining the `rootContent` element

The `application.xml` file also indicates the `rootContent` file. This is the file that is first loaded by the application. This may be either a SWF file or an HTML file.

The value specified for the `rootContent` property is a URL that is relative to the root of the `application.xml` file. For example, in the following `rootContent` property, the `AIRTunes.swf` file (in the same directory as the `application.xml` file) is defined as the root file for the application:

```
<rootContent
  systemChrome="none"
  transparent="true"
  visible="true"
  height="400"
  width="600">
  AIRTunes.swf
</rootContent>
```

The attributes of the `rootContent` element set the properties of the window into which the root content file will be loaded.

**systemChrome** If you set this attribute to `standard`, system chrome is displayed, and the application has no transparency. If you set it to `none`, no system chrome is displayed. When using the `Flex WindowedApplication` component, the component applies its custom chrome.

**transparent** Set this to `"true"` if you want the application window to support alpha blending. The transparent property of a window cannot be changed after the window has been created. A window with transparency may draw more slowly and require more memory than otherwise.

**Important:** You can only set `transparent="true"` when `systemChrome="none"`.

**visible** Set this to `"false"` if you want to have the main window be hidden when it is first created. The default value is `"true"`.

You may want to have the main window hidden initially, and then set its position and size in your application code. You can then display the window by setting the `stage.window.visible` property (for the main window) to `true`. For details, see [“Working with windows” on page 48](#).

**height, width** The height and width of the main window of the application. If you do not set these attributes, the window size is determined by the settings in the root SWF file or, in the case of HTML, by the operating system. If you set the `visible` attribute to `"false"` you can add code to adjust the `width` and `height` property of the window (before setting its `visible` property to `true`).

### Specifying icon files

The `icon` property specifies one or more icon files to be used for the application. Including an icon is optional. If you do not specify an `icon` property, the operating system will display a default icon.

The path specified is relative to the application root directory. The PNG, GIF, and JPEG formats are supported. You can specify all of the following icon sizes:

```
<icon>
  <image16x16>icons/smallIcon.png</image16x16>
  <image32x32>icons/mediumIcon.jpg</image32x32>
  <image48x48>icons/bigIcon.gif</image48x48>
  <image128x128>icons/biggestIcon.png</image128x128>
</icon>
```

If an image is specified, it must be the size specified. If all sizes are not provided, the closest size will be scaled to fit for a given use of the icon by the operating system.

**Note:** The icons specified are not automatically added to the AIR package. The icon files must be included in their correct relative locations when the application is packaged.

### Signaling the inclusion of an update interface

Normally, AIR installs and updates applications using the default installation dialog. However, you can define your own mechanism for updating an application using the AIR Updater API. To indicate that your application should handle the update process itself, you must include the `handleUpdates` element in the application descriptor file:

```
<handleUpdates/>
```

When the installed version of your application includes the `handleUpdates` element in the application descriptor file and the user then double-clicks on the AIR file for a new version (the `appID` attributes must match), the runtime opens the installed version of the application, rather than the default AIR application installer. Your application logic can then determine how to proceed with the update operation.

**Note:** The `handleUpdates` mechanism only works when the application is already installed and the user double-clicks the AIR file.

For more information, see [“Updating applications programatically” on page 75](#).

### Registering file types

The `fileTypes` property lets you specify any file types to be registered for the AIR application, as in this example:

```
<fileTypes>
  <fileType>
    <name>adobe.VideoFile</name>
    <extension>avf</extension>
    <description>Adobe Video File</description>
    <contentType>application/vnd.adobe.video-file</contentType>
  </fileType>
</fileTypes>
```

The `fileTypes` element is optional. If present, you can specify any number of file type registrations.

The `name` and `extension` properties are required for each `fileType` definition that you include. Note that the extension is specified without the preceding period. The `description` property is optional. If specified, the operating system may use the value of the `description` property to describe the file type. The `contentType` property is also optional.

When a file type is registered with an AIR application, the application will be invoked whenever a user opens a file of that type. If the application is already running, AIR will dispatch the invoke event to the running instance. Otherwise, AIR will launch the application first. In both cases, the file name and location can be retrieved from the `InvokeEvent` object dispatched by the application `Shell` object.

**See also**

“Capturing command-line arguments” on page 99

# Chapter 11: Working with windows

You use the classes provided by the Adobe Integrated Runtime (AIR) windowing API to create and manipulate windows. This chapter contains the following topics:

- “AIR window basics” on page 48
- “Creating windows” on page 52
- “Window transparency” on page 50
- “Maximizing, minimizing, and restoring a window” on page 60
- “Listening for window events” on page 64
- “Using full-screen window mode” on page 65

If you're developing Adobe AIR applications using Flex, refer to the following topics:

- [“Creating a transparent window application” on page 273](#)
- [“Interacting with a window” on page 7](#)
- [“Launching native windows” on page 288](#)
- [“Using the WindowedApplication component” on page 28](#)

## AIR window basics

The windowing API contains the following classes.

Package	Classes
flash.display	NativeWindow, NativeWindowInitOptions  Window string constants are defined in the following classes: NativeWindowDisplayState, NativeWindowResize, NativeWindowSystemChrome
flash.events	NativeWindowBoundsEvent, NativeWindowDisplayStateEvent, NativeWindowErrorEvent
flash.system	NativeWindowCapabilities

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 look-and-feel of your application. The windows you create can look like a standard desktop application, matching Apple style when run on the Mac, and conforming to Microsoft conventions when run on Windows. Or you can use the skinnable, extendible chrome provided by the Flex framework to establish your own style no matter where your application is run. You can even draw your own windows with vector and bitmap artwork with full support for transparency and alpha blending against the desktop. Tired of rectangular windows? Draw a round one.



AIR supports two distinct APIs for working with windows: the Flash-oriented `NativeWindow` class and the HTML-oriented JavaScript `Window` class. Windows created directly with the `NativeWindow` class use the Flash stage and display list. To add a visual object to a `NativeWindow`, you add the object to the display list of the window's stage. Windows created with JavaScript use HTML, CSS, and JavaScript to display content. To add a visual object to an HTML window, you add that content to the DOM (using any common browser-related technique).

**Note:** *HTML windows are a special category of `NativeWindow`. The AIR host adds a `nativeWindow` property to HTML windows that provides access to the underlying `NativeWindow` instance.*

The first window of your application is automatically created for you by AIR based on the parameters specified in the `rootContent` element of the application descriptor file. If the root content is a SWF file, a `NativeWindow` is created and the SWF is loaded into the window. If the root content is an HTML file, an HTML window is created and the HTML page is loaded.

Native windows use an event-based programming model. Changing any properties or calling methods on the window object that may affect the display or behavior of application components dispatches notification events to which interested components can listen. For example, when the system chrome maximize button is clicked by a user the following sequence of events occurs:

- 1 The user clicks the maximize button.
- 2 A `displayStateChanging` event is dispatched by the window
- 3 If no registered listeners cancel the event, the window is maximized
- 4 A `displayStateChange` event is dispatched by the window to notify listeners that the change has been made. In addition, events for related changes are dispatched:
- 5 A move event is dispatched if the top, left corner of the window moved because of the maximize operation.
- 6 A resize event is dispatched if the window size changed because of the maximize operation

A similar sequence of events is dispatched for minimizing, restoring, closing, moving, and resizing a window.

For detailed information about the window API classes, methods, properties, and events, see the [AIR ActionScript 3.0 Language Reference \(http://www.adobe.com/go/learn\\_flex3\\_aslr\)](http://www.adobe.com/go/learn_flex3_aslr). For general information about using the Flash display list, see the “Display Programming” section of the [Programming ActionScript 3.0 \(http://www.adobe.com/go/programmingAS3\)](http://www.adobe.com/go/programmingAS3) reference.

## See also

“The application descriptor file structure” on page 42

## Styles of Native Windows

The basic appearance and functional style of window is determined by three properties: `type`, `systemChrome`, and `transparent`. The `type` property sets the function of the window. The `systemChrome` property sets whether the window uses chrome supplied by native operating system. The `transparent` property sets whether the supports alpha blending with the desktop environment. These properties are set on the `NativeWindowInitOptions` object used to create the window and cannot be changed. The properties of the initial window created automatically on application start up are set in the application descriptor file. The `type` property cannot be set in the application descriptor and is always normal.

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

## Window Type

The type property combines certain chrome and visibility attributes of the native operating system to create three functional types of window. The type property can only be set when creating a new `NativeWindow`. AIR supports the following window types:

**Normal** A typical window. Normal windows use the full-size chrome and appear on the Windows task bar 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 task bar and the Mac OS-X window menu.

**Lightweight** Lightweight windows have little or no chrome and do not appear on the Windows task bar and the Mac OS X window menu. In addition, lightweight windows do not have the System (Alt-Space) menu on Microsoft Windows. Lightweight windows are suitable for notification bubbles and controls such as combo-boxes that open a short-lived display area. When the lightweight type is used, `systemChrome` must be set to "none".

*Note: In this Beta release, the modal window type is not supported.*

## Window chrome

Window chrome is the set of controls that allow users to manipulate a window in the desktop environment. For an AIR application, you have the following choices for window chrome:

**System chrome** System chrome displays your window within the set of standard controls created and styled by the user's operating system. Use system chrome to make your application look like a standard desktop application for the operating system in which it is run. 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. If system chrome is used for a window, the `transparency` property must be set to `false`.

*Note: In this Beta release, the Alternate systemChrome setting is not supported.*

**Flex chrome** When using the `Flex WindowedApplication` and `Window` components, your window will be displayed with chrome provided by the Flex framework. If the Flex components are used along with system chrome, the Flex chrome is not displayed.

**Custom chrome** When you create a window with no system chrome and you do not use the `Flex mx:WindowedApplication` or `mx:Window` components, then you must add your own controls to handle the interactions between a user and the window. You are also free to make transparent, non-rectangular windows.

## Window transparency

To allow alpha blending of a window with the desktop or other windows, set the window's `transparent` property to `true`. The `transparent` property must be set before the window is created and cannot be changed. The `transparent` property is set in the `NativeWindowInitOptions` object used to create a window. (For the initial window created for your application, the initialization options are taken from attributes of the `rootContent` element in the application descriptor file.)

A transparent window has no default background. Any window area not occupied by a display object will be invisible. If a display object has an alpha setting of less than one, then any thing below the object will show through, including other display objects in the same window, other windows, and the desktop. Rendering large alpha-blended areas can be slow, so the effect should be used conservatively.

Transparent windows are useful when you want to create:

- Applications with borders that are irregular in shape
- Applications that must "fade out" or appear to be invisible

Transparency cannot be used for windows with system chrome.

### Transparency in an MXML application window

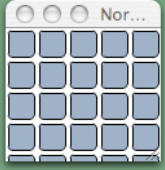

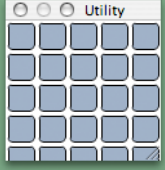

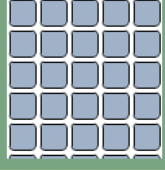
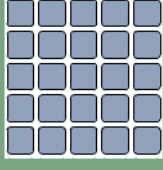
If your Flex application uses CSS style properties that set a background image or color, your window will not be transparent. Likewise, if the Application tag in your MXML file defines a background color, this setting will override a transparent window mode. To ensure the transparency of your application window, include the following code in your style sheet or in the <mx:Style> element that is contained in your application MXML file:

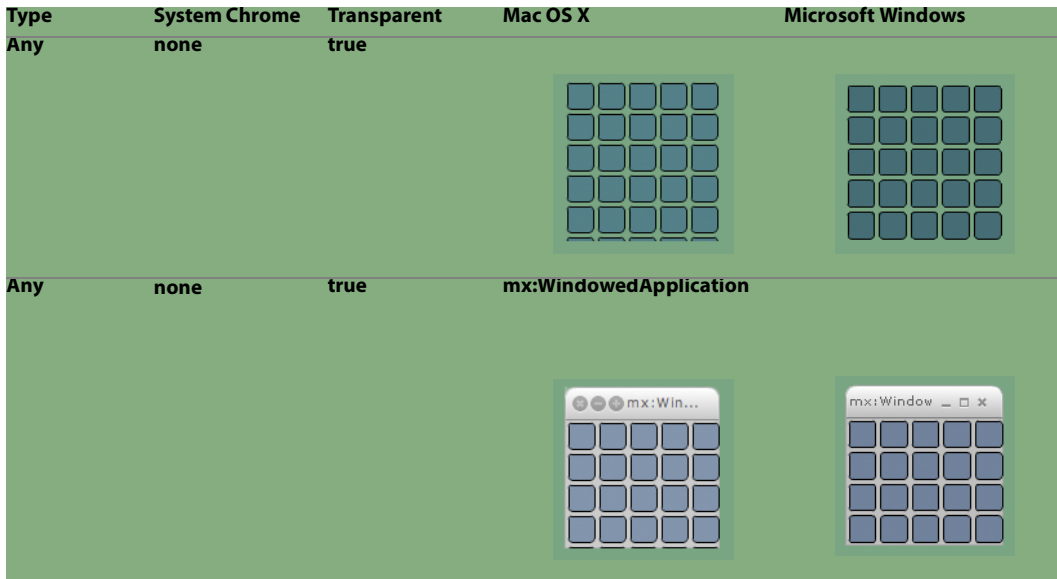
```
Application
{
    background-image:"";
    background-color:"";
}
```

These declarations suppress the appearance of background color and image in the application window.

### A visual window catalogue

The following table demonstrates the visual effects of different combinations of window property settings.

Type	System Chrome	Transparent	Mac OS X	Microsoft Windows
Standard	normal	Not supported		
Utility	normal	Not supported		
Any	none	false		



### Limitations in the Beta Release

The following window-related features are not supported in this Beta release:

- The windowing API does not currently support the Toolbars|OS X Toolbar or the OS X Proxy Icon.
- System tray icons
- Programmatic control of application icons
- Information about the desktop screens, such as the usable bounds, number, and relative location is not available
- Native window menus.

## Creating windows

AIR provides the following primary means for creating applications windows:

- AIR automatically creates the first window for every application. This window is initialized according to the settings defined in the application descriptor file. If the root content identified in the descriptor is a SWF file, then you can access the properties and methods of the window instance through the `Stage.window` property and the `NativeWindow` API. In addition, the main class of the SWF file must extend from `Sprite` or a subclass of `Sprite`. (The `Flex WindowedApplication` and `Application` components, which are `Sprite` subclasses, are used as the main class of Flex applications.) If the root content is an HTML file, then you can access the properties and methods of the window through the JavaScript `Window` object.
- You can create a new instance of the `NativeWindow` class. New `NativeWindows` are initialized by the `NativeWindowInitOptions` object passed to the window constructor. You can access the properties and methods directly from the object reference returned by the constructor.
- You can use the JavaScript `Window.open()` method to open new windows from JavaScript code. The properties and methods of JavaScript-created windows can only be accessed through JavaScript and the window can only display HTML. JavaScript-created HTML windows automatically include system chrome.

**Note:** You can create new `NativeWindows` from JavaScript code (through the `AIR window.runtime` property).

## Creating a new NativeWindow

To create a new `NativeWindow`, create a `NativeWindowInitOptions` object and pass it to the `NativeWindow` constructor.

```
var options:NativeWindowInitOptions = new NativeWindowInitOptions();
options.systemChrome = NativeWindowSystemChrome.STANDARD;
options.transparent = false;
var newWin:NativeWindow = new NativeWindow(false, options);
```

The first parameter of the `NativeWindow` constructor specifies whether the window should be visible as soon as it is created. To avoid displaying the intermediate states of the window as you set its bounds, position, and contents, set this parameter to `false`. When you have finished initializing the window, set the `NativeWindow.visible` property to `true`. The second parameter is a `NativeWindowInitOptions` object. This object sets those properties of a window that cannot be changed after the window has been created.

**Note:** Setting `systemChrome="standard"` and `transparent="true"` is not a supported combination.

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

**Note:** To determine the maximum and minimum window sizes allowed on the current operating system, use the `NativeWindowCapabilities` class:

```
var maxOSSize:Point = NativeWindowCapabilities.windowMaxSize;
var minOSSize:Point = NativeWindowCapabilities.windowMinSize;
```

## Adding content to a window

To add content to a native window, add display objects to the window stage. You can create display objects dynamically or load existing content with the `flash.display.Loader` class. Within an HTML window, you can add content by changing the `location` property (to load a new page) or inserting HTML content into the DOM.

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-resource:` URL scheme, will have full privileges to access all the AIR APIs. Any content loaded from outside this sandbox will be restricted both from accessing the security-restricted AIR APIs and from cross-scripting other content. JavaScript content outside the application sandbox will not be able to use the `nativeWindow` or `htmlControl` properties of the JavaScript Window object.

To allow safe cross-scripting, you can use the `flash.system.Door` API to create a guarded communication gateway that provides a limited interface between application content and non-application content.

### Loading a SWF or image

You can load Flash or images into the display list of a native window using the `flash.display.Loader` class.

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.net.URLRequest;
    import flash.display.Loader;
```

```

public class LoadedSWF extends Sprite
{
    public function LoadedSWF() {
        var loader:Loader = new Loader();
        loader.load(new URLRequest("visual.swf"));
        loader.contentLoaderInfo.addEventListener(Event.COMPLETE, loadFlash);
    }

    private function loadFlash(event:Event):void {
        addChild(event.target.loader);
    }
}

```

Visual Flash content added to an HTML window must be displayed either on top of the HTML content as an overlay, or beneath transparent HTML content. Such content must also be positioned and resized separately from the HTML content.

You can load a SWF file that contains library code for use in an HTML-based application. The simplest way to load a SWF in an HTML window is to use the script tag, but you can also use the `flash.display.Loader` API directly.

**Note:** Older SWF files created using ActionScript 1 or 2 will share global state such as class definitions, singletons, and global variables if they are loaded into the same window. If such a SWF relies on untouched global state to work correctly, it cannot be loaded more than once into the same window, or loaded into the same window as another SWF using overlapping class definitions and variables. This content can be loaded into separate windows.

### Loading HTML content into a NativeWindow

To load HTML content into a NativeWindow, you must add an HTMLControl to the window stage and load the HTML content into the HTMLControl.

```

//newWindow is a NativeWindow instance
var htmlView:HTMLControl = new HTMLControl();
html.width = newWindow.width;
html.height = newWindow.height;

//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 an HTML window, use a JavaScript method such as `window.open`.

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

### Loading Flash content within an HTML page

In this Beta release, displaying Flash content embedded in an HTML page is not supported. Any Flash objects in the page will be displayed as blank rectangles. You can, however, load or create Flash content using the AIR APIs and add it to the HTML window as an overlay or underlay to the HTML layer.

### Adding Flash 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.

To add a display object above the HTML layer, use the `addChild()` method of the `window.nativeWindow.stage` property. The `addChild()` method will add 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 will move 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 `window.htmlControl` 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.htmlControl.paintsDefaultBackground = false;
```



```
var bottom = new Bouncer(60,0xff2233);
var top = new Bouncer(30,0x2441ff);

//listen for the enterFrame event
window.htmlControl.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. Nemo enim ipsam voluptatem quia
voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui
ratione voluptatemsequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia
dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora
incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam,
quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea
commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit
esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas
nulla pariatur?</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. Et harum quidem
rerum facilis est et expedita distinctio. Nam libero tempore, cum soluta nobis est
eligendi optio cumque nihil impedit quo minus id quod maxime placeat facere possimus,
omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam
et aut officiis debitis aut rerum necessitatibus saepe eveniet ut et voluptates
repudiandae sint et molestiae non recusandae. Itaque earum rerum hic tenetur a
sapiente delectus, ut aut reiciendis voluptatibus maiores alias consequatur aut
perferendis doloribus asperiores repellat.</p>
</body>
</html>
```

**Note:** To access the `nativeWindow` and `htmlControl` properties of the JavaScript Window object, the HTML page must have been loaded from the `app-resource` directory. This will always be the case for the root page in an HTML-based application, but may not be true for other content.

### See also

[“Adding HTML content to SWF-based applications” on page 43](#)

[“About AIR security” on page 9](#)

[“Scripting between content in different domains” on page 67](#)

### Example: Creating windows with ActionScript

The following example illustrates how to create new windows. The example uses MXML to simply run an ActionScript function.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute" applicationComplete="createNativeWindow();" >
    <mx:Script>
        <![CDATA[
            public function createNativeWindow():void{
                //create the init options
                var options:NativeWindowInitOptions =
                    new NativeWindowInitOptions();

                options.transparent = false;
                options.systemChrome = NativeWindowSystemChrome.STANDARD;
                options.type = NativeWindowType.NORMAL;

                //create the window
                var newWindow:NativeWindow = new NativeWindow(false,options);
                newWindow.title = "A title";
                newWindow.width = 600;
                newWindow.height = 400;

                //add a sprite to the window
                newWindow.stage.align = StageAlign.TOP_LEFT;
                newWindow.stage.scaleMode = StageScaleMode.NO_SCALE;

                //show the new window
```

```

        newWindow.visible = true;
    }
}]>
</mx:Script>
</mx:WindowedApplication>

```

## Manipulating windows

This section highlights using the properties and methods of the `NativeWindow` class to manipulate the appearance and behavior of application windows.

