Developing Adobe® AIR™ 1.5 Applications with HTML and Ajax
© 2009 Adobe Systems Incorporated. All rights reserved.

Developing Adobe® AIR® 1.5 Applications with HTML and Ajax

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 or person names in sample templates are for demonstration purposes only and are not intended to refer to any actual organization or person.

Adobe, the Adobe logo, Acrobat, ActionScript, Adobe AIR, ColdFusion, Dreamweaver, Flash, Flex, Flex Builder, and Reader are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Apple, Macintosh, and Mac OS are trademarks of Apple Inc., registered in the United States and other countries. Java is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries. Linux is the registered trademark of Linus Torvalds in the U.S. and other countries. All other trademarks are the property of their respective owners.

This work is licensed under the Creative Commons Attribution Non-Commercial 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc/3.0/us/

This product includes software developed by the Apache Software Foundation (http://www.apache.org/)

MPEG Layer-3 audio compression technology licensed from Fraunhofer IIS and Thomson Multimedia (http://www.mp3licensing.com).

Speech compression and decompression technology licensed from Nellymoser, Inc. (www.nellymoser.com)


This product includes software developed by the OpenSymphony Group (http://www.opensymphony.com/)

This product contains either BSAFE and/or TIPEM software by RSA Security, Inc.

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

This product includes software developed by the IronSmith Project (http://www.ironsmith.org/).

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.
Contents

Chapter 1: Adobe AIR installation
Installing Adobe AIR ................................................................. 1
Removing Adobe AIR ............................................................. 2
Installing and running the AIR sample applications ...................... 2

Chapter 2: Setting up HTML development tools
Installing the AIR Extension for Dreamweaver ............................... 3
Installing the AIR SDK .......................................................... 4

Chapter 3: Introducing Adobe AIR
What’s new in AIR 1.1 .................................................................. 7
What’s new in AIR 1.5 .................................................................. 8

Chapter 4: Finding AIR Resources

Chapter 5: Creating your first HTML-based AIR application with the AIR SDK
Create the project files .............................................................. 10
Create the AIR application descriptor file ..................................... 10
Create the application HTML page ............................................... 11
Test the application ...................................................................... 13
Create the AIR installation file ..................................................... 13
Next Steps ................................................................................. 14

Chapter 6: Create your first HTML-based AIR application with Dreamweaver
Prepare the application files ......................................................... 15
Create the Adobe AIR application ............................................... 15
Install the application on a desktop ............................................... 17
Preview the Adobe AIR application ............................................. 17

Chapter 7: Using the AIR Extension for Dreamweaver
Creating an AIR application in Dreamweaver ............................... 18
Signing an application with a digital certificate ............................... 20
Editing associated AIR file types ............................................... 21
Editing AIR application settings .................................................. 21
Previewing a web page in an AIR application ............................... 22
Using AIR code hinting and code coloring .................................... 22
Accessing the Adobe AIR documentation ..................................... 22

Chapter 8: Creating an AIR application using the command line tools
Using the AIR Debug Launcher (ADL) .......................................... 23
Packaging an AIR installation file using the AIR Developer Tool (ADT) ........................................................................ 25
Signing an AIR file to change the application certificate ............... 35
Creating a self-signed certificate with ADT .................................... 36
Using Apache Ant with the SDK tools ......................................... 36
### Chapter 9: Debugging with the AIR HTML Introspector

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>About the AIR Introspector</td>
<td>40</td>
</tr>
<tr>
<td>Loading the AIR Introspector code</td>
<td>40</td>
</tr>
<tr>
<td>Inspecting an object in the Console tab</td>
<td>41</td>
</tr>
<tr>
<td>Configuring the AIR Introspector</td>
<td>43</td>
</tr>
<tr>
<td>AIR Introspector interface</td>
<td>43</td>
</tr>
<tr>
<td>Using the AIR Introspector with content in a non-application sandbox</td>
<td>49</td>
</tr>
</tbody>
</table>