### Getting a `NativeWindow` instance

To manipulate a window, you must first get the window instance. You can get a window instance from one of the following places:

**The window constructor** That is, the window constructor for a new `NativeWindow`.

**The window stage** That is, `stage.window`.

**Any display object on the stage** That is, `myDisplayObject.stage.window`.

**A window event** The `target` property of the event object references the window that dispatched the event.

**The global `nativeWindow` property of an `HTMLControl` or `HTML window`** That is `window.nativeWindow`.

Because the `Flex Application`, `WindowedApplication`, and `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.window;
                //set window properties
                appWindow.visible = true;
            }
        ]]>
    </mx:Script>
</WindowedApplication>

```

**Note:** Until the `WindowedApplication` or `Window` components is added to the window stage by the Flex framework, the component's `stage` property will be `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.

## Maximizing, minimizing, and restoring a window

To maximize the window, use the `NativeWindow.maximize()` method.

```
myWindow.maximize();
```

To minimize the window, use the `NativeWindow.minimize()` method.

```
myWindow.minimize();
```

Restoring a window returns it to the size that it was before it was either minimized or maximized.

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.

```
myWindow.restore();
```

## Closing a window

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 valid, but accessing most properties and methods on a closed window will generate an `IllegalOperationError`. You can check the `closed` property of a window to test whether a window has been closed.

If the `shell.autoExit` property is `true`, which is the default, then the application exits when its last window closes.

## Allowing cancellation of window operations

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 will 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 actually 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:

```
public function onCloseCommand(event:MouseEvent):void{
    var closingEvent:Event = new Event(Event.CLOSING,true,true);
    dispatchEvent(closing);
    if(!closingEvent.isDefaultPrevented()){
```

```

        win.close();
    }
}

```

Note that 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.

### Example: Minimizing, maximizing, restoring and closing a window

The following short MXML application demonstrates the `Window` `maximize()`, `minimize()`, `restore()`, and `close()` methods.

```

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

<mx:WindowedApplication
    xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical">

    <mx:Script>
    <![CDATA[
    public function minimizeWindow():void
    {
        this.stage.window.minimize();
    }

    public function maximizeWindow():void
    {
        this.stage.window.maximize();
    }

    public function restoreWindow():void
    {
        this.stage.window.restore();
    }

    public function closeWindow():void
    {
        this.stage.window.close();
    }
    ]]>
    
```

```

    }
  ]]>
</mx:Script>

<mx:VBox>

    <mx:Button label="Minimize" click="minimizeWindow()" />

    <mx:Button label="Restore" click="restoreWindow()" />

    <mx:Button label="Maximize" click="maximizeWindow()" />

    <mx:Button label="Close" click="closeWindow()" />

</mx:VBox>

</mx:WindowedApplication>

```

### Example: Resizing and moving windows

An AIR application can initiate a system-controlled operation to resize or move its containing window. The process that is used to complete the operation depends on whether the call to the method that completes the operation is made from within a `mouseDown` event.

**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 59.

To resize a window, use the `NativeWindow.startResize()` method. This method triggers a system-controlled resizing of the window. When this method is called from a `mouseDown` event, the resizing process is mouse-driven and completes when the operating system receives a `mouseUp` event. A call to the `NativeWindow.startResize()` method that is not made from a `mouseDown` event triggers a system-controlled resizing operation that may be mouse-driven or keyboard-driven but remains consistent with the default resizing sequence for the operating system.

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. A call to the `NativeWindow.startMove()` method that is not made from a `mouseDown` event triggers a system-controlled move that may be mouse-driven or keyboard-driven but remains consistent with the default move sequence for the operating system.

For more information about the `startResize` and `startMove` methods, see the AIR ActionScript 3.0 Language Reference.

The following example shows how to initiate resizing and moving operations on a window:

```

package
{
    import flash.display.Sprite;
    import flash.events.MouseEvent;
    import flash.display.NativeWindowResize;

```

```
public class NativeWindowResizeExample extends Sprite
{
    public function NativeWindowResizeExample():void
    {
        // Fills a background area.
        this.graphics.beginFill(0xFFFFFFFF);
        this.graphics.drawRect(0, 0, 400, 300);
        this.graphics.endFill();

        // Creates a square area where a mouse down will trigger a resize.
        var resizeHandle:Sprite = createSprite(0xCCCCCC, 20, this.width - 20,
this.height - 20);
        resizeHandle.addEventListener(MouseEvent.CLICK, onStartResize);

        // Creates a square area where a mouse down will trigger a move.
        var moveHandle:Sprite = createSprite(0xCCCCCC, 20, this.width - 20, 0);
        moveHandle.addEventListener(MouseEvent.CLICK, onStartMove);
    }

    public function createSprite(color:int, size:int, x:int, y:int):Sprite
    {
        var s:Sprite = new Sprite();
        s.graphics.beginFill(color);
        s.graphics.drawRect(0, 0, size, size);
        s.graphics.endFill();
        s.x = x;
        s.y = y;
        this.addChild(s);
        return s;
    }

    public function onStartResize(event:MouseEvent):void
    {
        this.stage.window.startResize(NativeWindowResize.BOTTOM_RIGHT);
    }
}
```

```
public function onStartMove(event:MouseEvent):void
{
    this.stage.window.startMove();
}
}
```

## Listening for window events

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:

```
myWindow.addEventListener(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.

Events in the `flash.events` package:

- ACTIVATE
- DEACTIVATE
- CLOSING
- CLOSE
- FOCUS\_IN
- FOCUS\_OUT

`NativeWindowBoundsEvent`:

Use the `beforeBounds` and `afterBounds` properties to determine the window bounds before and after the impending or completed change.

- MOVING
- MOVE
- RESIZING
- RESIZE

`NativeWindowDisplayStateEvent`:

Use the `beforeDisplayState` and `afterDisplayState` properties to determine the window display state before and after the impending or completed change.

- DISPLAY\_STATE\_CHANGING
- DISPLAY\_STATE\_CHANGE



## Using full-screen window mode

For the Beta 1 release of AIR, setting the `displayState` property of the Stage to `StageDisplayState.FULL_SCREEN` puts 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.

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;
                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>
```

# Chapter 12: Distributing, installing, and running AIR applications

AIR applications are easy to install and run. The *seamless install* feature lets users install the latest version of the AIR runtime (if it is not installed) when clicking the link to install a specific AIR application. Once the AIR application is installed, users simply double-click the application icon to run it, just like any other desktop application.

## Distributing an AIR application file

Once you package an AIR application (Flex Builder users see [“Package AIR applications with Flex Builder” on page 26](#), SDK users see [“Packaging an AIR application using the AIR Developer Tool” on page 34](#)), there are a couple of ways to distribute it:

- Using a seamless install installation link in a web page.
- You can install it by sending the AIR package to the end user just as you would distribute any file. For example, you can send the AIR package as an e-mail attachment or as a link in a web page.

## Distributing and installing using the seamless install feature

The seamless install feature lets you provide a link in a web page that lets the user install an AIR application by simply clicking the link. If the AIR runtime is not installed, the user is given the option to install it. The seamless install feature also lets users install the AIR application without downloading the AIR file to their machine.

The seamless install link is a link in a SWF file (badge.swf) that has special capabilities that allow it to download the AIR runtime. The SWF file, and its source code, is provided to you for distribution on your website.

### Add a seamless install link to a page

- 1 Add the badge.swf file to the web server.

This file is provided in the lib directory of the AIR SDK. To customize the SWF file, see [“Modifying the badge.swf file” on page 4](#).

- 2 Embed the SWF file in the HTML page, at the point that you want the link (contained in the badge.swf file) to appear.

- 3 Add FlashVars parameter definitions for the url variable, setting it to reference the path to your AIR file. Also, modify the path to the badge.swf file (if needed).

For details, see [“Adjusting the code in the HTML page that contains the badge.swf file” on page 2](#).

### Installing the AIR application from a seamless install link in a web page

**Important:** The seamless install functionality does not apply in the pre-release version of the beta version of AIR. This information is included to provide you with information on how the feature is planned to work in the upcoming public beta release.

Once you have added the seamless install link to a page, you (or any user) can install the AIR application by clicking the link in the SWF file.

- 1 Navigate to the HTML page in a web browser that has Flash Player (version 6 or later) installed.

2 In the web page, click the link in the badge.swf file.

- If you have installed the AIR runtime, skip to the next step.
- If you have not installed the AIR runtime, a dialog box is displayed asking whether you would like to install it. Install the AIR runtime (see “Install the runtime” on page 2), and then proceed with the next step.

3 In the Installation window, leave the default settings selected, and then click Continue.

On a Windows computer, AIR automatically does the following:

- Installs the application into Program Files\AIR Test\AIR Hello World
- Creates a desktop shortcut for application
- Creates a Start Menu shortcut in AIR Test\AIR Hello World
- Adds an entry for application in the Add/Remove Programs Control Panel

On Mac OS, the installer adds the application to the Applications subdirectory of the user directory (for example, in Macintosh HD/Users/JoeUser/Applications/).

4 When the installation is complete, click Finish.

5 Select the options you want, and then click the Install button.

## Adjusting the code in the HTML page that contains the badge.swf file

**Important:** The seamless install functionality does not apply in the pre-release version of the beta version of AIR. This information is included to provided you with information on how the feature is planned to work in the upcoming public beta release.

You need to add FlashVars parameter definitions for the url variable, and modify the path to the badge.swf file (if needed). SWF files are embedded in a page using the EMBED and OBJECT tags, as in the following:

```
<object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
  id="HelloWorld" width="215" height="138"
  codebase="http://fpdownload.macromedia.com/get/flashplayer/current/swflash.cab">
  <param name="movie" value="badge.swf" />
  <param name="FlashVars" value="appInstallUrl=AIRFileURL.air" />
  <embed src="badge.swf" quality="high" bgcolor="#869ca7"
    FlashVars="url=AIRFileURL.air"
    width="215" height="138" name="HelloWorld" align="middle"
    play="true"
    type="application/x-shockwave-flash"
    pluginspage="http://www.adobe.com/go/getflashplayer">
  </embed>
</object>
```

The boldface lines represent code that you need to add and modify to reflect the URL of your AIR file and the badge.swf file.

However, this is a simplified version of code embedding a SWF file. In most HTML pages, SWF files are embedded with more complicated code. For example, Flex Builder and Flash both insert code as described in the Flash Player Detection Kit ([http://www.adobe.com/products/flashplayer/download/detection\\_kit/](http://www.adobe.com/products/flashplayer/download/detection_kit/)), as shown below:

```
<script language="JavaScript" type="text/javascript">
    <!--
    // Version check for the Flash Player that has the ability to start Player Product Install
    (6.0r65)
    var hasProductInstall = DetectFlashVer(6, 0, 65);

    // Version check based upon the values defined in globals
    var hasRequestedVersion = DetectFlashVer(requiredMajorVersion, requiredMinorVersion,
    requiredRevision);

    // Check to see if a player with Flash Product Install is available and the version does
    not meet the requirements for playback
    if ( hasProductInstall && !hasRequestedVersion ) {
        // MMdoctitle is the stored document.title value used by the installation process to
        close the window that started the process
        // This is necessary in order to close browser windows that are still utilizing the
        older version of the player after installation has completed
        // DO NOT MODIFY THE FOLLOWING FOUR LINES
        // Location visited after installation is complete if installation is required
        var MMPlayerType = (isIE == true) ? "ActiveX" : "PlugIn";
        var MMredirectURL = window.location;
        document.title = document.title.slice(0, 47) + " - Flash Player Installation";
        var MMdoctitle = document.title;

        AC_FL_RunContent (
            "src", "playerProductInstall",
            "FlashVars",
            "MMredirectURL="+MMredirectURL+'&MMplayerType='+MMPlayerType+'&MMdoctitle='+MMdoctitle+"&a
            ppInstallUrl=AIRFileURL.air",
            "width", "215",
            "height", "138",
            "align", "middle",
            "id", "badge",
            "quality", "high",
            "bgcolor", "#869ca7",
            "name", "badge",
            "allowScriptAccess", "sameDomain",
            "type", "application/x-shockwave-flash",
            "pluginspage", "http://www.adobe.com/go/getflashplayer"
        );
    } else if (hasRequestedVersion) {
        // if we've detected an acceptable version
        // embed the Flash Content SWF when all tests are passed
        AC_FL_RunContent (
            "src", "badge",
            "width", "215",
            "height", "138",
            "align", "middle",
            "id", "badge",
            "quality", "high",
            "bgcolor", "#869ca7",
```

```

        "name", "badge",
        "flashvars", 'historyUrl=history.htm%3F&lconid=' + lc_id + '
&appInstallUrl=AIRFileURL.air ',
        "allowScriptAccess", "sameDomain",
        "type", "application/x-shockwave-flash",
        "pluginspage", "http://www.adobe.com/go/getflashplayer"
    );
} else { // flash is too old or we can't detect the plugin
    var alternateContent = 'Alternate HTML content should be placed here. '
    + 'This content requires the Adobe Flash Player. '
    + '<a href=http://www.adobe.com/go/getflash/>Get Flash</a>';
    document.write(alternateContent); // insert non-flash content
}
// -->
</script>
<noscript>
    <object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
        id="badge" width="100%" height="100%"
        codebase="http://fpdownload.macromedia.com/get/flashplayer/current/swflash.cab">
        <param name="movie" value="badge.swf" />
        <param name="FlashVars" value="appInstallUrl=AIRFileURL.air" />
        <param name="quality" value="high" />
        <param name="bgcolor" value="#869ca7" />
        <param name="allowScriptAccess" value="sameDomain" />
        <embed src="badge.swf" quality="high" bgcolor="#869ca7"
            FlashVars="appInstallUrl=AIRFileURL.air"
            width="215" height="138" name="badge" align="middle"
            play="true"
            loop="false"
            quality="high"
            allowScriptAccess="sameDomain"
            type="application/x-shockwave-flash"
            pluginspage="http://www.adobe.com/go/getflashplayer">
        </embed>
    </object>
</noscript>

```

In this code, there are many more lines to modify, as the code embeds the SWF file in a number of ways (including via JavaScript calls). Be sure to modify `AIRFileURL.air` to point to the URL (or relative path) of your AIR file, and modify it each place it is referenced in the code. If necessary, also modify the path to the `badge.swf` file, and modify it each place it is referenced in the code.

## Modifying the badge.swf file

**Important:** The seamless install functionality does not apply in the pre-release version of the beta version of AIR. This information is included to provided you with information on how the feature is planned to work in the upcoming public beta release.

The AIR SDK provides the source files for the `badge.swf` file. These files are included in the `src` folder of the SDK:

**badge.fla** The source Flash file used to compile the `badge.swf` file. The `badge.fla` file compiles into a SWF 6 file (which can be loaded in Flash Player versions 6 and later).

**install.as** An ActionScript 2.0 script file referenced in the `badge.fla` file. This script is written with ActionScript 2.0 class (rather than ActionScript 3.0) for support in Flash Player versions 6 and later. Details are provided below.

You can use Flash to redesign the visual interface of the badge.flc file.

The install.as file includes code for installing the AIR runtime. It uses the flash.system.ProductManager class, which is used to download and install extensions to Flash Player (in this case, the AIR runtime) that originate from www.adobe.com. In the badge.swf file, the class is used to install the AIR Detection Add-In, a browser add-in that is used to install the AIR runtime, check the runtime version, and launch the AIR application installer. The flash.system.ProductManager class is generally undocumented because it can only be used to install extensions that originate from www.adobe.com. We recommend that you do not modify the script in the install.as file.

If the AIR Detection Add-In is not installed, the user is asked to grant permission to install it. Once the AIR Detection Add-In is installed, it launches the AIR runtime. If the AIR runtime does not exist, the AIR Detection Add-In asks the user to grant permission to install the AIR runtime. Once the AIR runtime is installed, the badge.swf file proceeds to install the requested AIR file, by opening the AIR application installer (which also gives the user the choice to proceed with installing the AIR application).

Both the AIR Detection Add-In and the AIR runtime are signed.

Neither the badge.swf file, the AIR Detection Add-In, nor the AIR application installer send any information about the identity of the AIR file you are installing to adobe.com.

## Distributing and installing an AIR file without using the seamless install feature

If you choose not to use the seamless install feature, you can simply send the AIR file to the recipient. For example, you can send the AIR file as an e-mail attachment or as a link in a web page. Once the user downloads the AIR application, the user follows these instructions to install it.

- 1 Double-click the AIR file.
- 2 In the Installation window, leave the default settings selected, and then click Continue.

In Windows, AIR automatically does the following:

- Installs the application into the Program Files directory
- Creates a desktop shortcut for application
- Creates a Start Menu shortcut
- Adds an entry for application in the Add / Remove Programs Control Panel

In the Mac OS, by default the application is added to the Applications subdirectory of the user directory.

If the application is already installed, the installer gives the user the choice of opening the existing version of the application or updating to the version in the downloaded AIR file. The installer identifies the application using the application ID (appID) in the AIR file.

- 3 When the installation is complete, click Finish.

An application can also install a new version via ActionScript or JavaScript. For more information, see [“Updating applications programatically” on page 75](#).

## Running an AIR application

Once you have installed the runtime and the AIR application, running the application is as simple as running any other desktop application:

- On a Windows computer, double-click the application's icon (which may be installed on the desktop or in a folder), or select the application from the Start menu.

- On Mac OS, double-click the application in the folder in which it was installed. The default installation directory is the Applications subdirectory of the user directory.

# Chapter 13: Working with the file system

You use the classes provided by the Adobe Integrated Runtime (AIR) file system API to access the file system of the host computer.

This chapter contains the following sections:

- [AIR file basics](#)
- [Working with File objects](#)
- [Getting file system information](#)
- [Working with directories](#)
- [Working with files](#)

## See also

[“Building a text-file editor” on page 278](#)

[“Reading and writing from an XML preferences file” on page 11](#)

[“Building a directory search application” on page 15](#)

## AIR file basics

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:

**File** A 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 FileMode class defines string constants used in the `fileMode` parameter of the `open()` and `openAsync()` methods of the `FileStream` class. The `fileMode` parameter of these methods determines the capabilities available to the `FileStream` object once the file is opened, which include writing, reading, appending, and updating.

**FileStream** A `FileStream` object is used to open files for reading and writing. Once you’ve created a File object that points to a new or existing file, you pass that pointer to the `FileStream` object so that you can open and then manipulate data within the file.

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.listDirectory()` and `File.listDirectoryAsync()`
- `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()` or by calling `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 need to 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, may be paused.

## Working with File objects

The `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 Flash Player as well as AIR, represents a pointer to a file, but the `File` class adds properties and methods that are not exposed in Flash Player (in a SWF running in a browser), due to security considerations.

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 files, 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 new file or directory.

### Paths of File objects

Each `File` object has two properties that each define its path:

**`nativePath`** This 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". Note that a `nativePath` property uses the backslash (\) character as the directory separator character on Windows, and it uses the forward slash (/) character on Mac OS.

**`url`** 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 file, and these are described in [“Supported URL schemes” on page 12](#).

The `File` class includes properties for pointing to standard directories on both Mac and Windows, and these are described in the following section.

## Pointing a File object to a directory

There are a number of ways to set a File object to point to a directory.

### Pointing to an explicit directory

You can point a File object to the user's home directory. On Windows, the home directory is the parent of the "My Documents" directory (for example, "C:\Documents and Settings\*userName*\My Documents"). On Mac OS, it is the *Users/userName* directory. The following code sets a File object to point to an AIR Test subdirectory of the home directory:

```
var file:File = File.userDirectory.resolve("AIR Test");
```

You can point a File object to the user's documents directory. On Windows, this is typically the "My Documents" directory (for example, "C:\Documents and Settings\*userName*\My Documents"). On Mac OS, it is the *Users/userName/Documents* directory. The following code sets a File object to point to an AIR Test subdirectory of the documents directory:

```
var file:File = File.documentsDirectory.resolve("AIR Test");
```

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:File = File.desktopDirectory.resolve("AIR Test");
```

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 store directory. You may want to use this directory to store 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:File = File.applicationStorageDirectory;  
file = file.resolve("prefs.xml");
```

You can point a File object to the directory in which the application was installed, known as the application resource directory. You can reference this directory using the `File.applicationResourceDirectory` property. You may 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 resource directory:

```
var file:File = File.applicationResourceDirectory;  
file = file.resolve("images");
```

The `File.listRootDirectories()` method lists all root volumes, such as C: and mounted volumes, on a Windows computer. On Mac, this method always returns the unique root directory for the machine (the "/" directory).

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:File = new File();  
file.nativePath = "C:\\\\AIR Test\\";
```

You can use the `resolve()` 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:File = File.userDirectory;  
file = file.resolve("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:String = "file:/C:/AIR Test/";  
var file:File = new File()  
file.url = urlStr;
```

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 AIR Test subdirectory of the C: drive:

```
var file:File = new File();  
file.nativePath = "C:\\AIR Test";
```

For more information, see [“Modifying File paths” on page 12](#).

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 99](#).

### Letting the user browse to select a directory

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; it 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:File = new File();  
file.addEventListener(Event.SELECT, dirSelected);  
file.browseForDirectory();  
function dirSelected(e:Event):void {  
    trace(file.nativePath);  
}
```

### Pointing a File object to a file

There are a number of ways to set the file to which a `File` object points.

#### Pointing to an explicit file path

You can use the `resolve()` method to obtain a path relative to another given path. For example, the following code sets a `File` object to point to a `log.txt` file within the application storage directory:

```
var file:File = File.applicationStorageDirectory;  
file = file.resolve("log.txt");
```

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:String = "file:/C:/AIR Test/test.txt";  
var file:File = new File()  
file.url = urlStr;
```

You can also pass the URL to the `File()` constructor function, as in the following:

```
var urlStr:String = "file:/C:/AIR Test/test.txt";  
var file:File = new File(urlStr);
```

Note that `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";  
trace(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:File = new 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:File = new File();  
file.nativePath = "C:/AIR Test/test.txt";
```

On Windows you can use the forward slash (/) or backslash (\) character as the path delimiter for the `nativePath` property. On Mac OS, use the forward slash (/) character as the path delimiter for the `nativePath`:

```
var file:File = new File(Users/dijkstra/AIR Test/test.txt);
```

For more information, see [“Modifying File paths” on page 12](#).