### Chapter 10: Programming in HTML and JavaScript

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating an HTML-based AIR application</td>
<td>51</td>
</tr>
<tr>
<td>An example application and security implications</td>
<td>52</td>
</tr>
<tr>
<td>Avoiding security-related JavaScript errors</td>
<td>53</td>
</tr>
<tr>
<td>Accessing AIR API classes from JavaScript</td>
<td>57</td>
</tr>
<tr>
<td>About URLs in AIR</td>
<td>59</td>
</tr>
<tr>
<td>Embedding SWF content in HTML</td>
<td>59</td>
</tr>
<tr>
<td>Using ActionScript libraries within an HTML page</td>
<td>60</td>
</tr>
<tr>
<td>Converting Date and RegExp objects</td>
<td>62</td>
</tr>
<tr>
<td>Manipulating an HTML stylesheet from ActionScript</td>
<td>62</td>
</tr>
<tr>
<td>Cross-scripting content in different security sandboxes</td>
<td>63</td>
</tr>
</tbody>
</table>

### Chapter 11: About the HTML environment

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview of the HTML environment</td>
<td>67</td>
</tr>
<tr>
<td>AIR and WebKit extensions</td>
<td>70</td>
</tr>
</tbody>
</table>

### Chapter 12: Handling HTML-related events

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTMLLoader events</td>
<td>82</td>
</tr>
<tr>
<td>How AIR class-event handling differs from other event handling in the HTML DOM</td>
<td>82</td>
</tr>
<tr>
<td>Adobe AIR event objects</td>
<td>84</td>
</tr>
<tr>
<td>Handling runtime events with JavaScript</td>
<td>86</td>
</tr>
</tbody>
</table>

### Chapter 13: Scripting the HTML Container

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display properties of HTMLLoader objects</td>
<td>89</td>
</tr>
<tr>
<td>Accessing the HTML history list</td>
<td>91</td>
</tr>
<tr>
<td>Setting the user agent used when loading HTML content</td>
<td>92</td>
</tr>
<tr>
<td>Setting the character encoding to use for HTML content</td>
<td>92</td>
</tr>
<tr>
<td>Defining browser-like user interfaces for HTML content</td>
<td>93</td>
</tr>
</tbody>
</table>

### Chapter 14: AIR security

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIR security basics</td>
<td>100</td>
</tr>
<tr>
<td>Installation and updates</td>
<td>100</td>
</tr>
<tr>
<td>Sandboxes</td>
<td>104</td>
</tr>
<tr>
<td>HTML security</td>
<td>106</td>
</tr>
<tr>
<td>Scripting between content in different domains</td>
<td>111</td>
</tr>
<tr>
<td>Writing to disk</td>
<td>112</td>
</tr>
<tr>
<td>Working securely with untrusted content</td>
<td>113</td>
</tr>
<tr>
<td>Best security practices for developers</td>
<td>114</td>
</tr>
<tr>
<td>Code signing</td>
<td>115</td>
</tr>
</tbody>
</table>
Contents

Working with files ........................................................................................................................................... 198
Reading and writing files ............................................................................................................................ 201

Chapter 22: Drag and drop
Additional online information about dragging and dropping ........................................................................... 213
Drag and drop basics ....................................................................................................................................... 213
Default drag-and-drop behavior .................................................................................................................. 214
Drag-and-drop events in HTML ...................................................................................................................... 214
MIME types for the HTML drag-and-drop .................................................................................................... 215
Drag effects in HTML ....................................................................................................................................... 216
Dragging data out of an HTML element ......................................................................................................... 216
Dragging data into an HTML element ........................................................................................................... 217
Example: Overriding the default HTML drag-in behavior ............................................................................ 218
Handling file drops in non-application HTML sandboxes ........................................................................... 220

Chapter 23: Copy and paste
Additional online information about copying and pasting ............................................................................... 222
Copy-and-paste basics ....................................................................................................................................... 222
Reading from and writing to the system clipboard ......................................................................................... 223
HTML copy and paste ....................................................................................................................................... 224
Menu commands and keystrokes for copy and paste ...................................................................................... 225
Clipboard data formats ....................................................................................................................................... 228

Chapter 24: Working with byte arrays
Reading and writing a ByteArray ..................................................................................................................... 231
ByteArray example: Reading a .zip file ............................................................................................................ 237