### Enumerating files in a directory

You can use the `listDirectory()` 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 14](#).

### Letting the user browse to select a file

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()`
- `browseForMultiple()`

For example, the following code presents the user with an “Open” dialog box in which the user can select a file.

```
var fileToOpen:File = File.documentsDirectory;  
selectTextFile(fileToOpen);  
  
function selectTextFile(root:File):void  
{  
    var txtFilter:FileFilter = new FileFilter("Text", "*.as;*.css;*.html;*.txt;*.xml");  
    root.browseForOpen("Open", [txtFilter]);  
    root.addEventListener(Event.SELECT, fileSelected);  
}  
  
function fileSelected(event:Event):void  
{  
    trace(fileToOpen.nativePath);  
}
```

If the application has another browser dialog box open when you call a browse method, the runtime throws an error (hence the `try/catch` structure in the previous example).

## Modifying File paths

You can also modify the path of an existing File object by calling the `resolve()` method or by modifying the `nativePath` or `url` property of the object, as in the following examples (on Windows):

```
var file1:File = File.documentsDirectory;
file1 = file1.resolve("AIR Test");
trace(file1.nativePath); // C:\Documents and Settings\userName\My Documents\AIR Test
var file2:File = File.documentsDirectory;
file2 = file2.resolve("..");
trace(file2.nativePath); // C:\Documents and Settings\userName
var file3:File = File.documentsDirectory;
file3.nativePath += "/subdirectory";
trace(file3.nativePath); // C:\Documents and Settings\userName\My Documents\subdirectory
var file4:File = new File();
file.url = "file:///c:/AIR Test/test.txt"
trace(file3.nativePath); // C:\AIR Test\test.txt
```

When using the `nativePath` property, you use either the forward slash (/) or backslash (\) character as the directory separator character on Windows; use the forward slash (/) character on Mac OS. On Windows, remember to type the backslash character twice in a string literal.

## Supported URL schemes

You can use any of the following URL schemes in defining the `url` property of a File object:

**file** Use this to specify a path relative to the root of the file system. For example:

```
file:///c:/AIR Test/test.txt
```

**app-resource** Use this to specify a path relative to the root directory of the installed application (the directory that contains the `application.xml` file for the installed application). For example, the following path points to a `test.log` file in a logs subdirectory of the directory of the installed application:

```
app-resource:/logs/test.log
```

**app-storage** Use this 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 `prefs.xml` file in a settings subdirectory of the application store directory:

```
app-storage:/settings/prefs.xml
```

## Finding the relative path between two files

You can use the `relativize` method to find the relative path between two files:

```
var file1:File = File.documentsDirectory.resolve("AIR Test");
var file2:File = File.documentsDirectory
file2 = file2.resolve("AIR Test/bob/test.txt");

trace(file1.relativize(file2)); // bob/test.txt
```

The second parameter of the `relativize()` method, the `useDotDot` parameter, allows for `..` syntax to be returned in results, to indicate parent directories:

```
var file1:File = File.documentsDirectory;
file1 = file1.resolve("AIR Test");
var file2:File = File.documentsDirectory;
file2 = file2.resolve("AIR Test/bob/test.txt");
var file3:File = File.documentsDirectory;
file3 = file3.resolve("AIR Test/susan/test.txt");

trace(file2.relativize(file1, true)); // ../../
trace(file3.relativize(file2, true)); // ../../bob/test.txt
```

## Obtaining canonical versions of file names

File and path names are usually not case sensitive. In the following, two `File` objects point to the same file:

```
File.documentsDirectory.resolve("test.txt");
File.documentsDirectory.resolve("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:File = File.documentsDirectory.resolve("AIR test");
trace(file.nativePath); // ... AIR test
file.canonicalize();
trace(file.nativePath); // ... AIR Test
```

The `canonicalize` method converts the `nativePath` object to use the correct capitalization for the file or directory name.

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:File = new File();
path.nativePath = "C:\\AIR-1";
path.canonicalize();
trace(path.nativePath); // C:\AIR Test
```

## Getting file system information

The `File` class includes the following static properties that provide some useful information about the file system:

**File.encoding** 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.

**File.lineEnding** The line-ending character sequence used by the host operating system. On Mac OS, this is the line-feed character. On Windows, this is the carriage return character followed by the line-feed character.

**File.separator** The host operating system's path component separator character. On Mac OS, this is the forward slash ("/") character. On Windows, it is the backslash ("\") character.

The `Capabilities` class also includes useful system information that may be useful when working with files:

**Capabilities.hasIME** Specifies whether the player is running on a system that does (`true`) or does not (`false`) 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.

## Working with directories

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 9](#).

### Creating directories

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:File = File.userDirectory.resolve("AIR Test");  
dir.createDirectory();
```

If the directory already exists, the `createDirectory()` method does nothing.

Also, in some modes, a `FileStream` object will create 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 18](#).

### Creating a temporary directory

The `File` class includes a `createTempDirectory()` method, which creates a new directory in the temporary directory folder for the System, as in the following example:

```
var temp:File = File.createTempDirectory();
```

The `createTempDirectory()` method automatically creates a unique temporary directory (saving you the work of determining a new unique location).

You may 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.

### Enumerating directories

You can use the `listDirectory()` method or the `listDirectoryAsync()` 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:File = File.documentsDirectory;
var contents:Array = directory.listDirectory();
for (var i:uint = 0; i < contents.length; i++) {
    trace(contents[i].name, contents[i].size);
}
```

## Copying and moving directories

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:File = File.documentsDirectory.resolve("AIR Test");
var resultDir:File = File.documentsDirectory.resolve("AIR Test Copy");
sourceDir.copyTo(resultDir);
```

Note that when you specify `true` for the `clobber` 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).

For details, see [“Copying and moving files” on page 16](#).

## Deleting directory contents

The `File` class includes a `deleteDirectory()` method and a `deleteDirectoryAsync()` method. These both delete directories, the first working synchronously, the second working asynchronously (see [“AIR file basics” on page 7](#)). 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.

For example, the following code synchronously deletes the AIR Test subdirectory of the user's documents directory:

```
var directory:File = File.documentsDirectory.resolve("AIR Test");
directory.deleteDirectory(true);
```

The following code asynchronously deletes the AIR Test subdirectory of the user's documents directory:

```
var directory:File = File.documentsDirectory.resolve("AIR Test");
directory.addEventListener(Event.COMPLETE, completeHandler)
directory.deleteDirectoryAsync(true);

function completeHandler(event:Event):void {
    trace("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 18](#).



## Working with files

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, you should refer to [“Getting started with Adobe AIR” on page 5](#), 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 [“Registering file types” on page 24](#).

### Getting file information

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.)
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 Icon object containing the icons defined for the file.
isDirectory	Whether the File object reference is to a directory.
modificationDate	The date that the file on the local disk was last modified.
name	The name of the file on the local disk.
nativePath	The full path in the host operating system representation. See <a href="#">“Paths of File objects” on page 8</a> .
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 filesystem.
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 <a href="#">“Paths of File objects” on page 8</a> .

For details on these properties, see the File class entry in the *ActionScript 3.0 Language Reference*.

### Copying and moving files

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 7](#)).

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:File = File.documentsDirectory.resolve("AIR Test/test.txt");
var newFile:File = File.resolve("AIR Test/copy.txt");
original.copyTo(newFile, true);
```

Note that in this example, the value of `clobber` parameter of the `copyTo()` method (the second parameter) is set to `true`. By setting this to `true`, an existing target file is overwritten. This parameter is optional. If you set it to `true` (the default value), the operation dispatches an `IOErrorEvent` event if the target file already 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 = File.documentsDirectory;
original = original.resolve("AIR Test/test.txt");

var destination:File = File.documentsDirectory;
destination = destination.resolve("AIR Test 2/copy.txt");

original.addEventListener(Event.COMPLETE, fileMoveCompleteHandler);
original.addEventListener(IOErrorEvent.IO_ERROR, fileMoveIOErrorHandler);
original.moveToAsync(destination);

function fileMoveCompleteHandler(event:Event):void {
    trace(event.target); // [object File]
}
function fileMoveIOErrorHandler(event:IOErrorEvent):void {
    trace("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

The File class includes a `deleteFile()` method and a `deleteFileAsync()` method. These both delete files, the first working synchronously, the second working asynchronously (see [“AIR file basics” on page 7](#)).

For example, the following code synchronously deletes the test.txt file in the user's documents directory:

```
var directory:File = File.documentsDirectory.resolve("test.txt");
directory.deleteFile();
```

The following code asynchronously deletes the test.txt subdirectory of the user's documents directory:

```
var file:File = File.documentsDirectory.resolve("test.txt");
file.addEventListener(Event.COMPLETE, completeHandler)
file.deleteFileAsync();

function completeHandler(event:Event):void {
    trace("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 18](#).

## Moving a file to the trash

The `File` class includes a `moveToTrash()` method and a `moveToTrashAsync()` method. These both send a file or directory to the System trash, the first working synchronously, the second working asynchronously (see [“AIR file basics” on page 7](#)).

For example, the following code synchronously moves the `test.txt` file in the user's documents directory to the System trash:

```
var file:File = File.documentsDirectory.resolve("test.txt");
file.moveToTrash();
```

## Creating a temporary file

The `File` class includes a `createTempFile()` method, which creates a new file in the temporary directory folder for the System, as in the following example:

```
var temp:File = File.createTempFile();
```

The `createTempFile()` method automatically creates a unique temporary file (saving you the work of determining a new unique location).

You may 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 new, 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.

## Workflow for reading and writing files

The `FileStream` class lets AIR applications read and write to the file system. The workflow for reading and writing files is as follows.

### 1. Initialize a `File` object that points to the path.

This is the path of the file that you want to work with (or a file that you will later create).

```
var file:File = File.documentsDirectory;
file = file.resolve("AIR Test/testFile.txt");
```

This example uses the `File.documentsDirectory` property and the `resolve()` 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 10](#).

### 2. Initialize a `FileStream` object.

### 3. Call the `open()` method or the `openAsync()` method.

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 will be used to create a file and overwrite any existing data:

```
var fileStream:FileStream = new FileStream();  
fileStream.open(file, FileMode.WRITE);
```

For more information, see [“Initializing a FileStream object, and opening and closing files” on page 20](#) and [“FileStream open modes” on page 20](#).

#### **4. If you opened the file asynchronously (using the `openAsync()` method), add and set up event listeners for the `FileStream` object.**

These event listener methods will respond to events dispatched by the `FileStream` object in a variety of situations, such as 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 24](#).

#### **5. Include code for reading and writing data, as needed.**

There are a number of 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 25](#).

#### **6. Call the `close()` method of the `FileStream` object when you are done working with the file.**

This makes the file available to other applications.

For details, see [“Initializing a FileStream object, and opening and closing files” on page 20](#).

The sections that follow provide more details on using the AIR file APIs to read and write files.

For examples of using `FileStream` object to read and write files, see the following chapters:

- [“Building a text-file editor” on page 278](#)
- [“Reading and writing from an XML preferences file” on page 11](#)
- [“Building a directory search application” on page 15](#)

## **Working with `FileStream` objects**

This section contains the following topics:

- [FileStream open modes](#)
- [FileStream open modes](#)
- [The position property of a FileStream object](#)
- [The read buffer and the bytesAvailable property of a FileStream object](#)
- [Asynchronous programming and the events generated by a FileStream object opened asynchronously](#)
- [Data formats, and choosing the read and write methods to use](#)

### FileStream open modes

The `open()` and `openAsync()` method of a `FileStream` object each include a `fileMode` parameter, which defines a number of 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 table summarizes the various file modes (which you can specify as the `fileMode` parameter of the `open()` and `openAsync()` methods):

**FileMode.READ** Specifies that the file is open for reading only.

**FileMode.WRITE** Specifies that the file is open for writing. If the file does not exist, it is created when the `FileStream` object is opened. If the file does exist, any existing data is deleted.

**FileMode.APPEND** Specifies that the file is open for appending. The file is created if it does not exist. If the file already exists, existing data is not overwritten, and all writing begins at the end of the file.

**FileMode.UPDATE** 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, and 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

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:File = File.documentsDirectory.resolve("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.open(myFile, 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 20](#).

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:File = File.documentsDirectory.resolve("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.addEventListener(Event.COMPLETE, completeHandler);
myFileStream.addEventListener(ProgressEvent.PROGRESS, progressHandler);
myFileStream.addEventListener(IOErrorEvent.IOError, errorHandler);
myFileStream.open(myFile, FileMode.READ);

function completeHandler(event:Event):void {
    // ...
}

function progressHandler(event:ProgressEvent):void {
    // ...
}

function errorHandler(event:IOErrorEvent):void {
    // ...
}
```

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 7](#).

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 23](#).

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

The `position` property of a `FileStream` object determines where data will be read or written on the next read or write method.

Prior to 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:File = File.documentsDirectory.resolve("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.open(myFile, FileMode.UPDATE);
myFileStream.position = 8;
myFileStream.writeUTFBytes("hello");
```

When you first open a `FileStream` object, the `position` property is set to 0.

Prior to 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 will read or write. Depending on the read method you use, this will mean that 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 will read or write starting at the new position.

```
var myFile:File = File.documentsDirectory.resolve("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.open(myFile, FileMode.UPDATE);
myFileStream.position = 4000;
trace(myFileStream.position); // 4000
myFileStream.writeBytes(myByteArray, 0, 200);
trace(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.)

Note that 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 will execute them in order:

```
var myFile:File = File.documentsDirectory.resolve("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.openAsync(myFile, FileMode.WRITE);
myFileStream.writeUTFBytes("hello");
myFileStream.writeUTFBytes("world");
myFileStream.addEventListener(Event.CLOSE, closeHandler);
myFileStream.close();
trace("started.");

closeHandler(event:Event):void
{
    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:File = File.documentsDirectory.resolve("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.openAsync(myFile, FileMode.UPDATE);
myFileStream.position = 4000;
trace(myFileStream.position); // 4000
myFileStream.writeBytes(myByteArray, 0, 200);
myFileStream.position = 300;
trace(myFileStream.position); // 300
```

### The read buffer and the `bytesAvailable` property of a `FileStream` object

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:File = File.documentsDirectory.resolve("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.open(myFile, 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 signalled 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 prior to 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:ByteArray = new ByteArray();
var myFile:File = File.documentsDirectory.resolve("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.addEventListener(ProgressEvent.PROGRESS, progressHandler);
myFileStream.openAsync(myFile, FileMode.READ);

function progressHandler(event:ProgressEvent):void
{
    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 will need to ensure that the data exists in the read buffer prior to calling the read operation. You may do this by setting up a `progress` handler. For example, the following code reads 8000 bytes of data starting from position 4000 in the file:

```
var myFile:File = File.documentsDirectory.resolve("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.addEventListener(ProgressEvent.PROGRESS, progressHandler);
myFileStream.addEventListener(Event.COMPLETE, completed);
myFileStream.openAsync(myFile, FileMode.READ);
myFileStream.position = 4000;

var str:String = "";

function progressHandler(event:Event):void
{
    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**

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 will need to register for events generated by the `FileStream` object opened asynchronously.

By registering for the `progress` event, you can be notified as new becomes available for reading, as in the following code:

```
var myFile:File = File.documentsDirectory.resolve("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.addEventListener(ProgressEvent.PROGRESS, progressHandler);
myFileStream.openAsync(myFile, FileMode.READ);
var str:String = "";

function progressHandler(event:ProgressEvent):void
{
    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:File = File.documentsDirectory.resolve("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.addEventListener(Event.COMPLETE, completed);
myFileStream.openAsync(myFile, FileMode.READ);
var str:String = "";
function completeHandler(event:Event):void
{
    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 dispatched 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 `outputProgressEvent` 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

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:File = File.documentsDirectory.resolve("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.addEventListener(Event.COMPLETE, completed);
myFileStream.openAsync(myFile, FileMode.READ);

function completeHandler(event:Event):void
{
    myFileStream.readBytes(bytes, 0, myFileStream.bytesAvailable);
}
```

Similarly, the following code writes data from a `ByteArray` named `bytes` to a file:

```
var myFile:File = File.documentsDirectory.resolve("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.open(myFile, 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 will be 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:File = File.documentsDirectory.resolve("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.addEventListener(Event.COMPLETE, completed);
myFileStream.openAsync(myFile, FileMode.READ);
var str:String = "";

function completeHandler(event:Event):void
{
    str = myFileStream.readMultiByte(myFileStream.bytesAvailable, "iso-8859-1");
}
```

Note that the second parameter of the `readMultiByte()` method specifies the text format that ActionScript uses to interpret the data ("iso-8859-1" in the example). ActionScript supports a number of common character set encodings, and these are listed in the ActionScript Language Reference (see [Supported character sets](http://livedocs.macromedia.com/flex/2/langref/charset-codes.html) (<http://livedocs.macromedia.com/flex/2/langref/charset-codes.html>)).

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 a variable length, you should not try to 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, you should read the entire 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.

Note that 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 27](#).

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). This format is proprietary to ActionScript. Applications other than AIR, Flash Player, Flash Media Server, and Flex Data Services do not have built-in APIs for working with data in this format.

There are a number of 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 metatag 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 28](#) section.

Other files formats (for images, databases, application documents, etc.) have quite different structures, and to work with their data in `ActionScript`, you need to understand how the data is structured.

### Example: Reading an XML file into an XML object

This section shows how to read and write to a text file that contains XML data.

Reading from the file is easy. Simply initialize the `File` and `FileStream` objects, and set up an event listener for the complete event:

```
var file:File = File.documentsDirectory.resolve("AIR Test/preferences.xml");
var fileStream:FileStream = new FileStream();
fileStream.open(file, FileMode.READ);
var prefsXML:XML = XML(fileStream.readUTFBytes(fileStream.bytesAvailable));
fileStream.close();
```

Similarly, writing the data to the file is as easy as setting up an appropriate `File` and `FileStream` objects, and then calling a write method of the `FileStream` object, passing the string version of the XML data to the write method as in the following code:

```
var file:File = File.documentsDirectory.resolve("AIR Test/preferences.xml");
fileStream = new FileStream();
fileStream.open(file, FileMode.WRITE);

var outputString:String = '<?xml version="1.0" encoding="utf-8"?>\n';
outputString += prefsXML.toXMLString();

fileStream.writeUTFBytes(outputString);
fileStream.close();
```

Note that these examples use the `readUTFBytes()` and `writeUTFBytes()` methods, because they assume that the files are in UTF-8 format. If this is not the case, you may need to use a different method (see [“Data formats, and choosing the read and write methods to use” on page 25](#)).

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:File = File.documentsDirectory.resolve("AIR Test/preferences.xml");
var fileStream:FileStream = new FileStream();
fileStream.addEventListener(Event.COMPLETE, processXMLData);
fileStream.open(file, FileMode.READ);
var prefsXML:XML;

function processXMLData(event:Event):void
{
    prefsXML = XML(fileStream.readUTFBytes(fileStream.bytesAvailable));
    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” on page 11](#).

### Example: Reading and writing data with random access

MP3 files can include ID3 tags, which are sections at the beginning or end of the file that contain metadata identifying the recording. The ID3 tag format itself has gone through a number of revisions. This section describes how to read and write from an MP3 file that contains the simplest ID3 format: ID3 version 1.0. This example will not be reading and writing to the file sequentially from start to finish, and this is known as random access to file data.

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:File = File.documentsDirectory.resolve("My Music/Sample ID3 v1.mp3");
var fileStr:FileStream = new FileStream();
fileStr.open(file, FileMode.UPDATE);

var file = air.File.documentsDirectory.resolve("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;
```

We set the `position` 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 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():void
{
    if (fileStr.readMultiByte(3, "iso-8859-1").match(/tag/i))
    {
        var id3Title:String = fileStr.readMultiByte(30, "iso-8859-1");
        var id3Artist:String = fileStr.readMultiByte(30, "iso-8859-1");
        var id3Album:String = fileStr.readMultiByte(30, "iso-8859-1");
        var id3Year:String = fileStr.readMultiByte(4, "iso-8859-1");
        var id3Comment:String = fileStr.readMultiByte(30, "iso-8859-1");
        var id3GenreCode:String = fileStr.readByte().toString(10);
    }
}

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 with the code 0 (`String.fromCharCode(0)`).

# Chapter 14: Drag and Drop

Use the classes in the drag-and-drop API to support user-interface drag-and-drop gestures. With the drag-and-drop API, you can allow a user to drag data between your Adobe Integrated Runtime (AIR) application and other applications, as well as between components within your application. Supported transfer formats include:

- Bitmaps
- Files
- Text
- URL strings
- Serialized objects
- Object references (only valid within the originating application)

This chapter contains the following sections:

- [Drag and drop basics](#)
- [Supporting the drag-out gesture](#)
- [Supporting the drag-in gesture](#)
- [Transferable formats](#)

## Drag and drop basics

The drag-and-drop API contains the following classes.