Chapter 25: Working with local SQL databases
Additional online information about local SQL databases ............................................................................... 242
About local SQL databases ............................................................................................................................. 243
Creating and modifying a database ................................................................................................................ 246
Manipulating SQL database data .................................................................................................................. 250
Using synchronous and asynchronous database operations ........................................................................ 269
Using encryption with SQL databases ......................................................................................................... 274
Strategies for working with SQL databases .................................................................................................... 286

Chapter 26: Storing encrypted data
Adding data to the encrypted local store .......................................................................................................... 291
Accessing data in the encrypted local store ..................................................................................................... 291
Removing data from the encrypted local store ............................................................................................... 292

Chapter 27: Adding PDF content
Detecting PDF Capability ............................................................................................................................... 293
Loading PDF content ....................................................................................................................................... 294
Scripting PDF content ....................................................................................................................................... 294
Known limitations for PDF content in AIR ..................................................................................................... 296
Chapter 28: Working with sound
Basics of working with sound ................................................................. 298
Understanding the sound architecture .................................................. 298
Loading external sound files ................................................................. 299
Working with embedded sounds .......................................................... 301
Working with streaming sound files ...................................................... 302
Working with dynamically generated audio ........................................... 302
Playing sounds ..................................................................................... 304
Working with sound metadata .............................................................. 308
Accessing raw sound data .................................................................... 309
Capturing sound input .......................................................................... 312

Chapter 29: Using digital rights management
Additional online information about digital rights management ............... 316
Understanding the encrypted FLV workflow .......................................... 317
DRM-related members and events of the NetStream class ......................... 320
Using the DRMStatusEvent class .......................................................... 322
Using the DRMAuthenticateEvent class ................................................. 322
Using the DRMErrorEvent class ........................................................... 324
Using the DRMMember class ............................................................... 326
Using the DRMMember class ............................................................... 327

Chapter 30: Application launching and exit options
Additional online information about launch and exit options .................... 328
Application invocation .......................................................................... 328
Capturing command line arguments ...................................................... 329
Launching on login ................................................................................ 331
Browser invocation .............................................................................. 332
Application termination ........................................................................ 334

Chapter 31: Reading application settings
Reading the application descriptor file .................................................... 336
Getting the application and publisher identifiers ..................................... 336

Chapter 32: Working with runtime and operating system information
Managing file associations ..................................................................... 337
Getting the runtime version and patch level .......................................... 337
Detecting AIR capabilities ..................................................................... 338
Tracking user presence ......................................................................... 338

Chapter 33: Monitoring network connectivity
Detecting network connectivity changes ................................................. 339
Service monitoring basics ..................................................................... 339
Detecting HTTP connectivity ............................................................... 340
Detecting socket connectivity ............................................................... 340
Chapter 34: URL requests and networking
Basics of networking and communication .......................................................... 342
Using the URLRequest class ............................................................................. 343
Working with external data ................................................................................. 346
Using the URLStream class ................................................................................. 350
Socket connections .............................................................................................. 350
Opening a URL in the default system web browser ............................................. 354
Sending a URL to a server .................................................................................... 355

Chapter 35: Inter-application communication
About the LocalConnection class ........................................................................ 357
Sending messages between two applications ..................................................... 357
Connecting to content in different domains and to other AIR applications .......... 358

Chapter 36: Distributing, Installing, and Running AIR applications
Installing and running an AIR application from the desktop ............................ 360
Installing and running AIR applications from a web page ................................. 361
Enterprise deployment ........................................................................................ 369
Digitally signing an AIR file ................................................................................ 369

Chapter 37: Updating AIR applications
About updating applications ............................................................................... 376
Presenting a custom application update user interface ...................................... 378
Downloading an AIR file to the user’s computer .............................................. 378
Checking to see if an application is running for the first time ........................... 379
Using the update framework .............................................................................. 381

Chapter 38: Viewing Source Code
Loading, configuring, and opening the Source Viewer ...................................... 391
Source Viewer user interface ................................................................................ 394

Chapter 39: Localizing AIR applications
Introduction to localization .................................................................................. 395
Localizing the application name and description in the application installer ....... 395
Choosing a locale ................................................................................................ 396
Localizing HTML content ................................................................................... 396
Localizing dates, times, and currencies .............................................................. 404