Package	Classes
flash.desktop	DragManager, DragOptions, Transferable-Data  Constants used with the drag-and-drop API are defined in the following classes: DragActions, TransferableFormat, TransferableTransferModes
flash.events	NativeDragEvent

The drag-and-drop gesture has three stages:

**Initiation** A user initiates a drag-and-drop operation by dragging from a component, or an item in a component, while holding down the mouse button. If the drag is initiated from an AIR application, the component is typically designated as the drag initiator and will dispatch `nativeDragStart` and `nativeDragComplete` events for the drag operation. An AIR application starts a drag operation by calling the `DragManager.doDrag()` method in response to a `mouseDown` or `mouseMove` event.

**Dragging** While holding down the mouse button, the user moves the mouse cursor to another component, application, or to the desktop. AIR optionally displays a proxy image during the drag. When the user moves the mouse over a possible drop target in an AIR application, the drop target dispatches a `nativeDragEnter` event. The event handler can inspect the event object to determine whether the dragged data is available in a format that the target accepts and, if so, let the user drop the data onto it by calling the `DragManager.acceptDrop()` method.

**Drop** The user releases the mouse over an eligible drop target. If the target is an AIR application or component, then the component dispatches a `nativeDragDrop` event. The event handler can access the transferred data from the event object. If the target is outside AIR, the operating system handles the drop. In both cases, a `nativeDragComplete` event is dispatched by the initiating object.

The `DragManager` class controls both drag-in and drag-out gestures. All the members of the `DragManager` class are static, so an instance of this class does not need to be created.

### The TransferableData object

Data that is dragged into or out of an application or component is contained in a `TransferableData` object. (Copy-and-paste operations also use `TransferableData` objects.) A single `TransferableData` object can make available different representations of the same information to increase the likelihood that another application can 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 data is not actually created until it is read. A `TransferableData` object has a limited lifespan. Once a drag gesture has started, the `TransferableData` object can only be accessed from within an event handler for the `nativeDragEnter`, `nativeDragOver`, and `nativeDragDrop` events.

An application object can be transferred as a reference and as a serialized object. References are only valid within the originating application. Serialized object transfers are valid between AIR applications, but can only be used with objects that remain valid when serialized and deserialized.

### Working with the Flex framework

In most cases, it is better to use the Flex drag-and-drop API when building Flex applications. The Flex framework provides an equivalent feature set when a Flex application is run in AIR (it uses the AIR `DragManager` internally), while also maintaining a more limited feature set when an application or component is running within the more restrictive browser environment. AIR classes cannot be used in components or applications that run outside the AIR run-time environment.

## Supporting the drag-out gesture

To support the drag-out gesture, your application (or, more typically, a component of your application) must create a `TransferableData` object in response to a `mouseDown` event and send it to the `DragManager.doDrag()` method. Your application should then listen for the `nativeDragComplete` event on the initiating object to determine what action to take when the gesture is completed or abandoned by the user.

This section contains the following sections:

- [“Preparing data for transfer” on page 32](#)
- [“Starting a drag-out operation” on page 32](#)
- [“Completing a drag-out transfer” on page 34](#)
- [“Dragging Sprites” on page 34](#)



## Preparing data for transfer

To prepare data or an object for transfer out of a component or application, create a `TransferableData` object and add the information to be transferred in one or more formats. By supplying information in multiple formats, you increase the ability of other applications to use that information. You can use the standard data formats to pass data that can be translated automatically to native clipboard formats, and application-defined formats to pass objects. If it is computationally expensive to convert the information to be transferred into a particular format, you can supply the name of a handler function that will perform the conversion if and only if that format is read by the receiving component or application. For more information see “Transferable formats” on page 36.

The following example illustrates how to create a `TransferableData` object containing a bitmap in several formats: a `Bitmap` object, a native bitmap format, and a file list format containing the file from which the bitmap was originally loaded.

```
import flash.desktop.TransferableData;
import flash.display.Bitmap;
import flash.filesystem.File;

public function createTransferableData(image:Bitmap, sourceFile:File):TransferableData{
    var transfer:TransferableData = new TransferableData();
    transfer.addData("CUSTOM_BITMAP",image,true); //Flash object by value and by reference
    transfer.addData(TransferableFormats.BITMAP_FORMAT,image.bitmapData,false);
    transfer.addData(TransferableFormats.FILE_LIST_FORMAT,new Array(sourceFile),false);
    return transfer;
}
```

## Starting a drag-out operation

To start a drag operation, call the `DragManager.doDrag()` method in response to a mouse down event. `doDrag()` is a static method that takes the following parameters:

**initiator** The object from which the drag originates, and which will receive the `dragStart` and `dragComplete` events. The initiator must be a `DisplayObject`.

**transferable** The `TransferableData` object containing the data to be transferred. The `TransferableData` object is referenced in the `NativeDragEvent` objects.

**dragImage** (Optional) A `BitmapData` object to display during the drag. The image can specify an alpha value. (Note that Microsoft Windows always applies a fixed alpha fade to drag images).

**offset** (Optional) A `Point` object specifying the offset of the drag image from the mouse hotspot. Use negative coordinates to move the drag image up and left relative to the mouse cursor. If no offset is provided, the top, left corner of the drag image will be positioned at the mouse hotspot.

**actionsAllowed** (Optional) A `NativeDragOptions` object specifying which actions (copy, move, or link) are valid for drag operation. If no object is provided, all actions are permitted. The `DragOptions` object is referenced in `NativeDragEvent` objects to enable a potential drag target to check that the allowed actions are compatible with the target component's purpose. For example, a “trash” component might only accept drag gestures that allow the move action.

The following example illustrates how to start a drag operation for a bitmap object loaded from a file. The example loads an image and, on a `mouseDown` event, starts the drag operation.

```
import flash.desktop.DragManager;
import mx.core.UIComponent;
import flash.desktop.TransferableData;
import flash.display.Bitmap;
import flash.filesystem.File;

public class DragOutExample extends UIComponent{
    protected var fileURL:String = "app-resource:/image.jpg";
    protected var display:Bitmap;
    private function init():void{
        DragManager.init();
        loadImage();
    }
    private function onMouseDown(event:MouseEvent):void{
        var bitmapFile:File = new File(fileURL);
        var transferObject:TransferableData =
            createTransferableData(display,bitmapFile);
        DragManager.doDrag(this,transferObject,display.bitmapData,new Point(-mouseX,-
mouseY));
    }
    public function createTransferableData(image:Bitmap, sourceFile:File):TransferableData{
        var transfer:TransferableData = new TransferableData();
        transfer.addData(Bitmap,image,false); //Flash object by value and by reference
        transfer.addData("bitmap",image.bitmapData,false); //Standard bitmap format
        transfer.addData("file list",new Array(sourceFile),false); //Standard file list
format
        return transfer;
    }
    private function loadImage():void{
        var url:URLRequest = new URLRequest(fileURL);
        var loader:Loader = new Loader();
        loader.load(url,new LoaderContext());
        loader.contentLoaderInfo.addEventListener(Event.COMPLETE,onLoadComplete);
    }
    private function onLoadComplete(event:Event):void{
        display = event.target.loader.content;
        var flexWrapper:UIComponent = new UIComponent();
```

```
flexWrapper.addChild(event.target.loader.content);  
addChild(flexWrapper);  
flexWrapper.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);  
}  
}
```

## Completing a drag-out transfer

When a user drops the dragged item by releasing the mouse, a `nativeDragComplete` event is dispatched by the initiator object. If your application should modify the source data according to the drop action, you can check the `dropAction` property of the event object and then take the appropriate action. For example, if the action is “move,” you might remove the source item from its original location. If a drag gesture is abandoned by the user, the action will always be “none.”

When passing `DisplayObjects` by reference within an application, adding the transferred object as a child of a new parent will automatically remove the object from its original parent.

## Dragging Sprites

Sprite objects have a separate drag API which allows drag gestures to be used to move sprites around on the stage of a window. You can use both APIs at the same time to allow moving a sprite around its current stage as well as out of the stage. However, the `DragManager` API will take precedence visually — that is, the drag image or mouse pointer will be shown rather than the live, possibly animated, sprite.

# Supporting the drag-in gesture

To support the drag-in gesture, your application (or, more typically, a visual component of your application) must respond to `nativeDragEnter` or `nativeDragOver` events. The following sequence of events is typical for a drop operation:

- The user drags a transferable object over a component.
- The component dispatches a `nativeDragEnter` event.
- The `nativeDragEnter` event handler checks the formats available and, if relevant, the allowed actions. If the component can handle the drop, it calls `DragManager.acceptDragDrop()`.
- The `DragManager` changes the mouse cursor to indicate that the object can be dropped.
- The user drops the object over the component.
- A `nativeDragDrop` event is dispatched by the receiving component.
- The receiving component reads the data in the desired format from the `TransferableData` object within the event object.
- If the drag gesture originated within an AIR application, then a `nativeDragComplete` event is dispatched by the initiating display object. If the gesture originated outside AIR, no feedback is sent.

## Acknowledging a drag-in gesture

When a user drags a transferable item within the bounds of a visual component, the component dispatches native-`DragEnter` and native-`DragOver` events. To determine whether the transferable item can be dropped on the component, the handlers for these events should check the available formats in the `TransferableData` object referenced by the event transferable property. If desired, the component can also check the allowed actions by examining the `DragOptions` object referenced in the event. If the component can handle the drop, the event handler must call the `DragManager.acceptDragDrop()` method, passing a reference to the receiving component and the drag actions that the component supports. If more than one component responds to the drag-in gesture, the last component to respond takes precedence. The `acceptDragDrop()` call is valid until the mouse leaves the bounds of the accepting object, triggering the native-`DragExit` event (or until the drop occurs).

The following example illustrates an event handler for a native-`DragEnter` or native-`DragOver` event:

```
import flash.desktop.DragManager;
import flash.desktop.DragActions;
import flash.events.DragEvent;

public function onDragIn(event:DragEvent):void{
    if(event.transferable.hasFormat("text")){
        DragManager.setFeedback(DragActions.MOVE);
        DragManager.acceptDragDrop(this); //this is the receiving component
    }
}
```

If more than one drag action is specified, or no action is specified, the effective action reported back to the initiating object will follow precedence: copy, move, link. If none of the actions set by the receiving component matches an action allowed by the drag gesture, then the receiving component will not be eligible to take the drop.

## Completing the drop

When the user drops a transferable item on a component that has accepted the gesture, the component will dispatch a native-`DragDrop` event. The handler for this event can extract the data from the `TransferableObject` referenced by the transferable property of the event object.

When the transferable item contains an application-defined format, the transfer mode parameter determines whether the reference or the serialized version of the object within the data format is obtained.

The following example illustrates an event handler for the native-`DragDrop` event:

```
import flash.desktop.TransferableData;
import flash.events.DragEvent;

public function onDrop(event:DragEvent):void{
    if(event.transferable.hasFormat(TransferableFormat.TEXT_FORMAT)){
        var text:String =
            String(event.transferable.dataForFormat(TransferableFormat.TEXT_FORMAT));
    }
}
```

```
}
```

Once the event handler exits, the `TransferableData` object is no longer valid. Any attempt to access the object or its data will throw an exception.

## Updating the visual appearance of a component

A component can update its visual appearance based on the `NativeDragEvents`:

**`nativeDragStart`** The initiating component can use the `nativeDragStart` event to provide visual feedback that the drag gesture originated from that component.

**`nativeDragEnter`** A potential receiving component can use this event take the focus, or indicate visually that it can or cannot accept the drop.

**`nativeDragOver`** A potential receiving component can use this event to respond to the movement of the mouse within the component, such as when the mouse enters a “hot” region of a complex component like a map.

**`nativeDragExit`** A potential receiving component can use this event to restore its state when a drag gesture moves outside its bounds.

**`nativeDragComplete`** The initiating component can use this event to update its associated data model, such as by removing an item from a list, and to restore its visual state.

## Tracking mouse position during a drag-in gesture

Once a drag gesture has entered a component, `nativeDragOver` events are dispatched by that component. These events are dispatched whenever the mouse moves and also on a short interval. The `nativeDragOver` event object can be used to determine the position of the mouse over the component. This can be helpful in situations where the receiving component is complex, but is not made up of sub-components — such as a map.

# Transferable formats

Transferable formats describe the data placed in a `TransferableData` object. 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 AIR applications using application-defined formats.

A `TransferableData` object can contain representations of the same information in different formats. For example, a `TransferableData` object representing a `Sprite` could include a reference format for use within the same application, a serialized format for use by another AIR application, a bitmap representation for use by an image editor, and a file list format, perhaps with deferred rendering to encode a PNG file, for dragging a representation of the `Sprite` to the file system.

## Standard data formats

The constants defining the standard format names are provided in the `TransferableFormats` class:

**`TEXT_FORMAT = "text"`** Text-format data is translated to and from the ActionScript `String` class.

**`BITMAP_FORMAT = "bitmap"`** Bitmap-format data is translated to and from the ActionScript `BitmapData` class.

**`FILE_LIST_FORMAT = "file list"`** File-list-format data is translated to and from an array of ActionScript `File` objects.

`URL_FORMAT = "url"` URL-format data is translated to and from the `ActionScript String` class.

## Objects

Objects can be transferred as a reference or in a serialized format. References are only valid within the originating application. Serialized objects may be valid in other AIR applications as well.

To add a serialized object to a `TransferableData` object, set the `serializable` parameter to `true` when calling the `TransferableData.addData()` method. The format name can be one of the standard formats or an arbitrary string defined by your application.

```
public function createTransferableObject(object:Object):TransferableData{
    var transfer:TransferableData = new TransferableData();
    transfer.addData(object,"object", true);
}
```

To extract a serialized object from the transferable object (after a drop or paste operation), use the same format name and the `cloneOnly` or `clonePreferred` transfer modes.

```
var transfer:Object = transferable.dataForFormat("object",
TransferableTransferMode.CLONE_ONLY);
```

A reference is always added to the `TransferableData` object. To extract the reference from the transferable object (after a drop or paste operation), instead of the copy, use the `originalOnly` or `originalPreferred` transfer modes:

```
var transferedObject:Object = transferable.dataForFormat("object",
TransferableTransferMode.ORIGINAL_ONLY);
```

References are only valid if the `TransferableData` object originates from the current AIR application. Use the `originalPreferred` transfer mode to access the reference when it is available, and the `serialized clone` when the reference is not available.

## Deferred rendering

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 only called if a receiver of the drop or paste operation requests data in the deferred format.

The rendering function is added to a `TransferableData` object using the `addHandler()` method and must return the data in the expected format.

If a data format of the same type is added to a `TransferableData` object with the `addData()` method, that data will take precedence over the deferred version (the rendering function will never be called).

The following example illustrates implementing a function to provide deferred rendering.

When the Copy button in the example is pressed, the application creates a `TransferableData` object, adds the `renderData()` function, and writes the object to the clipboard.

When the Paste button is pressed, the application accesses the `TransferableData` object through the `ClipboardManager`. The first time a `TransferableData` object on the clipboard is accessed, the `ClipboardManager` will call the `renderData()` function. `renderData()` returns the text in the source `TextArea`, which is assigned to the destination `TextArea`. 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. 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 copy or protect the source data in some way to prevent this problem.)

```
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
  applicationComplete="init();" width="326" height="330">
  <mx:Script>
    <![CDATA[
      import flash.desktop.TransferableData;
      import flash.desktop.TransferableFormats;
      import flash.desktop.ClipboardManager;

      public function doCopy():void{
        var transfer:TransferableData = new TransferableData();
        transfer.addHandler(renderData,TransferableFormats.TEXT_FORMAT);
        ClipboardManager.accessClipboard(function():void{
          ClipboardManager.data = transfer;
        });
      }

      public function doPaste():void{
        ClipboardManager.accessClipboard(function():void{
          destination.text =
String(ClipboardManager.data.dataForFormat(TransferableFormats.TEXT_FORMAT));
        });
      }

      public function renderData():String{
        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="doPaste();" x="166" y="156" label="Paste"/>
```

```
<mx:Button click="doCopy();" x="91" y="156" label="Copy"/>  
</mx:WindowedApplication>
```



# Chapter 15: Copy and Paste

Use the classes in the copy-and-paste API to copy information to and from the system clipboard. With the copy-and-paste API, you can allow a user to copy data between your Adobe Integrated Runtime (AIR) application and other applications, as well as between components within your application. Supported transfer formats include:

- Bitmaps
- Files
- Text
- URL strings
- Serialized objects
- Object references (only valid within the originating application)

This chapter contains the following sections:

- [Copy-and-paste basics](#)
- [Reading from the clipboard](#)
- [Writing to the clipboard](#)
- [Transferable formats](#)

## Copy-and-paste basics

The copy-and-paste API contains the following classes.

Package	Classes
flash.desktop	TransferableData, ClipboardManager (Access the system clipboard through the ClipboardManager class. ClipboardManager is a static class, so an instance does not need to be created before it is used.)  Constants used with the copy-and-paste API are defined in the following classes: TransferableFormats, TransferableTransferModes

The data on the clipboard is represented by a TransferableData object. Different representations of the same information can be made available in a single TransferableData 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.

An application object can be transferred as a reference and as a serialized object. References are only valid within the originating application. Serialized object transfers are valid between Adobe AIR applications, but can only be used with objects that remain valid when serialized and deserialized.

The ClipboardManager maintains a single data object. Whenever the clipboard is written to (inside or outside of an AIR application), the previous contents are no longer accessible.

### See also

“Transferable formats” on page 36

## Reading from the clipboard

To read the clipboard, access the ClipboardManager.data property, which is a TransferableData object. The clipboard data property can only be accessed within a function passed to the ClipboardManager.accessClipboard property:

```
import flash.desktop.ClipboardManager;
import flash.desktop.TransferableData;
import flash.desktop.TransferableFormats;

public function readClipboard():void{
    ClipboardManager.accessClipboard(paste);
}

public function paste():void{
    var text:String =
        String(ClipboardManager.data.dataForFormat(TransferableFormats.TEXT_FORMAT));
}
```

You can also pass an anonymous function to the accessClipboard method:

```
ClipboardManager.accessClipboard(function():void{
    var text:String =
        String(ClipboardManager.data.dataForFormat(TransferableFormats.TEXT_FORMAT));
});
```

## Writing to the clipboard

To write to the clipboard, create a TransferableData object, add the data to it in one or more formats, and assign the object to the ClipboardManager.data property. Whether reading or writing, the clipboard data property can only be accessed within a function passed to the ClipboardManager.accessClipboard property:

```
import flash.desktop.ClipboardManager;
import flash.desktop.TransferableData;
import flash.desktop.TransferableFormats;

public function writeClipboard(textToCopy:String):void{
    var transferObject:TransferableData = new TransferableData();
    transferObject.addData(textToCopy, TransferableFormats.TEXT_FORMAT, false);
}
```

```
ClipboardManager.accessClipboard(function():void{  
    ClipboardManager.data = transferObject;  
});  
}
```

# Chapter 16: Adding HTML content to SWF-based applications

The Adobe Integrated Runtime lets you render complex HTML content in a SWF file, using the `HTMLControl` class.

The AIR `HTMLControl` class is built on WebKit technology (<http://webkit.org/>), used by the Apple Safari web browser, and has full support for all standard HTML tags, JavaScript, images, and CSS, allowing you to build complex applications using Ajax techniques .

This chapter contains the following sections:

- [About the `HTMLControl` class](#)
- [Loading HTML content from a URL](#)
- [Loading HTML content from a string](#)
- [Events dispatched by an `HTMLControl` object](#)
- [Accessing the HTML history list from `ActionScript`](#)
- [Accessing the HTML DOM and JavaScript in an HTML page](#)
- [Converting `JavaScriptObject` objects to the appropriate `ActionScript` type](#)
- [Manipulating an HTML stylesheet from `ActionScript`](#)
- [Registering `ActionScript` functions to respond to JavaScript events](#)
- [Responding to uncaught JavaScript exceptions in `ActionScript`](#)
- [Accessing the runtime classes and functions from JavaScript](#)
- [The `AIRAliases.js` file](#)
- [Making objects in the container display object available to JavaScript](#)
- [Defining browser-like user interfaces for HTML content](#)
- [Creating subclasses of the `HTMLControl` class](#)
- [Unsupported HTML and JavaScript functionality](#)

## About the `HTMLControl` class

The `HTMLControl` class of Adobe Integrated Runtime (AIR) defines a display object that can display HTML content in a SWF file.

For example, the following code loads a URL into an HTMLControl object and sets the object as a child of a Sprite object:

```
var container:Sprite;  
var html:HTMLControl = new HTMLControl;  
html.width = 400;  
html.height = 600;  
var urlReq:URLRequest = new URLRequest("http://www.adobe.com/");  
html.load(urlReq);  
container.addChild(html);
```

**Note:** In the Flex framework, only classes that extend the `UIComponent` class can be added as children of a Flex Container components. For this reason, you cannot directly add an `HTMLControl` as a child of a Flex Container component; however you can use the Flex HTML control, you can build a custom class that extends `UIComponent` and contains an `HTMLControl` as a child of the `UIComponent`, or you can add the `HTMLControl` as a child of a `UIComponent` and add the `UIComponent` to the Flex container.

You can also render HTML text by using the `TextField` class, but its capabilities are limited. The Flash Player's `TextField` class supports a subset of HTML markup, but because size limitations, its capabilities are limited. (The `HTMLControl` class included in Adobe AIR is not available in Flash Player.)

### See also

[“Using the Flex AIR components” on page 27](#)

## Loading HTML content from a URL

The `load()` method of an `HTMLControl` object loads content into the `HTMLControl` object from a URL defined by a `URLRequest` object:

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

### URLRequest objects in AIR

The `load()` method of an `HTMLControl` takes a `URLRequest` object as its single parameter (the `urlRequestToLoad` parameter). The following properties of the `URLRequest` are used:

- `data`
- `requestHeaders`
- `method`
- `url`

The corresponding static properties of the `URLRequestDefaults` class are also used (if the property is not set in the `URLRequest` object).

The username and password set with the `setLoginCredentials()` of the `URLRequest` object (and with the static `setLoginCredentialsForHost()` method of the `URLRequestDefaults` class) are also used.

Certain headers defined in the `requestHeader` property, such as the `Accept` header, are not used.

In defining the `url` property of a `URLRequest` object, use any of the following URL schemes:

**file** A path relative to the root of the file system. For example:

```
file:///c:/AIR Test/test.txt
```

**app-resource** A path relative to the root directory of the installed application (the directory that contains the application.xml file for the installed application). For example, the following path points to an images subdirectory of the directory of the installed application:

```
app-resource:/images
```

You can access the location of the app-resource directory using the `File.applicationResourceDirectory` property in ActionScript or the `air.File.applicationResourceDirectory` property in JavaScript.

**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. 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
```

You can access the location of the app-storage directory using the `File.applicationStorageDirectory` in ActionScript or the `air.File.applicationStorageDirectory` property in JavaScript.

**http** A standard HTTP request:

```
http://www.adobe.com
```

**https** A standard HTTPS request:

```
https://secure.example.com
```

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

In SWF content running in Flash Player (not in Adobe Integrated Runtime), only the `http:` and `https:` URL schemes are supported.

You can use a `URLRequest` object that uses any of these URL schemes to define the URL request for a number of different objects, such as an `HTMLControl` object, a `File` object, a `Loader` object, a `URLStream` object, or a `Sound` object.

### Loading a PDF document into an HTMLControl object

You can also use an `HTMLControl` object's `load()` method to load a PDF file into the object. However, if the user of an AIR application does not have the Acrobat Reader plug-in version 8.1 or later installed, PDF content will not display. You can obtain information on the user's Acrobat Reader version from the static `HTMLControl.pdfCapability` property. If the property evaluates to `PDFCapability.STATUS_OK`, then the application can load PDF content; otherwise you may want to display an error message.

For more information on using PDF content in AIR, see [“PDF” on page 97](#).

### Setting the user agent used when loading HTML content

The `HTMLControl` class has a `userAgent` property, which lets you set the the user agent string used by the `HTMLControl`. You must set the `userAgent` property of the `HTMLControl` object before calling the `load()` method. If you set this property, 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 `HTMLControl` objects in an application domain by setting the `URLRequestDefaults.userAgent` property. The static `URLRequestDefaults` properties apply as defaults for all `URLRequest` objects, not only those used by the `load()` method of `HTMLControl` objects. Setting the `userAgent` property of an `HTMLControl` overrides the default `URLRequestDefaults.userAgent` setting.

If a value is set for neither the `userAgent` property of the `HTMLControl` object nor for `URLRequestDefaults.userAgent`, a default value is used as the user agent string. 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) Safari/419.3 Apollo/1.0"
- "Mozilla/5.0 (Windows; U; en) AppleWebKit/420+ (KHTML, like Gecko) Safari/419.3 Apollo/1.0"

## Loading HTML content from a string

The `loadString()` method of an `HTMLControl` object loads a string of HTML content into the `HTMLControl` object:

```
var html:HTMLControl = new HTMLControl();
var htmlStr:String = "<html><body>Hello <b>world</b>.</body></html>";
html.loadString(htmlStr);
```

Consider using the XML class to represent large amounts of HTML data, and then using the `toXMLString()` method of the XML object to convert the data to a string:

```
var xHTML:XML =
    <html>
        <head>
            <style>
                tr { font-family: Verdana }
            </style>
            <script>
                function init() {
                    alert("Document loaded.");
                }
            </script>
        </head onload="init()">
        <body>
            <table>
                <tr>
                    <th>HTMLControl feature</th>
                    <th>Description</th>
                </tr>
                <tr>
                    <td>Full HTML tag support</td>
                    <td>Including tables, etc.</td>
                </tr>
                <tr>
                    <td>CSS support</td>
                    <td>As shown here</td>
                </tr>
                <tr>
                    <td>JavaScript support</td>
                    <td>Including cross-script communication to ActionScript.</td>
                </tr>
            </table>
        </body>
    </html>;

var htmlStr:String = xHTML.toXMLString();
var html:HTMLControl = new HTMLControl();
html.width = 400;
html.height = 200;
addChild(html);
html.loadString(htmlStr);
```

**Note:** When defining XML literals in a script tag in a Flex MXML file, use the `&#123;` character code in place of the `{` character. The MXML compiler does not correctly interpret a `{` character in this context.



## Display properties of HTMLControl objects

Because the `HTMLControl` class is a subclass of the `Sprite` class, you can set and get any of the properties in that class. For example, you can set the height, width, and position. You can also set properties, such as the `filters` array, that control more advanced visual effects:

```
var html:HTMLControl = new HTMLControl();
var urlReq:URLRequest = new URLRequest("http://www.adobe.com/");
html.load(urlReq);
html.width = 800;
html.height = 600;

var blur:BlurFilter = new BlurFilter(8);
var filters:Array = [blur];
html.filters = filters;
```

An `HTMLControl` object's `width` and `height` properties are both set to 0 by default. You will want to set these before adding an `HTMLControl` object to the stage.

### Transparency of HTMLControl content

The `paintsDefaultBackground` property of an `HTMLControl` object, which is `true` by default, determines whether the `HTMLControl` object uses an opaque background. With this property set to `false`, the `HTMLControl` object uses its display object container as a background for the HTML and it uses the opacity (alpha value) of the display object container as the HTML background.

However, if the body element or any other element of the HTML document has an opaque background color (specified by `style="background-color:gray"`, for instance), then that portion of the rendered HTML will use the specified opaque background color.

**Note:** *To you can use a transparent, png-format graphic to provide an alpha-blended background for an element in an HTML document.*

### Scaling HTMLControl content

You should not scale content in an `HTMLControl` object beyond a scale factor of 1.0. Text in `HTMLControl` content is rendered at a specific resolution, so it appears pixelated if scaled up. You may want to set the `scaleMode` property of the Stage to `StageScaleMode.NO_SCALE`.

### Considerations when loading PDF content

PDF content loaded into an `HTMLControl` object disappears in the following conditions:

- If you scale the `HTMLControl` object to a factor other than 1.0
- If you set the alpha property of the `HTMLControl` object to a value other than 1.0.
- If you rotate the `HTMLControl` content.

The PDF content reappears if you remove the offending property setting.

## Scrolling HTML content

The `HTMLControl` class includes the following properties that let you control the scrolling of HTML content:

**htmlHeight** The height, in pixels, of the HTML content.

**htmlWidth** The width, in pixels, of the HTML content.

**scrollH** The horizontal scroll position of the HTML content within the HTMLControl object.

**scrollV** The vertical scroll position of the HTML content within the HTMLControl object.

The HTMLControl does not include horizontal and vertical scroll bars. You can implement these in ActionScript or by using a Flex component. The Flex HTML component automatically includes scroll bars for HTML content.

## Events dispatched by an HTMLControl object

An HTML control dispatches the following events:

**domInitialize** Dispatched when the HTML document is created, but before any DOM nodes are actually created in the page.

**complete** Dispatched when the HTML DOM has been created in response to a load operation, immediately after the `onload` event in the HTML page.

**htmlBoundsChanged** Dispatched when one or both of the `htmlWidth` and `htmlHeight` properties have changed.

**locationChange** Dispatched when the location property of the HTMLControl has changed.

**scroll** Dispatched anytime the HTML engine changes the scroll position. This can be due to navigation to anchor links (# links) in the page or to JavaScript calls to the `window.scrollTo()` method. Typing in a text input or text area can also cause a scroll.

**uncaughtJavaScriptException** Dispatched when a JavaScript exception occurs in the HTMLControl and the exception is not caught in JavaScript code.

You can also register an ActionScript function for a JavaScript event (such as `onClick`). For details, see [“Registering ActionScript functions to respond to JavaScript events” on page 54](#).

## Accessing the HTML history list from ActionScript

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

**historyLength** The overall length of the history list, including back and forward entries.

**historyPosition** The current position in the history list. History items before this position represent “back” navigation, and items after this position represent “forward” navigation.

**historyAt()** Returns the `URLRequest` object corresponding to the history entry at the specified position in the history list.

**historyBack()** Navigates back in the history list, if possible.

**historyForward()** Navigates back in the history list, if possible.

`historyGo()` Navigates the indicated number of steps in the browser history. Navigates forward if positive, backward if negative. Navigation by zero forces a reload. Attempts to set position beyond the end will set it to the end.

## Accessing the HTML DOM and JavaScript in an HTML page

Once the HTML page is loaded, and the `HTMLControl` object dispatches the `complete` event, you can access the objects in the HTML DOM (document object model) for the page. This includes display elements (such as `div` and `p` objects in the page) as well as JavaScript variables and functions. If possible, wait for the `complete` event before accessing the HTML DOM.

For example, consider the following HTML page:

```
<html>
  <script>
    foo = 333;
    function test() {
      return "OK.";
    }
  </script>
  <body>
    <p id="p1">Hi.</p>
  </body>
</html>
```

This simple HTML page has a JavaScript variable named "foo" and a JavaScript function named "test()". Both of these are properties of the window object of the page (the page's DOM). Also, the window.document object includes a named P element (with the ID "p1"), which you can access using the getElementById() method. Once the page is loaded (when the HTMLControl object dispatches the complete event), you can access each of these from ActionScript, as shown in the following ActionScript code:

```
var html:HTMLControl = new HTMLControl();
html.width = 300;
html.height = 300;
html.addEventListener(Event.COMPLETE, completeHandler);
var xhtml:XML =
    <html>
        <script>
            foo = 333;
            function test() {
                return "OK.";
            }
        </script>
        <body>
            <p id="p1">Hi.</p>
        </body>
    </html>;
html.loadString(xhtml.toString());

function completeHandler(e:Event):void {
    trace(html.window.foo); // 333
    trace(html.window.document.getElementById("p1").innerHTML); // Hi.
    trace(html.window.test()); // OK.
}
```

To access the content of an HTML element, use the innerHTML property. For example, the previous code uses html.window.document.getElementById("p1").innerHTML to get the contents of the HTML element named "p1".

You can also set properties of the HTML page from ActionScript. For example, the following sets the contents of the p1 element on the page and it sets the value of the foo JavaScript variable on the page:

```
html.window.document.getElementById("p1").innerHTML = "eeee";
html.window.foo = 66;
```

## Converting JavaScriptObject objects to the appropriate ActionScript type

From ActionScript, all DOM objects are of the ActionScript type JavaScriptObject, except for JavaScript functions, which are of type JavaScriptFunction. The JavaScriptObject and JavaScriptFunction class are defined in the flash.html package. This is true for both objects declared in a JavaScript script and other DOM elements.

For example, consider the following HTML:

```
<html>
<script>
    var jsArray = ['foo', 'bar', 33];
    function insertIntoA(value) {
        jsArray = jsArray.splice(0, 0, value);
    };
</script>
<body>
    <p id='p1'>Buenos días.</p>
</body>
</html>
```

When you load this content into an `HTMLControl` object, the `a` property (defined in the JavaScript script) and the `p` element (of the document) are both of type `JavaScriptObject` (when accessed from `ActionScript`):

```
var html:HTMLControl = new HTMLControl();
var urlReq:URLRequest = new URLRequest("test.html");
html.load(urlReq);
html.addEventListener(Event.COMPLETE, htmlLoaded)
function htmlLoaded(e:Event):void {
    trace(html.window.jsArray is JavaScriptObject); // true
    trace(html.window.insertIntoA is JavaScriptFunction); // true
    trace(html.window.document.getElementById('p1') is JavaScriptObject); // true
}
```

Consider a JavaScript array, such as the `jsArray` JavaScript property in the example HTML. If you tried to assign its contents to an `ActionScript` object of type `Array`, using the following code, an error would be thrown:

```
var asArray:Array = html.window.jsArray;
```

The error results because the runtime cannot convert a `JavaScriptObject` to `Array` object. However, you can iterate through the members of the JavaScript object and pass them as properties of the `ActionScript` object:

```
var asArray:Array = new Array();
var jsArray = html.window.a;
for(var prop:Object in jsArray) {
    asArray[prop] = jsArray[prop];
}
trace(asArray[1]); // bar
```

## Manipulating an HTML stylesheet from ActionScript

Once the `HTMLControl` object has dispatched the `complete` event, you can examine and manipulate CSS styles in a page.

For example, consider the following simple HTML document:

```
<html>
<style>
  .style1A { font-family:Arial; font-size:12px }
  .style1B { font-family:Arial; font-size:24px }
</style>
<style>
  .style2 { font-family:Arial; font-size:12px }
</style>
<body>
  <p class="style1A">
    Style 1A
  </p>
  <p class="style1B">
    Style 1B
  </p>
  <p class="style2">
    Style 2
  </p>
</body>
</html>
```

After an `HTMLControl` object loads this content, you can manipulate the CSS styles in the page via the `cssRules` array of the `window.document.styleSheets` array, as shown here:

```
var html:HTMLControl = new HTMLControl( );
var urlReq:URLRequest = new URLRequest("test.html");
html.load(urlReq);
html.addEventListener(Event.COMPLETE, completeHandler);
function completeHandler(event:Event):void {
  var styleSheet0:Object = html.window.document.styleSheets[0];
  styleSheet0.cssRules[0].style.fontSize = "32px";
  styleSheet0.cssRules[1].style.color = "#FF0000";
  var styleSheet1:Object = html.window.document.styleSheets[1];
  styleSheet1.cssRules[0].style.color = "blue";
  styleSheet1.cssRules[0].style.font-family = "Monaco";
}
```

This code adjusts the CSS styles so that the resulting HTML document appears like the following:

**Style 1A**

**Style 1B**

Style 2

Keep in mind that code can add styles to the page after the `HTMLControl` object dispatches the `complete` event.

## Registering ActionScript functions to respond to JavaScript events

You can register ActionScript functions to respond to JavaScript events. For example, consider the following HTML content:

```
<html>
<body>
  <a href="#" id="testLink">Click me.</a>
</html>
```

You can register an ActionScript function as a handler for any event in the page. For example, the following code adds the `clickHandler()` function as the listener for the `onclick` event of the `testLink` element in the HTML page:

```
var html:HTMLControl = new HTMLControl( );
var urlReq:URLRequest = new URLRequest("test.html");
html.load(urlReq);
html.addEventListener(Event.COMPLETE, completeHandler);

function completeHandler(event:Event):void {
    html.window.document.getElementById("testLink").onclick = clickHandler;
}

function clickHandler():void {
    trace("You clicked it!");
}
```

It is good practice to wait for the HTMLControl to dispatches the `complete` event before adding these event listeners, as shown in the previous example. HTML pages often load multiple files and the HTML DOM is not fully built until all the files are loaded and parsed, at which point the HTMLControl to dispatches the `complete` event.

## Responding to uncaught JavaScript exceptions in ActionScript

Consider the following HTML:

```
<html>
<head>
  <script>
    function throwError() {
      var x = 400 * melbaToast;
    }
  </script>
</head>
<body>
  <a href="#" onclick="throwError()">Click me.</a>
</html>
```

It contains a JavaScript function, `throwError()`, that references an unknown variable, `melbaToast`:

```
var x = 400 * melbaToast;
```

When a JavaScript operation encounters an illegal operation that is not caught in the JavaScript code with a `try/catch` structure, the `HTMLControl` object dispatches an `HTMLUncaughtJavaScriptExceptionEvent` event. You can register an `ActionScript` method to listen for this event, as in the following code:

```
var html:HTMLControl = new HTMLControl();
var urlReq:URLRequest = new URLRequest("test.html");
html.load(urlReq);
html.width = container.width;
html.height = container.height;
container.addChild(html);
html.addEventListener(HTMLUncaughtJavaScriptExceptionEvent.UNCAUGHT_JAVASCRIPT_EXCEPTION,
    htmlErrorHandler);
function htmlErrorHandler(event:HTMLUncaughtJavaScriptExceptionEvent):void
{
    event.preventDefault();
    trace("exceptionValue:", event.exceptionValue)
    for (var i:uint = 0; i < event.stackTrace.length; i++)
    {
        trace("_____");
        trace("sourceURL:", event.stackTrace[i].sourceURL);
        trace("line:", event.stackTrace[i].line);
        trace("function:", event.stackTrace[i].functionName);
    }
}
```

In this example, the `htmlErrorHandler()` event handler cancels the default behavior of the event (which is to send the JavaScript error message to the trace output), and generates its own output message. It outputs the value of the `exceptionValue` of the `HTMLUncaughtJavaScriptExceptionEvent` object. It outputs the properties of each object in the `stackTrace` array:

```
exceptionValue: ReferenceError: Can't find variable: melbaToast

_____
sourceURL: app-resource:/test.html
line: 5
function: throwError

_____
sourceURL: app-resource:/test.html
line: 10
function: onclick
```



## Accessing the runtime classes and functions from JavaScript

JavaScript can call classes and methods in the Adobe Integrated Runtime. This is done using a special `runtime` property of the `window` object of the HTML page, which is available to any JavaScript code loaded in the runtime. However, only HTML pages in the application security domain (those HTML pages in the application resource directory, the directory in which the application is installed) have unrestricted access to the AIR runtime classes. HTML pages from other security domains will have access to the runtime classes, but using APIs that provide special functionality to AIR applications (APIs would not be available to SWF content running outside of the browser) will cause the application to dispatch a `SecurityErrorEvent`.

For example, JavaScript in an HTML page that is in the application security domain can use the AIR file system API to enumerate the contents of a directory on the user's computer:

```
var directory = window.runtime.filesystem.File.documentsDirectory;  
var contents = directory.listDirectory;
```

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

## The AIRAliases.js file

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

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

Included in the AIR SDK for HTML 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.Shell`
- `air.ClipboardManager`
- `air.FileStream`
- `air.SQLDatabase`

This is just a short subset of the classes in the `AIRAliases.js` file. You can open the `AIRAliases.js` file to see the full list.

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" />
```

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.

## Making objects in the container display object available to JavaScript

JavaScript in the HTML page loaded by an HTMLControl object can call the classes, objects, and functions of the parent display object, via the `window.htmlControl` object in the HTML page.

Code in the container SWF file must set properties of the `window` property of an HTMLControl to make properties and methods available to the loaded HTMLContent, as properties and methods of the JavaScript `window.htmlControl` object. The following ActionScript code and the HTML code that follows illustrate this:

```
var html:HTMLControl = new HTMLControl();
var foo:String = "Hello from container SWF."
function helloFromJS(message:String):void {
    trace("JavaScript says:", message);
}
var urlReq:URLRequest = new URLRequest("test.html");
html.addEventListener(Event.COMPLETE, loaded);
html.load(urlReq);

function loaded(e:Event):void
{
    html.window.foo = foo;
    html.window.helloFromJS = helloFromJS;
}
```

The HTML content (in a file named `test.html`) loaded into the HTMLControl in the previous example can access the `foo` property and the `helloFromJS()` method in the parent SWF file:

```
<html>
  <script>
    function alertFoo() {
      alert(foo);
    }
  </script>
  <body>
    <button onClick="alertFoo()">
      What is foo?
    </button>
    <p><button onClick="helloFromJS('Hi.')">
      Call helloFromJS() function.
    </button></p>
  </body>
</html>
```

To make ActionScript classes defined in a SWF file available in JavaScript, you need to make sure that the loaded HTML content is in the same application domain as the display object container, by setting the `useApplicationDomain` property of the HTMLControl object to `ApplicationDomain.currentDomain` (the application domain of the SWF content), as shown in the following code:

```
html.useApplicationDomain = ApplicationDomain.currentDomain;
```

The HTML content must be from a compatible security domain.

## Defining browser-like user interfaces for HTML content

The `HTMLControl` class defines no default user interface behavior for these settings, because you may not want changes to the `window.document.title` property, for example, in an `HTMLControl` object to set the title in the window that contains the application. For example, if the application presents multiple `HTMLControl` objects in a tabbed interface, you may want the title settings to apply to each tab, but not to the main window title. Similarly, your code could respond to a `window.moveTo()` call by repositioning the `HTMLControl` object in its parent display object container, by moving the window that contains the `HTMLControl` object, by doing nothing at all, or by doing something else entirely.

By default, an `HTMLControl` object does not make any changes based on JavaScript code that is intended to change the user interface. For example, the following properties in JavaScript have no effect, by default:

- `window.status`
- `window.document.title`
- `window.location`

Similarly, calling the following methods in JavaScript has no effect, by default:

- `window.blur()`
- `window.close()`
- `window.focus()`
- `window.moveBy()`
- `window.moveTo()`
- `window.open()`
- `window.resizeBy()`
- `window.resizeTo()`

The `HTMLHost` class lets you determine how to handle JavaScript that modifies these properties or calls these methods. The methods in the `HTMLHost` class correspond to each of these window settings.