Chapter 40: XML signature validation
Additional online information about XML signature validation ....................... 406
Basics of XML signature validation ................................................................... 406
About XML signatures ......................................................................................... 410
Implementing the IURIDereferencer interface .................................................... 413

Index ................................................................................................................... 421
Chapter 1: Adobe AIR installation

Adobe® AIR® allows you to run AIR applications on the desktop. You can install the runtime in the following ways:

- By installing the runtime separately (without also installing an AIR application)
- By installing an AIR application for the first time (you are prompted to also install the runtime)
- By setting up an AIR development environment such as the AIR SDK, Adobe® Flex™ Builder™ 3, or the Adobe Flex™ 3 SDK (which includes the AIR command line development tools)

The runtime only needs to be installed once per computer.

The system requirements for installing AIR and running AIR applications are detailed here: Adobe AIR: System requirements (http://www.adobe.com/products/air/systemreqs/).

Installing Adobe AIR

Use the following instructions to download and install the Windows®, Mac OS X, and Linux versions of AIR.

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

Install the runtime on a Windows computer

1. Download the runtime installation file.
2. Double-click the runtime installation file.
3. In the installation window, follow the prompts to complete the installation.

Install the runtime on a Mac computer

1. Download the runtime installation 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.

Install the runtime on a Linux computer

1. Download the runtime installation file.
2. Set the file permissions so that the installer application can be executed:
   From a command line, you can set the file permissions with the `chmod +x installer.bin` command. Some versions of Linux allow you to set the file permissions on the Properties dialog opened through a context menu.
3. Run the installer from the command line or by double-clicking the runtime installation file.
4. In the installation window, follow the prompts to complete the installation.

AIR is installed as either rpm or dpkg packages, with package names: `adobeair.v.n` and `adobecerts`. Installation requires a running X server. AIR registers the mime type: `application/vnd.adobe.air-application-installer-package+zip`. 
Removing Adobe AIR

Once you have installed the runtime, you can remove it using the following procedures.

Remove the runtime on a Windows computer
1 In the Windows Start menu, select Settings > Control Panel.
2 Select the Add or Remove Programs control panel.
3 Select “Adobe AIR” to remove the runtime.
4 Click the Change/Remove button.

Remove the runtime on a Mac computer
• Double-click the “Adobe AIR Uninstaller”, which is located in the /Applications/Utilities folder.

Remove the runtime on a Linux computer
Do one of the following:
• Select the “Adobe AIR Uninstaller” command from the Applications menu.
• Run the AIR installer binary with the -uninstall option
• Remove the AIR packages (adobeairv.n and adobecerts) with your package manager.

Installing and running the AIR sample applications

Some sample applications are available that demonstrate AIR features. You can access and install them using the following instructions:

1 Download and run the AIR sample applications. The compiled applications as well as the source code are available.
2 To download and run a sample application, click the sample application Install Now button. You are prompted to install and run the application.
3 If you choose to download sample applications and run them later, select the download links. You can run AIR applications at any time by:
  • On Windows, double-clicking the application icon on the desktop or selecting it from the Windows Start menu.
  • On Mac OS, double-clicking the application icon, which is installed in the Applications folder of your user directory (for example, in Macintosh HD/Users/JoeUser/Applications/) by default.
  • On Linux, double-clicking the application icon on the desktop or selecting it from the Applications menu. AIR applications are installed in their own folder under the /opt directory.

Note: Check the AIR release notes for updates to these instructions, which are located here: http://www.adobe.com/go/learn_air_relnotes.
Chapter 2: Setting up HTML development tools

To develop HTML-based Adobe® AIR® applications, you can use the Adobe®AIR® Extension for Dreamweaver, the AIR SDK command-line tools, or other Web development tools that support Adobe AIR. This topic explains how to install the Adobe AIR Extension for Dreamweaver and the AIR SDK.

Installing the AIR Extension for Dreamweaver

The AIR Extension for Dreamweaver helps you to create rich Internet applications for the desktop. For example, you might have a set of web pages that interact with each other to display XML data. You can use the Adobe AIR Extension for Dreamweaver to package this set of pages into a small application that can be installed on a user’s computer. When the user runs the application from their desktop, the application loads and displays the website in its own application window, independent of a browser. The user can then browse the website locally on their computer without an Internet connection.