- 1 Create a new 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):HTMLControl
        {
            var initOptions:NativeWindowInitOptions = new NativeWindowInitOptions();
            var window:NativeWindow = new NativeWindow(true, initOptions);
            window.stage.scaleMode = StageScaleMode.NO_SCALE;
            window.stage.addChild(htmlControl);
            return htmlControl;
        }
        override public function updateTitle(title:String):void
        {
            htmlControl.stage.window.title = title;
        }
    }
}
```

**3** In the code that contains the HTMLControl (not the code of the new subclass of HTMLHost) instantiate an object of the type defined by the new class, and assign that object to the htmlHost property of the HTMLControl. 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.HTMLControl;
            import CustomHost;
            private function init():void
            {
                var html:HTMLControl = new HTMLControl();
                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, in the application resource directory include an HTML file with the following content:

```
<html>
    <head>
        <title>Test</title>
    </head>
    <body>
        <a href="javascript:window.open('Test.html')">window.open('Test.html')</a>
    </body>
</html>
```

### Handling changes of the window.location property

To handle changes of the URL of the HTML page, requested when JavaScript in a page changes the value of window.location, override the locationChange() method of the class that extends the HTMLHost class, as shown in the following example:

```
override public function updateLocation(locationURL:String):void
{
    trace(locationURL);
}
```

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

To handle changes in the bounds of the HTML content, requested when JavaScript in a page calls these methods, override the `set windowRect()` method of the class that extends the `HTMLHost` class, as shown in the following example:

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

**Handling JavaScript calls to `window.open()`**

To handle JavaScript calls to `window.open()`, override the `createWindow()` method of the class that extends the `HTMLHost` class, as shown in the following example:

```
override public function createWindow(windowCreateOptions:HTMLWindowCreateOptions):void
{
    var initOptions:NativeWindowInitOptions = new NativeWindowInitOptions();
    var window:NativeWindow = new NativeWindow(true, initOptions);
    htmlControl.width = window.width;
    htmlControl.height = window.height;
    window.stage.scaleMode = StageScaleMode.NO_SCALE;
    window.stage.addChild(htmlControl);
    return htmlControl;
}
```

The object passed as a parameter to the `createWindow()` method is an `HTMLWindowCreateOptions` object. The `HTMLWindowCreateOptions` class includes properties that define values that are 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 <code>window.open()</code>
<code>fullscreen</code>	<code>fullscreen</code>
<code>height</code>	<code>height</code>
<code>locationBarVisible</code>	<code>location</code>
<code>menuBarVisible</code>	<code>menubar</code>
<code>resizeable</code>	<code>resizable</code>
<code>scrollBarsVisible</code>	<code>scrollbars</code>
<code>statusBarVisible</code>	<code>status</code>
<code>toolBarVisible</code>	<code>toolbar</code>
<code>width</code>	<code>width</code>
<code>x</code>	<code>left or screenX</code>
<code>y</code>	<code>top or screenY</code>

**Handling JavaScript calls to window.close()**

To handle JavaScript calls to `window.close()`, override the `closeWindow()` method of the class that extends the `HTMLHost` class, as shown in the following example:

```
override public function closeWindow():void
{
    htmlControl.stage.window.close();
}
```

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

```
override public function closeWindow():void
{
    htmlControl.parent.removeChild(htmlControl);
}
```

**Handling changes of the window.status property**

To handle JavaScript changes to the value of `window.status`, override the `updateStatus()` method of the class that extends the `HTMLHost` class, as shown in the following example:

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

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

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

To handle JavaScript changes to the value of `window.document.title`, override the `updateTitle()` method of the class that extends the `HTMLHost` class, as shown in the following example:

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

The requested title is passed as a string to the `updateTitle()` method.

You may not want the change to `document.title` to affect the title of the window containing the `HTMLControl` object. You may, for example, want the change to show up in another interface element, such as a text field.

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

To handle JavaScript calls to `window.blur()` and `window.focus()`, override the `windowBlur()` and `windowFocus()` methods of the class that extends the `HTMLHost` class, as shown in the following example:

```
override public function windowBlur():void
{
    htmlControl.alpha = 0.5;
}
override public function windowFocus():void
{
    htmlControl.alpha = 1;
}
```

## Creating subclasses of the HTMLControl class

You can create a subclass of the HTMLControl class, to create new behaviors. For example, you can create a subclass that defines default event listeners for HTMLControl events (such as those dispatched when HTML is rendered or when the user clicks a link).

For example, the following defines a subclass of the HTMLControl class:

```
package
{
    import flash.html.HTMLControl;
    import flash.filters.BlurFilter;

    public class FuzzyHTML extends HTMLControl
    {
        public function FuzzyHTML()
        {
            super();
            var filters:Array = new Array();
            var blur:BlurFilter = new BlurFilter(2);
            filters = [blur];
            this.filters = filters;
        }
    }
}
```

## Unsupported HTML and JavaScript functionality

AIR does not support the `Window.print()` method.

For the Beta 1 release, the HTMLControl class does not support the JavaScript `prompt()` function.

Also, for the Beta 1 release, certain fonts or effects (such as anti-aliasing) might not render correctly. The HTMLControl class is in development, and Adobe is aware that there will be some missing or incorrectly functioning features in the Beta 1 release.



# Chapter 17: Networking and communication

The new Adobe Integrated Runtime (AIR) functionality related to networking and communications is not available to SWF content running in the browser. This functionality is only available to content in the application security sandbox. For content in other sandboxes (such as content in a network sandbox, like the sandbox for `www.example.com`), calling any of these APIs will cause Adobe AIR to throw a `SecurityError` exception.

For other information on using ActionScript 3.0 networking and communications capabilities, see *Programming ActionScript 3.0*, delivered with both Adobe Flash CS3 and Adobe Flex Builder 2.0.

This chapter contains the following sections:

- [Using the URLRequest class](#)
- [Changes to the URLStream class](#)
- [Connecting to other AIR applications and to SWF content](#)
- [Scripting between content in different domains](#)
- [Opening a URL in the default system web browser](#)

This chapter describes AIR networking and communication API—functionality uniquely provided to applications running in the Adobe Integrated Runtime. It does not describe networking and communications functionality inherent to HTML and JavaScript that would function in a web browser (such as the capabilities provided by the `HTMLHttpRequest` class).

## Using the URLRequest class

The `URLRequest` class lets you define more than simply the URL string. AIR adds some new properties to the `URLRequest` class, which are not available to SWF content running in the browser.

### URLRequest properties in AIR

The `URLRequest` class defines the following properties:

**contentType** The MIME content type of any data sent with the URL request.

**data** An object containing data to be transmitted with the URL request.

**followRedirects** Specifies whether redirects are to be followed (`true`, the default value) or not (`false`). This is only supported in the runtime.

**manageCookies** Specifies whether the HTTP protocol stack should manage cookies (`true`, the default value) or not (`false`) for this request. This is only supported in the runtime.

**method** Controls the HTTP request method, such as a GET or POST operation. (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 [“Understanding AIR security” on page 10.](#))

**requestHeaders** The array of HTTP request headers to be appended to the HTTP request.

**shouldAuthenticate** Specifies whether authentication requests should be handled (`true`) for this request. This is only supported in the runtime. The default is to authenticate requests—this may cause an authentication dialog box to be displayed if the server requires credentials to be shown. You can also set the user name and password—see [“Setting login credentials for a URLRequest object” on page 65](#).

**shouldCacheResponse** Specifies whether successful response data should be cached for this request. This is only supported in the runtime. The default is to cache the response (`true`).

**useCache** Specifies whether the local cache should be consulted before this URLRequest fetches data. This is only supported in the runtime. The default (`true`) is to use the local cached version, if available.

**userAgent** Specifies the user-agent string to be used in the HTTP request.

Note that the URLRequest class in AIR includes support for the following properties, which are not supported in SWF content running in Flash Player:

- `followRedirects`
- `manageCookies`
- `shouldAuthenticate`
- `shouldCacheResponse`
- `useCache`

These properties, which define certain aspects of loading and caching content, are not available because the browser defines these aspects for SWF content running in the browser. For details of these objects, see the URLRequest class in the *AIR ActionScript 3.0 Language Reference*.

### Setting login credentials for a URLRequest object

The `setLoginCredentials()` method lets you set the user name and password to use for the URLRequest object, as illustrated in the following code:

```
var urlReq:URLRequest = new URLRequest();
urlReq.setLoginCredentials("www.example.com", "Babbage", "cs1791!!");
```

The URLRequestDefaults class lets you define default settings for URLRequest objects. For example, the following code sets the default values for the `manageCookies` and `useCache` properties:

```
URLRequestDefaults.manageCookies = false;
URLRequestDefaults.useCache = false;
```

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 you cannot change the authenticated user by calling the `setLoginCredentialsForHost()` method.

For example, the following code sets the default user name and password to use at `www.example.com`:

```
URLRequestDefaults.setLoginCredentialsForHost("www.example.com", "Ada", "love1816$X");
```

Each property of `URLRequestDefaults` settings apply to only the application domain of the content setting the property. However, the `setLoginCredentialsForHost()` method applies to content in all application domains within an AIR application. This way, an application can log into a host and have *all* content within the application be logged in with the specified credentials.

For more information, see the `URLRequestDefaults` class in the *Flex Language Reference for Apollo*.

### Using AIR URL schemes in URLRequest object URLs

You can also use the following schemes when defining a URL for a `URLRequest` object:

**http:** and **https:** Use these as you would use them in a web browser.

**file:** Use this to specify a path relative to the root of the file system. For example:

```
file:///c:/AIR_Test/test.txt
```

**app-resource:** Use this to specify a path relative to the root directory of the installed application (the directory that contains the application descriptor file for the installed application). For example, the following path points to a resources subdirectory of the directory of the installed application:

```
app-resource:/resources
```

When running in the ADL application, the application resource directory is set to the directory that contains the application descriptor file.

**app-storage:/** Use this 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 `prefs.xml` file in a settings subdirectory of the application store directory:

```
app-storage:/settings/prefs.xml
```

You can use a `URLRequest` object that uses any of these URL 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 URL schemes in content in the application resource directory. For more information, see [“About AIR security” on page 9](#).

## Changes to the URLStream class

The `URLStream` class provides low-level access to downloading data from URLs. In the runtime, the `URLStream` class includes a new event: `httpResponseStatus`. Unlike the `httpStatus` event, the `httpResponseStatus` event is delivered before any response data. The `httpResponseStatus` event (defined in 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.

## Connecting to other AIR applications and to SWF content

The `LocalConnection` class enables communications between AIR applications, as well as among AIR applications and SWF content running in the browser.

The `connect()` method of the `LocalConnection` class uses a `connectionName` parameter to identify applications. In content running in the AIR application security sandbox (content installed with the AIR application), AIR uses the string `app#` followed by the application ID for the AIR application (defined in the application descriptor file) in place of the domain used by SWF content running in the browser. For example a `connectionName` for an application with the application ID `com.example.air.MyApp`, the `connectionName` resolves to `"app#com.example.air.MyApp:connectionName"`.

## Scripting between content in different domains

Content from a remote domain does not have access to scripts in the main AIR application. The application could grant access to a remote SWF file using the `Security.allowDomain()` method, but this is not a secure practice since the remote file would then have access to read any public method or variable within the main application. Instead, the `Door` class should be used as a gateway between the two, providing explicit interaction between remote and application security sandboxes.

Suppose an AIR music store application wants to allow remote SWF files to broadcast the price of albums, but does not want the remote SWF file to disclose whether the price is a sale price. To do this, a `StoreAPI` class that subclasses `Door` provides a method to acquire the price, but obscures the sale price. An instance of this `StoreAPI` class is then assigned to the `parentDoor` property of the `LoaderInfo` object of the `Loader` object that loads the remote SWF.

The following is the code for the AIR music store:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:AIRApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute" title="Hello
World" creationComplete="initApp()">
  <mx:Script>
    import flash.display.Loader;
    import flash.net.URLRequest;
    import flash.system.Door;

    private var child:Loader;
    private var isSale:Boolean = false;

    private function initApp():void {
      var url:String = "http://[www.yourdomain.com]/PriceQuoter.swf";
      var request:URLRequest = new URLRequest(url)

      child = new Loader();
      child.contentLoaderInfo.parentDoor = new StoreAPI(this);
      child.load(request);
      container.addChild(child);
    }
    public function getRegularAlbumPrice():String {
      return "$11.99";
    }
    public function getSaleAlbumPrice():String {
      return "$9.99";
    }
    public function getAlbumPrice():String {
      if(isSale) {
        return getSaleAlbumPrice();
      }
      else {
        return getRegularAlbumPrice();
      }
    }
  </mx:Script>
  <mx:UIComponent id="container" />
</mx:AIRApplication>
```

The following code provides the gateway that allows remote SWF files scripting access into the main application:

```
package
{
  import flash.system.Door;

  public class StoreAPI extends Door
  {
    private var mainApp:*;

    public function StoreAPI(mainApp:*) {
      this.mainApp = mainApp;
    }
    public function getAlbumPrice():String {
      return this.mainApp.getAlbumPrice();
    }
  }
}
```

```

        public function getSaleAlbumPrice():String {
            return "Unknown";
        }
    }
}

```

The following code represents an example of a PriceQuoter SWF file that reports the store's price, but cannot report the sale price:

```

package
{
    import flash.display.Sprite;
    import flash.system.Security;
    import flash.text.*;

    public class PriceQuoter extends Sprite
    {
        private var storeRequester:Object;

        public function PriceQuoter() {
            trace("Initializing child SWF");
            trace("Child sandbox: " + Security.sandboxType);
            storeRequester = loaderInfo.parentDoor;

            var tf:TextField = new TextField();
            tf.autoSize = TextFieldAutoSize.LEFT;
            addChild(tf);

            tf.appendText("Store price of album is: " + storeRequester.getAlbumPrice());
            tf.appendText("\n");
            tf.appendText("Sale price of album is: " + storeRequester.getSaleAlbumPrice());
        }
    }
}

```

## Opening a URL in the default system web browser

You can use `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.

# Chapter 18: Native menus

The classes in the Native Menu API to define application, window, and context menus. The Beta 1 release of Adobe® Integrated Runtime (AIR™) only supports using this class for application menus on Mac OS.

## Working with native menus

[Defining a menu](#)

[Defining menu items, submenus, and separators](#)

[Assigning a keyboard equivalent](#)

[Assigning a mnemonic](#)

[Handling menu events](#)

### Defining a menu

Each Adobe AIR application is initialized with a `NativeMenu` instance that is prepopulated with a default set of menu items that correspond to basic commands of the operating system. You can access the default `NativeMenu` instance through the `shell` singleton instance assigned to the `Shell` class.

The following code assigns this instance to a variable:

```
var menu:NativeMenu = Shell.shell.menu;
```

This `menu` variable contains an `items` property which is an array of `NativeMenuItem` instances. These instances make up the content of your application's menu. Default menu items can be removed using the `removeItem()` method. Additional commands to support the function of your AIR application can be added using the `addItem()` method. The order of menu items follows order of creation. Each instance of the `NativeMenuItem` class has several properties that define the characteristics of your menu item.

### Defining menu items, submenus, and separators

An application menu can be more than just a list of commands. Applications commonly employ hierarchy by grouping items into submenus. You create a submenu as a new instance of `NativeMenu` and assigning it to the `submenu` property of an existing `NativeMenuItem`.

For example, the following code creates a submenu called `Change Color` with color choices as menu items and adds it to an AIR application's application menu:

```
var submenu:NativeMenu = new NativeMenu();
submenu.addItem(new NativeMenuItem("Red"));
submenu.addItem(new NativeMenuItem("Green"));
submenu.addItem(new NativeMenuItem("Blue"));
var menuItem:NativeMenuItem = new NativeMenuItem("Change Color");
menuItem.submenu = submenu
Shell.shell.menu.addItem(menuItem);
```

Use separators to further cue the relationship between menu items. The constructor for `NativeMenuItem` accepts an optional second parameter. When `true`, the menu item is displayed as a separator.

The following code adds a Reset Color option to the Change Color submenu, but separates it from the other items to signal its distinct behavior to the user:

```
var submenu:NativeMenu = new NativeMenu();
submenu.addItem(new NativeMenuItem("Red"));
submenu.addItem(new NativeMenuItem("Green"));
submenu.addItem(new NativeMenuItem("Blue"));
submenu.addItem(new NativeMenuItem("",true)); // set isSeparator parameter to true
submenu.addItem(new NativeMenuItem("Reset Color"));
var menuItem:NativeMenuItem = new NativeMenuItem("Change Color");
menuItem.submenu = submenu
Shell.shell.menu.addItem(menuItem);
```

## Assigning a keyboard equivalent

To streamline workflow for users who use menu commands frequently, applications provide keyboard shortcuts so that users can run menu commands without necessitating the use of the mouse. Typically, these shortcuts are activated when the user presses a modifier key (Shift, Control, etc.) in combination with another key.

Any menu item in an AIR application can be assigned a shortcut key, with or without a modifier key. You assign shortcuts by specifying the key as a one-character string to the `keyEquivalent` property of the menu item. To limit the shortcut so that it only activates when used in combination with a modifier key, pass an array containing the key codes of the required modifier key or keys. When assigned, keyboard shortcuts display beside the menu item name automatically, though its exact visual representation depends on the user's operating system and system preferences. The `Keyboard` class contains public constants for commonly used modifiers and is recommended.

The following code sets keyboard shortcuts for each of the colors in the Change Color menu illustrated earlier:

```
var submenu:NativeMenu = new NativeMenu();
var red:NativeMenuItem = new NativeMenuItem("Red");
red.keyEquivalent = "R";
red.keyEquivalentModifiers = [ Keyboard.CONTROL ];
var green:NativeMenuItem = new NativeMenuItem("Green");
green.keyEquivalent = "G";
green.keyEquivalentModifiers = [ Keyboard.CONTROL ];
var blue:NativeMenuItem = new NativeMenuItem("Blue");
blue.keyEquivalent = "B";
blue.keyEquivalentModifiers = [ Keyboard.CONTROL ];
var menuItem:NativeMenuItem = new NativeMenuItem("Change Color");
submenu.addItem(red);
submenu.addItem(green);
submenu.addItem(blue);
menuItem.submenu = submenu;
Shell.shell.menu.addItem(menuItem);
```

## Assigning a mnemonic

Mnemonics allow a user to “type through” your menu system. They are similar but distinct from keyboard shortcuts. Mnemonics only function when a user has focus on a menu, such as clicking to open a submenu. Behavior differs depending on the operating system. On Windows, a menu item's mnemonic is represented by an underlined letter in the name of the item. Pressing this key will activate the command. Mac OS X applications typically do not use single-key mnemonics; instead, they highlight a menu item as a user types it, at which point the user presses enter to activate it. For this reason, AIR applications running on OS X follow operating system conventions and do not support mnemonics.



You can assign a mnemonic to any instance of a `NativeMenuItem` by assigning the index that corresponds to the letter you wish to use as a mnemonic to the `mnemonicIndex` property.

## Handling menu events

When a menu item is clicked, an event is dispatched. An AIR application responds to a user's selection by adding listeners to `NativeMenuItem` instances.

The following code traces the name of the menu item that was activated:

```
var submenu:NativeMenu = new NativeMenu();
var red:NativeMenuItem = new NativeMenuItem("Red");
red.addEventListener(Event.SELECT, announceSelection)
var green:NativeMenuItem = new NativeMenuItem("Green");
green.addEventListener(Event.SELECT, announceSelection)
var blue:NativeMenuItem = new NativeMenuItem("Blue");
blue.addEventListener(Event.SELECT, announceSelection)
var menuItem:NativeMenuItem = new NativeMenuItem("Change Color");
submenu.addItem(red);
submenu.addItem(green);
submenu.addItem(blue);
menuItem.submenu = submenu;
Shell.shell.menu.addItem(menuItem);

function announceSelection(e:Event):void {
    var menuItem:NativeMenuItem = e.target as NativeMenuItem;
    trace(menuItem.label + " has been selected")
}
```

# Chapter 19: Monitoring Network Connectivity

Your Adobe Integrated Runtime (AIR) application can run in environments with uncertain and changing network connectivity. To help an application manage connections to online resources, Adobe AIR sends a network change event whenever a network connection becomes available or unavailable. The network change event is dispatched by the application's `flash.system.Shell` object. To react to this event, add a listener:

```
Shell.shell.addEventListener(Event.NETWORK_CHANGE, onNetworkChange);
```

And define an event handler function:

```
function onNetworkChange(event:Event) {  
    //Check resource availability
```

The `Event.NETWORK_CHANGE` event does not indicate a change in all network activity, only that a network connection has changed. AIR does not attempt to interpret the meaning of the network change. A networked computer may have many real and virtual connections, so losing a connection does not necessarily mean losing a resource. On the other hand, new connections do not guarantee improved resource availability, either. Sometimes a new connection can even block access to resources previously available (for example, when connecting to a VPN).

In general, the only way for an application to determine whether it can connect to a remote resource is to try it. To this end, the service monitoring frameworks in the `air.net` package provide AIR applications with an event-based means of responding to changes in network connectivity to a specified host.

**Note:** *The service monitoring framework detects whether a server responds acceptably to a request. This does not guarantee full connectivity. Scalable web services often use caching and load-balancing appliances to redirect traffic to a cluster of web servers. In this situation, service providers only provide a partial diagnosis of network connectivity.*

## Service monitoring basics

The service monitor framework, separate from the AIR framework, resides in the file `servicemonitor.swc`. This file must be included in your build process in order to use the framework. Flex Builder includes this automatically.

The `ServiceMonitor` class implements the framework for monitoring network services and provides a base functionality for service monitors. By default, an instance of the `ServiceMonitor` class will dispatch events regarding network connectivity. These events will be dispatched, when the instance is created and whenever a network change is detected by the Adobe Integrated Runtime (AIR). Additionally, you can set the `pollInterval` property of a `ServiceMonitor` instance to check connectivity at a specified interval in milliseconds, regardless of general network connectivity events. An instance of `ServiceMonitor` will not check network connectivity until the `start()` method is called.

The `URLMonitor` class, subclassing the `ServiceMonitor` class, detects changes in HTTP connectivity for a specified `URLRequest`.

The `SocketMonitor` class, also subclassing the `ServiceMonitor` class, detects changes in connectivity to a specified host at a specified port.

## Detecting HTTP connectivity

The `URLMonitor` class determines if HTTP requests can be made to a specified address at port 80 (the typical port for HTTP communication). The following code uses an instance of the `URLMonitor` class to detect connectivity changes to the Adobe website:

```
import air.net.URLMonitor;
import flash.net.URLRequest;
import flash.events.StatusEvent;

var monitor:URLMonitor;
monitor = new URLMonitor(new URLRequest('http://www.adobe.com'));
monitor.addEventListener(StatusEvent.STATUS, announceStatus);
monitor.start();

function announceStatus(e:StatusEvent):void {
    trace("Status change. Current status: " + monitor.available);
}
```

## Detecting Socket connectivity

AIR applications can also use socket connections for push-model connectivity. Firewalls and network routers typically restrict network communication on unauthorized ports for security reasons. For this reason, developers must consider that users may not have the capability of making socket connections.

Similar to the `URLMonitor` example, the following code uses an instance of the `SocketMonitor` class to detect connectivity changes to a socket connection at 6667, a common port for IRC.

```
import air.net.ServiceMonitor;
import flash.events.StatusEvent;

socketMonitor = new SocketMonitor('www.adobe.com', 80);
socketMonitor.addEventListener(StatusEvent.STATUS, socketStatusChange);
socketMonitor.start();

function announceStatus(e:StatusEvent):void {
    trace("Status change. Current status: " + socketMonitor.available);
}
```

# Chapter 20: Updating applications programmatically

Users can install or update an AIR application by double-clicking an AIR file on their computer, and the AIR installer application will manage the installation, alerting the user if they are updating an already existing application.

However, you can also have an installed application update itself to a new version, using the Updater class. The Updater class includes an `update()` method that lets you point to an AIR file on the user's computer and update to that version.

This chapter contains information on using the Updater class. The chapter contains the following sections:

- [About updating applications](#)
- [Presenting a custom application update user interface](#)
- [Downloading an AIR file to the user's computer](#)
- [Checking to see if an application is running for the first time](#)

## About updating applications

The Updater class (in the `flash.system` package) includes one method, `update()`, which you can use to update the currently running application with a different version. For example, if the user has a version of the AIR file ("Sample\_App\_v2.air") located on the desktop, the following code updates the application:

```
var updater:Updater = new Updater();  
var airFile:File = File.desktopDirectory.resolve("Sample_App_v2.air");  
var version:String = "2.01";  
updater.update(airFile, version);
```

### Results of the method call

When an application in the runtime calls the `update()` method, the runtime closes the application, and it then attempts to install the new version from the AIR file. The runtime checks that the application ID specified in the AIR file matches the application ID for the application calling the `update()` method. It also checks that the version string matches the version string passed to the `update()` method. If installation completes successfully, the runtime opens the new version of the application. Otherwise (if the installation cannot complete), it re-opens the existing (pre-install) version of the application.

When testing an application using ADL, calling the `update()` method installs and runs a new version of the application only if the runtime is installed. If the runtime is *not* installed, the call to the `update()` results in a runtime exception.

### About the version string

The string that is specified as the `version` parameter of the `update()` method must match the string in the `version` attribute of the main `application` element of the application descriptor file for the AIR file to be installed. Specifying the `version` parameter is required for security reasons. By requiring the application to verify the version number in the AIR file, the application will not inadvertently install an older version, which might contain a security vulnerability that has been fixed in the currently installed application.

The version string can be of any format. For instance, it can be "2.01" or "version 2". The format of this string is left for you, the application developer, to decide.

If an Adobe Integrated Runtime (AIR) application downloads an AIR file via the web, it is a good practice to have a mechanism by which the web service can notify the Adobe AIR application of the version being downloaded. The application can then use this string as the `version` parameter of the `update()` method. If the AIR file is obtained by some other means, in which the version of the AIR file is unknown, the AIR application can examine the AIR file to determine the version information. (An AIR file is a ZIP-compressed archive, and the application descriptor file is the third record in the archive.)

For details on the application descriptor file, see [“Setting application properties” on page 42](#).

## Presenting a custom application update user interface

AIR includes a default update interface:



This interface is always used the first time a user installs a version of an application on a machine. However, you can define your own interface to use for subsequent instances. To do this specify a `handleUpdates` element in the application descriptor file for the application:

```
</handleUpdates/>
```

When the application is installed, when the user opens an AIR file with an application ID matching the installed application, the runtime opens the application, rather than the default AIR application installer. For more information, see [“Signaling the inclusion of an update interface” on page 46](#).

The application can decide, when it is invoked (when the `Shell.shell` object dispatches an `invoke` event) whether to update the application (using the `Updater` class). If it decides to update, it can present its own installation interface (which differs from its standard running interface) to the user.

## Downloading an AIR file to the user's computer

To use the `Updater` class, you must first save an AIR file locally to the user's machine. For example, the following code reads an AIR file from a URL (`http://example.com/air/updates/Sample_App_v2.air`) and saves the AIR file to the desktop:

```
var urlString:String = "http://example.com/air/updates/Sample_App_v2.air";
var urlReq:URLRequest = new URLRequest(urlString);
var urlStream:URLStream = new URLStream();
var fileData:ByteArray = new ByteArray();
urlStream.addEventListener(Event.COMPLETE, loaded);
urlStream.load(urlReq);

function loaded(event:Event):void {
    urlStream.readBytes(fileData, 0, urlStream.bytesAvailable);
    writeAirFile();
}

function writeAirFile():void {
    var file:File = File.desktopDirectory.resolve("My App v2.air");
    var fileStream:FileStream = new FileStream();
    fileStream.addEventListener(Event.CLOSE, fileClosed);
    fileStream.openAsync(file, FileMode.WRITE);
    fileStream.writeBytes(fileData, 0, fileData.length);
    fileStream.close();
}

function fileClosed(event:Event):void {
    trace("The AIR file is written.");
}
```

### See also

[“Workflow for reading and writing files” on page 18.](#)

## Checking to see if an application is running for the first time

Once you have updated an application you may want to provide the user with "getting started" or "welcome" message. To do this, you will want the application, upon launching, to check to see it is running for the first time, so that you can determine whether to display the message upon.

One way to do this is to save a file to the application store directory upon initializing the application (for example, when a top-level Flex component dispatches the `applicationComplete` event). Every time the application starts up, it should check for the existence of that file. If the file does not exist, then the application is running for the first time. If the file exists, the application has already run at least once. If the file exists and contains a version number older than the current version number, then you know the user is running the new version for the first time.

Here is a Flex example:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical"
    title="Sample Version Checker Application"
    applicationComplete="init()">
    <mx:Script>
        <![CDATA[
            import flash.filesystem.*;
            public var file:File;
            public var currentVersion:String = "1.2";
            public function init():void {
                file = File.applicationStorageDirectory.resolve("Preferences/version.txt");
                trace(file.nativePath);
                if(file.exists) {
                    checkVersion();
                } else {
                    firstRun();
                }
            }
            private function checkVersion():void {
                var stream:FileStream = new FileStream();
                stream.open(file, FileMode.READ);
                var prevVersion:String = stream.readUTFBytes(stream.bytesAvailable);
                stream.close();
                if (prevVersion != currentVersion) {
                    log.text = "You have updated to version " + currentVersion + ".\n";
                } else {
                    saveFile();
                }
                log.text += "Welcome to the application.";
            }
            private function firstRun():void {
                log.text = "Thank you for installing the application. \n"
                    + "This is the first time you have run it.";
                saveFile();
            }
            private function saveFile():void {
                var stream:FileStream = new FileStream();
                stream.open(file, FileMode.WRITE);
                stream.writeUTFBytes(currentVersion);
                stream.close();
            }
        ]]>
    </mx:Script>
    <mx:TextArea id="log" width="100%" height="100%" />
</mx:WindowedApplication>
```

# Chapter 21: Detecting AIR capabilities

For a file that is bundled with the Adobe Integrated Runtime (AIR) application, the `Security.sandboxType` property is set to the value defined by the `Security.APPLICATION` constant. So, you can check to see if a file is in the Adobe AIR security sandbox, and load other content (which may or may not contain APIs specific to AIR) accordingly, as in the following:

```
if (Security.sandboxType == Security.APPLICATION)
{
    // Load SWF that contains AIR APIs
}
else
{
    // Load SWF that does not contain AIR APIs
}
```

All resources that are not installed with the AIR application are put in the same security sandboxes as they would be placed in if they were running in Flash Player in a web browser. Remote resources are put in sandboxes according to their source domains, and local resources are put in the local-with-networking, local-with-filesystem, or local-trusted sandbox.

Although content outside of the AIR application security sandbox may access AIR runtime APIs, but some functionality (such as attempting to read files from the filesystem) will result in a runtime security exception.

You can check if the `Capabilities.playerType` static property is set to `Security.APPLICATION` to see if content is in the runtime (and not running in Flash Player running in a browser).

For more information about AIR security, see [“Understanding AIR security” on page 10](#)



# Chapter 22: Working with local SQL databases

Adobe Integrated Runtime (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. A local SQL database can be used for storing local, persistent data, which can be application data, application user settings, documents, or any other type of data that you might want your application to save locally.

This chapter contains the following sections:

- [About local SQL databases](#)
- [Common SQL database tasks](#)
- [Creating and modifying a database](#)
- [Strategies for working with SQL databases](#)

## About local SQL databases

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, if you have experience with SQL and relational databases, the runtime's local SQL database engine provides a convenient mechanism for storing persistent, local application data. However, because the data is local to a single computer, data is not automatically shared among users on different computers, and the local SQL database engine doesn't provide any capability to execute SQL statements against a remote or server-based database.

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 any AIR application would be able to open the database file, so a separate encryption mechanism would be 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.

## About AIR databases and database files

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 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). Because the runtime stores each database as a single file on the file system, a database file can be accessed by all application users on a single computer, or each user can have a separate database file, according to the design of the application and the constraints on file access specified by the operating system.

## About relational databases

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 might contain a “friends” table. Each row in the table could represent a single friend stored in the database. The table's columns might 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 could contain all the personal information for each friend, and a separate “phone numbers” table could 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 is no 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 instance, 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 a column containing the “friend id” of the friend to whom the phone number belongs, and another column containing the actual phone number. That way, no matter how many phone numbers are 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

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, and the computer 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 the appendix “SQL support in local databases” in the Adobe AIR language reference.

## About SQL database classes

To work with local SQL databases in ActionScript 3.0, you use instances of the classes in the `flash.data` package:

- **flash.data.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.
- **flash.data.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.
- **flash.data.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.

Other classes in the `flash.data` package provide constants that are used with the `SQLConnection` class:

- **flash.data.SQLColumnNameStyle** Defines a set of constants representing the possible values for the `SQLConnection.columnNameStyle` property.
- **flash.data.SQLTransactionLockType** Defines a set of constants representing the possible values for the `option` parameter of the `SQLConnection.begin()` method.

In addition, the following classes in the `flash.events` package represent the events (and supporting constants) that you will use:

- **flash.events.SQLEvent** Defines the events that are dispatched when any of the `SQLConnection` or `SQLStatement` operations execute successfully. Each operation has an associated event type, all of which are defined by the `SQLEvent` class.
- **flash.events.SQLErrorEvent** Defines the event that is dispatched when any of the `SQLConnection` or `SQLStatement` operations results in an error.
- **flash.events.SQLUpdateEvent** Defines the event that is dispatched by a `SQLConnection` instance when table data in one of its connected databases changes as a result of an `INSERT`, `UPDATE`, or `DELETE` SQL statement being executed.

Finally, the following classes in the `flash.errors` package provide information about database operation errors:

- **flash.errors.SQLError** Provides information about a database operation error, including the operation that was being attempted and the cause of the failure.
- **flash.errors.SQLErrorCode** Defines a set of constants representing the possible values for the `SQLError` class's `code` property, which indicates the type of error that led to a failed operation.
- **flash.errors.SQLErrorOperation** Defines a set of constants representing the possible values for the `SQLError` class's `operation` property, which indicates the database operation that resulted in an error.

## Common SQL database tasks

This section describes the most common tasks that you will perform when you're working with local SQL databases. This includes connecting to a database, adding data to and retrieving data from tables in a database, and issues 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 you'll often need to do before you can perform the tasks in this section. 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 the section [“Creating and modifying a database” on page 92](#).

## Connecting to a database

Before you can perform any database operations, you need to 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, known as the “main” database, is connected using the `open()` method. Opening a database is an asynchronous operation, meaning you must register for the `SQLConnection` instance's `open` event in order to know when the `open()` operation completes, and the `SQLConnection` instance's `error` event to determine if the operation fails. The following example shows how to open an existing database file named “DBSample.db,” located in the user's application storage directory.

```
import flash.data.SQLConnection;
import flash.events.SQLErrorEvent;
import flash.events.SQLEvent;
import flash.filesystem.File;

var conn:SQLConnection = new SQLConnection();
conn.addEventListener(SQLEvent.OPEN, openHandler);
conn.addEventListener(SQLErrorEvent.ERROR, errorHandler);

var dbFile:File = File.applicationStorageDirectory.resolve("DBSample.db");

conn.open(dbFile, false);

function openHandler(event:SQLEvent):void
{
    trace("the database was opened successfully");
}

function errorHandler(event:SQLErrorEvent):void
{
    trace("Error code:", event.error.code);
    trace("Details:", event.error.message);
}
```

Notice that in the `open()` method call, the second argument is `false`. Specifying `false` for the second parameter (`autoCreate`) causes the runtime to dispatch an error if the specified file doesn't exist. If you pass `true` for the `autoCreate` parameter (or leave it off and only specify a file location as the first parameter), the runtime will attempt to create a new database file if the specified file doesn't exist.

## Working with SQL statements

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:

### 1. Specify which database the query will run against.

To do this, set the `SQLStatement` object's `sqlConnection` property to the `SQLConnection` instance that's connected with the desired database.

## 2. Specify the actual SQL statement.

Create the statement as a String and assign it to the `SQLStatement` instance's `text` property.

## 3. (Optional) Define functions to handle the result of the execute operation.

Use the `addEventListener()` method to register functions as listeners for the `SQLStatement` instance's `result` and `error` events. Alternatively, you can specify listener functions by creating a `Responder` object that refers to the listener functions and passing it to the `execute()` method call.

## 4. If the statement text includes parameter definitions, assign values for those parameters.

To do this, use the `SQLStatement` instance's `parameters` associative array property.

## 5. Execute the SQL statement.

Call the `SQLStatement` instance's `execute()` method.

For specific examples that demonstrate these steps, see the following sections:

- [Retrieving data from a database](#)
- [Inserting data](#)
- [Changing or deleting data](#)

### Using parameters in statements

See `SQLStatement.parameters`

## Retrieving data from a database

To retrieve existing data from a database, you create a `SQLStatement` instance, assign the desired SQL `SELECT` statement to the instance's `text` property, then call the `execute()` method. For details on the syntax of the `SELECT` statement, see the appendix “SQL support in local databases” in the Adobe AIR language reference.

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 in the `execute()` call, the `Responder` object's result handler function is called. Each row of data in the result becomes an `Object` instance whose properties' names match the result set's column names and whose properties contain the values from the result set's columns. For example, if 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`, containing the values from their respective columns. These result objects are contained in an array that is available as the `data` property of a `SQLResult` instance. If you're using an event listener, to retrieve that `SQLResult` instance you call the `SQLStatement` instance's `getResult()` method once the `result` event is dispatched. If you specify a `Responder` argument in the `execute()` call, the `SQLResult` instance is passed to the result handler function as an argument. In either case, once you have the `SQLResult` object you can access the result rows using the `data` array property.

For example, 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`. When the execution completes, the `selectResult()` method is called, and the resulting rows of data are retrieved 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.

```
import flash.data.SQLConnection;
import flash.data.ResultSet;
import flash.data.SQLStatement;
import flash.events.SQLErrorEvent;
import flash.events.SQLEvent;

// ... create and open the SQLConnection instance named conn

// create the SQL statement
var selectStmt:SQLStatement = new SQLStatement();
selectStmt.sqlConnection = conn;

// define the SQL text
var sql:String =
    "SELECT firstName, lastName " +
    "FROM employees";
selectStmt.text = sql;

// register listeners for the result and error events
selectStmt.addEventListener(SQLEvent.RESULT, selectResult);
selectStmt.addEventListener(SQLErrorEvent.ERROR, selectError);

// execute the statement
selectStmt.execute();
```

```
function selectResult(event:SQLEvent):void
{
    var result:SQLResult = selectStmt.getResult();
    var numRows:int = result.data.length;
    for (var i:int = 0; i < numRows; i++)
    {
        var output:String = "";
        for (var columnName:String in result.data[i])
        {
            output += columnName + ": " + result.data[i][columnName] + "; ";
        }
        trace("row[" + i.toString() + "]\t", output);
    }
}

function selectError(event:SQLErrorEvent):void
{
    trace("Error code:", event.error.code);
    trace("Details:", event.error.message);
}
```

### Defining the data type of SELECT result data

See `SQLStatement.itemClass`

### Retrieving SELECT results in parts

By default, a `SELECT` statement execution retrieves all the rows of the result set at one time. Once the statement completes, you will 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 “pause” and not refresh 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 will cause 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. When you call a `SQLStatement` instance's `execute()` method, specify a value for the `prefetch` parameter (the `execute()` method's first parameter) to indicate the number of result rows to retrieve when the statement executes:

```
var stmt:SQLStatement = new SQLStatement();
stmt.sqlConnection = conn;
stmt.text = "SELECT ...";
stmt.addEventListener(SQLEvent.RESULT, selectResult);
stmt.execute(20); // only the first 20 rows (or fewer) will be returned
```

The statement dispatches the `result` event, indicating that the first set of result rows are 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 subsequent 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:SQLEvent):void
{
    var result:SQLResult = 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(SQLEvent.RESULT, selectResult);
        }
    }
}
```

The `SQLStatement` dispatches a `result` event each time the `next()` method returns a subsequent set of result rows, so the same listener function can be used to continue processing results until all the rows are retrieved.

For more information, see the language reference descriptions for the `SQLStatement.execute()` method (the `prefetch` parameter description) and the `SQLStatement.next()` method.



## Inserting data

To add data to a table in a database, you create and execute a `SQLStatement` instance using a SQL `INSERT` statement. The following example uses a `SQLStatement` instance to add a row of data to the already-existing `employees` table. 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.

```
import flash.data.SQLConnection;
import flash.data.ResultSet;
import flash.data.SQLStatement;
import flash.events.SQLErrorEvent;
import flash.events.SQLEvent;

// ... create and open the SQLConnection instance named conn

// create the SQL statement
var insertStmt:SQLStatement = new SQLStatement();
insertStmt.sqlConnection = conn;

// define the SQL text
var sql:String =
    "INSERT INTO employees (firstName, lastName, salary) " +
    "VALUES ('Bob', 'Smith', 8000)";
insertStmt.text = sql;

// register listeners for the result and failure (status) events
insertStmt.addEventListener(SQLEvent.RESULT, insertResult);
insertStmt.addEventListener(SQLErrorEvent.ERROR, insertError);

// execute the statement
insertStmt.execute();

function insertResult(event:SQLEvent):void
{
    trace("INSERT statement succeeded");
}

function insertError(event:SQLErrorEvent):void
{
    trace("Error code:", event.error.code);
    trace("Details:", event.error.message);
}
```

## Retrieving a database-generated primary key of an inserted row

Often after inserting a row of data into a table, you will want to retrieve 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, inserting the primary key value as a foreign key in the related table. The primary key of a newly-inserted row can be retrieved in a way that’s similar to the way result data is retrieved after a `SELECT` statement is executed—using the `ResultSet` object associated with the statement execution.

Like with a `SELECT` or any other SQL statement, when an `INSERT` statement execution completes and the `result` event is dispatched, the runtime creates a `SQLResult` instance that is accessed by calling the `SQLStatement` object's `getResult()` method (if you're using an event listener) or as the argument passed to the result handler function (if you specified a `Responder` in the `execute()` call). 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. 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 structure, according to the following rule:

- 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 a row identifier value for the row, and 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 will contain the row identifier of the last row inserted by the trigger, rather than the row created by the `INSERT` statement.

Consequently, 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. Note, however, that 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 the sections titled “CREATE TABLE” and “Expressions” in the appendix “SQL support in local databases” in the Adobe AIR language reference.

## Changing or deleting data

The process for executing other data manipulation operations is identical to the process used to execute a SQL `SELECT` or `INSERT` statement; you 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 the appendix “SQL support in local databases” in the Adobe AIR language reference.

## Handling database errors

In general, database error handling is similar to 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. The `SQLErrorCode` class defines a set of constants that represent the set of errors that can occur during the various database operations. In a general sense, the possible database errors can be divided into three categories: connection errors, SQL syntax errors, and constraint errors.

### Connection errors

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 will occur if the user doesn't have permission to create a database file in a particular location on the file system. The following strategies will help you 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, stored in a location that's associated with the user's account such as 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 will have administrator permission on the computer, or that the individual who installed the application is the user who's running the application.

**Consider various file locations** If you allow users to specify the location where a database file is saved or the location of a database file to open, consider the possible file locations that users might specify. 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 is likely to occur on the first attempt to create or open the database. This means that the user will be 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 implement application logic that copies the database file to (or creates a new database file in) a different location where the user does have permission to create and write to files.

### Syntax errors

A syntax error occurs when a SQL statement is incorrectly formed, and the application attempts to prepare or execute the statement. Because local database SQL statements are created as strings, compile-time SQL syntax checking is not possible, and all SQL statements must be executed (or prepared) 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, and carefully validate any user input to make sure it won't cause syntax errors.

**Use prepared SQL statements** In many ways this is simply a combination of the two previous strategies. Prepared statements can't be dynamically generated. Syntax checking happens when a statement is prepared, so if you use prepared SQL statements in your application, and explicitly prepare all statements, each statement is tested. For more on using prepared statements, see [“Using prepared statements” on page 95](#).

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, you should also be able to identify any potential run-time sources of SQL syntax errors (such as user input used in a statement). To recover from a syntax error, you could provide the user with a specific error message guiding him or her to what needs to be corrected to make the statement execute correctly.

### Constraint errors

Constraint errors occur when an `INSERT` or `UPDATE` statement attempts to add data to a column that violates one of the defined constraints for the table or column. The following constraints can be defined:

**Unique constraint** Indicates that across all the rows in a table, there cannot be duplicate values in one column (or, 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(s), 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. Common examples of check constraints are 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), or one that specifies relationships between column values (for example, that a column's value must be different than the value of another column in the same row).

The runtime does not enforce constraints on foreign key values (foreign key values aren't required to match an existing primary key value). The runtime also does not enforce the data type of columns' values (although under some conditions the runtime converts an inserted values to another data type that matches its column's specified type). See the section “Data type support” in the appendix “SQL support in local databases” in the Adobe AIR language reference for more information.

In addition to the predefined constraint types, the runtime SQL engine supports the use of triggers. A trigger is a predefined set of instructions that are carried out when a certain action happens, such as 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 code for trigger-generated errors (the constant `SQLExceptionCode.ABORT`) is different from the error code for constraint errors (the constant `SQLExceptionCode.CONSTRAINT`).

Because constraints are specified when a table is created, triggers must be explicitly created, and Adobe AIR doesn't specify any additional constraints on tables or columns, the set of constraints that apply to a particular table can be identified ahead of time and can be taken into account both in preventing and in recovering from constraint errors.

Because constraint errors are dependent on data that is to be added to a database, and often on the relationship between the new data and other data that already exists in the database, constraint errors are difficult to systematically predict and prevent. 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, you will want to identify rules about 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 will 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 you should also specify a default value in the column definition. Application code can also provide default values, for example checking if a String variable is `null` and assigning 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 can prevent a constraint error from occurring.

Fortunately, while constraint errors are perhaps more difficult to prevent than other types of errors, 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, defining what action the database should take when a constraint violation occurs. The possible actions can range from ending a single statement or a whole transaction, to ignoring the error or even replacing old data with the new data in order to conform to a constraint. For more information see the section "ON CONFLICT (conflict algorithms)" in the appendix "SQL support in local databases" in the Adobe AIR language reference.

**Provide corrective feedback** Since the set of constraints that can affect a particular SQL command can be identified ahead of time, you can anticipate constraint errors that a statement could cause. With that knowledge, you can build application logic to respond to a constraint error, such as by showing the user a helpful error message that allows him or her to correct the error and attempt the operation again. For example, if an application includes a data entry form for entering new products, and 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, if a constraint error occurs, the application could prompt the user to indicate that the specified product name is already in use, and encourage the user to choose a different name (or perhaps allow the user to view information about the other product with the same name).

## Data type conversion between SQL and ActionScript

# Creating and modifying a database

This section covers database tasks that are less-frequently used, but are generally necessary for most applications. This section covers the following topics:

- [Creating a new database](#)
- [Creating database tables](#)

## Creating a new database

To create a new database file, you create a `SQLConnection` instance and call its `open()` method, as you would to open a connection to an existing database file. By specifying a file parameter that indicates a file location that does not exist, the `open()` method will create a new database file at that location, then open a connection to the newly-created database. When using the `open()` method to create a new database file, you can use the `autoClean` and `pageSize` parameters to define those characteristics of the database being created.

The following code listing shows the process of creating a new database file (a new database). In this case, the database file is saved in the application's storage directory, with the file name "DBSample.db":

```
import flash.data.SQLConnection;
import flash.events.SQLErrorEvent;
import flash.events.SQLEvent;
import flash.filesystem.File;

var conn:SQLConnection = new SQLConnection();
conn.addEventListener(SQLEvent.OPEN, openHandler);
conn.addEventListener(SQLErrorEvent.ERROR, errorHandler);

var dbFile:File = File.applicationStorageDirectory.resolve("DBSample.db");

conn.open(dbFile);

function openHandler(event:SQLEvent):void
{
    trace("the database was created successfully");
}

function errorHandler(event:SQLErrorEvent):void
{
    trace("Error code:", event.error.code);
    trace("Details:", event.error.message);
}
```

A database's file name can be any valid file name, with any file extension. If you call the `open()` method with `null` for the file parameter, a new in-memory database is created rather than a database file on disk.

## Creating database tables

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.

The following example demonstrates creating a new table named “employees” in an existing database file. Note that this code assumes there is a `SQLConnection` instance named `conn` that is already instantiated and is already connected to a database.

```
import flash.data.SQLConnection;
import flash.data.SQLStatement;
import flash.events.SQLErrorEvent;
import flash.events.SQLEvent;

// ... create and open the SQLConnection instance named conn

var createStmt:SQLStatement = new SQLStatement();
createStmt.sqlConnection = conn;

var sql:String =
    "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(SQLEvent.RESULT, createResult);
createStmt.addEventListener(SQLErrorEvent.ERROR, createError);

createStmt.execute();

function createResult(event:SQLEvent):void
{
    trace("Table created");
}

function createError(event:SQLErrorEvent):void
{
    trace("Error code:", event.error.code);
    trace("Details:", event.error.message);
}
```

## Strategies for working with SQL databases

There are various ways that an application can access and work with a local SQL database, in terms of how the application code can be organized, the sequence and timing of how operations are performed, and so forth. The techniques you choose can have an impact on how easy it is to develop your application, how easy it is to update and expand, and how well it will perform from the users’ perspective.

### Improving database performance

This section describes several techniques that are built into Adobe AIR that allow you to improve the performance of database operations in your application:

- [Using prepared statements](#)

- [Minimizing runtime processing](#)

In addition to the techniques described in this section, the way a SQL statement is written can also impact database performance. Frequently, there are multiple ways to write a SQL `SELECT` statement to retrieve a particular result set, and in some cases the different approaches will require more or less effort from the database engine. This aspect of improving database performance—designing SQL statements for better performance—is not covered in the Adobe AIR documentation.

### Using prepared statements

See the language reference description for `SQLStatement.prepare()`

### Minimizing runtime processing

Using the following techniques can prevent unneeded work on the part of the database engine and make applications perform better:

- Always explicitly specify database names in statements—use “main” if it’s the main database—to prevent the runtime from having to check each database to find the matching table (and to avoid the possibility of having the runtime choose the wrong database).
- Always explicitly specify column names in `SELECT` and `INSERT` statements.
- Break up `SELECT` statements that retrieve large numbers of rows: see [“Retrieving SELECT results in parts” on page 86](#).

## Working with multiple databases

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, 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. One important point to keep in mind if you use this approach is that database transactions are not shared across `SQLConnection` instances. Consequently, if you connect to the same database file using multiple `SQLConnection` instances, you cannot rely on both connections’ data changes being applied in the expected manner. For example, if two operations are performed against the same database through different `SQLConnection` instances, and the application has an error after one operation takes place, the database data could be left in an intermediate state that would not be reversible and might affect the application.

## Distributing a pre-populated database

When you use an AIR local SQL database in your application, it’s likely that the application will expect a certain database structure. Some applications will also expect pre-populated data in their database file. One approach to creating the initial structure of the database for the application is to have the application check for the existence of its database file in a particular location. If the file doesn’t exist, the application can execute a set of commands to create the database file, create the database structure, and populate the tables with the initial data.



As an alternative to creating the database, structure, and data programmatically, you can distribute a pre-populated database with your application by including the database file in the application's AIR package. This has the benefit of removing the need for a complex set of program instructions (the code that creates the database and its tables) that are only used once in the lifetime of the application, but still add to the size and complexity of the application.

One important issue to keep in mind if you're using the approach of including a pre-created database in your AIR file has to do with the initial location where the database file is placed when the application is installed. Like all files that are included in an AIR package, a bundled database file will be installed in the application's resource directory. However, because the user who is running the application may not necessarily be the user who installed the application, that user may not have the necessary permission from the operating system to write to the file in its default location. Instead, it is recommended that you use the file that is included in the AIR package as a "template" database, from which individual users' databases can be created. The first time a user runs the application, the original database file can be copied into the user's application storage directory (or another location), then that database can be the one that's actually used by the application, thereby avoiding the issue.

## Best practices for working with local SQL databases

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** Even if your application doesn't need to execute any statements when it first loads, instantiate a `SQLConnection` object and call its `open()` method ahead of time (such as after the initial application startup) to avoid delays when running statements.

**Re-use database connections** If you will be accessing a certain database throughout the execution time of your application, keep a reference to a `SQLConnection` instance, and reuse it throughout the application.

**Prepare SQL statements** For any SQL statement that will be executed more than once in an application, create the `SQLStatement` instance and call its `prepare()` method ahead of time to reduce processing time later. For example, your application might load its initial screen or initial set of data to get the user started, then instantiate the `SQLStatement` instances and call their `prepare()` methods.

**Use statement parameters** Use `SQLStatement` parameters—never concatenate user input into statement text. Doing so makes your application more secure (it prevents the possibility of SQL injection attacks), makes it possible to use objects in queries (rather than only SQL literal values), and makes statements run more efficiently (because they can be reused without needing to be compiled with each execution).

**Use constants for column and parameter names** When you don't specify an `itemClass` for a `SQLStatement`, to avoid spelling errors, define constants for column names and use the constants in the statement text and for the property names when retrieving values from result objects.

## See also

["Improving database performance" on page 94](#)

# Chapter 23: PDF

This chapter contains the following sections:

- [AIR and PDF](#)

## AIR and PDF

Adobe Integrated Runtime (AIR) applications can render not only SWF files and HTML content but also PDF files. PDF files, implemented using HTMLControl objects and the WebKit engine, can either stretch across the full height and width of your application or alternatively as a portion of the interface. The Acrobat Reader browser plug-in controls display of PDF files in an Adobe AIR application, so modifications to Reader's toolbar interface (position, anchoring, visibility, etc.) will persist in subsequent viewing of PDF files in both AIR applications and the browser.

### Detecting PDF Capability

If the user does not have an installed version of Adobe Reader 8.1 plug-in or greater, PDF content will not display in an AIR application; the HTMLControl instance will remain unaffected. To detect if a user can render PDF content, first check the `HTMLControl.pdfCapability` property. This property will return one of the following constants of the `HTMLPDFCapability` class:

**HTMLPDFCapability.STATUS\_OK** A sufficient version (8.1 or greater) of Acrobat Reader is detected and PDF content can be loaded into an HTMLControl object

**HTMLPDFCapability.ERROR\_INSTALLED\_READER\_NOT\_FOUND** A No version of Acrobat Reader is detected. An HTMLControl object cannot display PDF content.

**HTMLPDFCapability.ERROR\_INSTALLED\_READER\_TOO\_OLD** A Acrobat Reader has been detected, but the version is too old. An HTMLControl object cannot display PDF content.

**HTMLPDFCapability.ERROR\_PREFERRED\_READER\_TOO\_OLD** A A sufficient version (8.1 or later) of Acrobat Reader is detected, but the version of Acrobat Reader that is setup to handle PDF content is older than Reader 8.1. An HTMLControl object cannot display PDF content.

***Note:** On Windows, if a Acrobat Acrobat or Acrobat Reader version 7.x or above, is currently running on the user's system, that version is used even if a later version that supports loading PDF loaded in an HTMLControl object is installed. In this case, if the the value of the pdfCampability property is HTMLPDFCapability.STATUS\_OK, when an AIR application attempts to load PDF content into an HTMLControl object, the older version of Acrobat or Reader displays an alert, without an error message displayed the Apollo runtime. If this is a possibility situation for your end users, you may consider providing them with instructions to close Acrobat while running your application. You may consider displaying these instructions if the PDF content does not load within an acceptable timeframe.*

The following code detects whether a user can display PDF content in an AIR application, and if not traces the error code that corresponds to the HTMLPDFCapability error object:

```
import flash.html.HTMLControl;
import flash.net.URLRequest;

if(HTMLControl.pdfCapability == HTMLPDFCapability.STATUS_OK) {
    trace("PDF content can be displayed");
}
else {
    trace("PDF cannot be displayed. Error code: " + HTMLControl.pdfCapability);
}
```

## Adding a PDF to the display

You can add a PDF within the Adobe Reader interface to an AIR application by creating an HTMLControl instance, setting its dimensions, and loading the path of a PDF.

The following example loads a PDF from an external site. Replace the URLRequest with the path to an available external PDF.

```
var request:URLRequest = new URLRequest("http://www.exampledomain.com/test.pdf");
pdf = new HTMLControl();
pdf.height = 800;
pdf.width = 600;
pdf.load(request);
container.addChild(pdf);
```

## Known limitations in the Beta release

- The display order of a PDF file operates differently than other display objects in an Apollo application. Although a PDF will correctly clip according to HTML display order, it will always sit on top of content in the AIR application's display order.
- The visual properties of an HTMLControl object that contains a PDF file cannot be changed. Changes to an HTMLControl object's filters, alpha, rotation, or scaling will render the PDF file invisible until the properties are reset.
- Within a PDF file, links to content within the PDF file will update the scroll position of the PDF. Links to content outside the PDF will redirect the HTMLControl object that contains the PDF. Links that target a new window will be ignored.

## Forms

PDF files that contain forms will only post correctly if form posting uses a supported link format and returns PDF or HTML. Adobe discourages the use of forms that return FDF or rely heavily on JavaScript communication. Collaborative PDF forms will not function correctly in AIR 1.0.

# Chapter 24: Capturing command-line arguments

An Adobe Integrated Runtime (AIR) application receives command-line arguments through invocation events. An invocation event is dispatched whenever the operating system attempts to start a given Adobe AIR application. If command-line arguments are supplied, the `InvokeEvent` object contains an array holding the argument values.

This chapter contains the following sections;

- [Invocation event basics](#)
- [Getting invocation information](#)
- [Example: Invocation event log](#)

## Invocation event basics

An invocation event is dispatched to your application whenever it is invoked. Your application can be invoked by:

- a user double-clicking the application icon in the desktop environment
- running the application from a command shell
- opening a file of a type registered with the application
- a user opening an AIR file containing a version of the application when the installed application has signaled that it should handle application updates itself by including a `<handleUpdates/>` flag in the application descriptor

Only one instance of an AIR application will be started. On subsequent invocations, AIR dispatches a new invocation event to the running instance.

The invocation event object contains any arguments passed to the application, and 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 command line arguments. Likewise, if the application was invoked because of an application update, the full path to the update AIR file is provided. The invocation event object is defined by the AIR `InvokeEvent` class.

Your application can handle invocation events by registering a listener with its `Shell` object:

```
Shell.shell.addEventListener(InvokeEvent.INVOKE,onInvokeEvent);
```

And defining an event handler:

```
var arguments:Array;  
var currentDir:File;  
  
public function onInvokeEvent(invocation:InvokeEvent):void{  
    arguments = invocation.arguments;  
    currentDir = invocation.currentDirectory;  
}
```

**Note:** Until a listener is registered, invocation events are queued, not discarded. As soon as a listener is registered, all the queued invocation events are delivered.

### See also

“Setting application properties” on page 42

“Updating applications programmatically” on page 75

## Getting invocation information

**Command-line arguments** 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
foo bar	{foo,bar}
foo "foo bar"	{foo,foo bar}
"foo" "bar"	{foo,bar}
"foo\"bar\""	{\"foo\",bar\"}

**Note:** The Mac OS X Finder automatically passes the process serial number as an argument with the form `-psn_NNN` (where `NNN` is the process serial number).

**Launch directory** The `InvokeEvent.currentDirectory` property contains a `File` object representing the directory from which the application was launched.

The current directory will be one of the following:

- the directory in which the application icon or shortcut used to launch the application is located
- the current directory of the command shell used to launch the application
- the directory in which the file that triggered the invocation is located
- the directory in which the update AIR file is located, when an application handles updates itself

**Path to the triggering file** 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. (It is your application's responsibility to open or perform the intended operation on the file.)

Likewise, when an application handles updates itself, the native path to the AIR file will be included when a user double-clicks an AIR file containing an application with a matching application ID.

**Command-line arguments from subsequent invocations** The new `InvokeEvent` object dispatched for each subsequent invocation of your application contains the command line arguments and invocation directory.

## Example: Invocation event log

The following example demonstrates how to register listeners for and handle the `InvokeEvent`. The example logs all the invocation events received and displays the current directory and command-line arguments.

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

<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical"
    applicationComplete="init()" title="Invocation Event Log">

    <mx:Script>
        <![CDATA[

import flash.events.InvokeEvent;
import flash.system.Shell;

public function init():void{
    Shell.shell.addEventListener(InvokeEvent.INVOKE,onInvoke);
}

public function onInvoke(invocationEvent:InvokeEvent):void{
    var now:String = new Date().toString();
    logEvent("Invoke event received: " + now);

    if(invocationEvent.currentDirectory != null){
        logEvent("Current directory=" + invocationEvent.currentDirectory.nativePath);
    } else {
        logEvent("--no directory information available--");
    }

    if(invocationEvent.arguments.length > 0){
        logEvent("Arguments: " + invocationEvent.arguments.toString());
    } else {
        logEvent("--no arguments--");
    }
}

public function logEvent(entry:String):void {
    log.text += entry + "\n";
    trace(entry);
}
```

```
}  
]]>  
</mx:Script>  
  
<mx:TextArea id="log" width="100%" height="100%" editable="false"  
    valueCommit="log.verticalScrollPosition=log.textHeight;"/>  
</mx:WindowedApplication>
```

# Appendix: New functionality in Adobe Integrated Runtime

This appendix provides an overview of the new functionality in Adobe Integrated Runtime (AIR), which is not available to SWF content running in Flash Player.

This appendix contains the following sections:

- [New runtime classes](#)
- [Runtime classes with new functionality](#)
- [New Flex components](#)
- [Service monitoring framework classes](#)

## New runtime classes

The following runtime classes are new in Adobe AIR. They are not available to SWF content running in the browser:

Class	Package
ClipboardManager	flash.desktop
Door	flash.system
DragActions	flash.desktop
DragManager	flash.desktop
DragOptions	flash.desktop
File	flash.filesystem
FileListEvent	flash.events
FileMode	flash.filesystem
FileStream	flash.filesystem
HTMLControl	flash.html
HTMLHost	flash.html
HTMLUncaughtJavaScriptExceptionEvent	flash.html
HTMLWindowCreateOptions	flash.html
Icon	flash.desktop
InvokeEvent	flash.events
JavaScriptFunction	flash.html
JavaScriptObject	flash.html
NativeDragEvent	flash.events



Class	Package
NativeMenu	flash.display
NativeMenuItem	flash.display
NativeWindow	flash.display
NativeWindowBoundsEvent	flash.events
NativeWindowCapabilities	flash.system
NativeWindowDisplayState	flash.display
NativeWindowDisplayStateEvent	flash.events
NativeWindowErrorEvent	flash.events
NativeWindowInitOptions	flash.display
NativeWindowResize	flash.display
NativeWindowSystemChrome	flash.display
NativeWindowType	flash.display
OutputProgressEvent	flash.events
Shell	flash.system
SQLDatabase	flash.data
SQLResultEvent	flash.events
SQLStatement	flash.data
SQLStatusEvent	flash.events
SQLStatusEventCodes	flash.events
SQLStatusEventLevels	flash.event
SQLUpdateEvent	flash.event
TransferableData	flash.desktop
TransferableFormats	flash.desktop
TransferableTransferMode	flash.desktop
URLRequestDefaults	flash.net
Updater	flash.system

Most of these classes are available only to content in the AIR application security sandbox (see [“Getting started with Adobe AIR” on page 5](#)). However, the following classes are also available to content running in other sandboxes:

- Door
- URLRequest.

## Runtime classes with new functionality

The following classes are available to SWF content running in the browser, but AIR provides additional properties or methods:

Class	Property or Method
ByteArray	inflate() deflate()
HTTPStatusEvent	HTTP_RESPONSE_STATUS responseURL responseHeaders
URLRequest	followRedirects manageCookies shouldAuthenticate shouldCacheResponse userAgent userCache setLoginCredentials()
URLStream	httpResponseStatus event
Stage	window
Security	APPLICATION

Most of these new properties and methods are available only to content in the AIR application security sandbox. However, the new members in the following classes are also available to content running in other sandbox:

- ByteArray
- URLRequest

## New Flex components

The following Flex components are available when developing content running the Adobe Integrated Runtime:

- FileEvent
- FileSystemComboBox
- FileSystemDataGrid
- FileSystemEnumerationMode
- FileSystemHistoryButton
- FileSystemList
- FileSystemSizeDisplayMode
- FileSystemTree

- HTML
- WindowedApplication

## Service monitoring framework classes

The `air.net` package contains classes for network detection. This package is only available to content running in Adobe Integrated Runtime. It is included in the `ServiceMonitor.swc` file.

- `air.net.ServiceMonitor`
- `air.net.SocketMonitor`
- `air.net.URLMonitor`