Dynamic pages such as Adobe® ColdFusion® and PHP pages won’t run in Adobe AIR. The runtime only works with HTML and JavaScript. However, you can use JavaScript in your pages to call any web service exposed on the Internet—including ColdFusion- or PHP-generated services—with Ajax methods such as XMLHttpRequest or Adobe AIR-specific APIs.

For more information about the types of applications you can develop with Adobe AIR, see “Introducing Adobe AIR” on page 6.

System requirements

To use the Adobe AIR Extension for Dreamweaver, the following software must be installed and properly configured:

- Dreamweaver CS3 or Dreamweaver CS4
- Adobe® Extension Manager CS3
- Java JRE 1.4 or later (necessary for creating the Adobe AIR file). The Java JRE is available at http://java.sun.com/.

The preceding requirements are only for creating and previewing Adobe AIR applications in Dreamweaver. To install and run an Adobe AIR application on the desktop, you must also install Adobe AIR on your computer. To download the runtime, see www.adobe.com/go/air.

Install the Adobe AIR Extension for Dreamweaver

2. Double-click the .mxp extension file in Windows Explorer (Windows) or in the Finder (Macintosh).
3. Follow the onscreen instructions to install the extension.
4. After you’re finished, restart Dreamweaver.

For information about using the Adobe AIR Extension for Dreamweaver, see “Using the AIR Extension for Dreamweaver” on page 18.
Installing the AIR SDK

The Adobe AIR SDK contains the following command-line tools that you use to launch and package applications:

**AIR Debug Launcher (ADL)**  Allows you to run AIR applications without having to first install them. See “Using the AIR Debug Launcher (ADL)” on page 23.

**AIR Development Tool (ADT)**  Packages AIR applications into distributable installation packages. See “Packaging an AIR installation file using the AIR Developer Tool (ADT)” on page 25.

The AIR command-line tools require Java to be installed on your computer. You can use the Java virtual machine from either the JRE or the JDK (version 1.4 or newer). The Java JRE and the Java JDK are available at [http://java.sun.com/](http://java.sun.com/).

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

Download and install the AIR SDK

You can download and install the AIR SDK using the following instructions:

**Install the AIR SDK in Windows**

1. Download the AIR SDK installation file.

2. The AIR SDK is distributed as a standard file archive. To install AIR, extract the contents of the SDK to a folder on your computer (for example: C:\Program Files\Adobe\AIRSDK or C:\AIRSDK).

3. The ADL and ADT tools are contained in the bin folder in the AIR SDK; add the path to this folder to your PATH environment variable.

**Install the AIR SDK in Mac OS X**

1. Download the AIR SDK installation file.

2. The AIR SDK is distributed as a standard file archive. To install AIR, extract the contents of the SDK to a folder on your computer (for example: /Users/<userName>/Applications/AIRSDK).

3. The ADL and ADT tools are contained in the bin folder in the AIR SDK; add the path to this folder to your PATH environment variable.

   For information about getting started using the AIR SDK tools, see “Creating an AIR application using the command line tools” on page 23.

**What's included in the AIR SDK**

The following table describes the purpose of the files contained in the AIR SDK:
<table>
<thead>
<tr>
<th>SDK folder</th>
<th>Files/tools description</th>
</tr>
</thead>
</table>
| **BIN**    | **adl.exe** - The AIR Debug Launcher (ADL) allows you to run an AIR application without first packaging and installing it. For information about using this tool, see "Using the AIR Debug Launcher (ADL)" on page 23.  
Diff2 - The AIR Developer Tool (ADT) packages your application as an AIR file for distribution. For information about using this tool, see "Packaging an AIR installation file using the AIR Developer Tool (ADT)" on page 25. |
| **FRAMEWORKS** | **AIRAliases.js** - Provides "alias" definitions that allow you to access the ActionScript runtime classes. For information about using this alias file, see "Using the AIRAliases.js file" on page 58  
**servicemonitor.swf** - Provides AIR applications with an event-based means of responding to changes in network connectivity to a specified host. For information about using this framework, see "Monitoring network connectivity" on page 339. |
| **LIB** | **adt.jar** - The adt executable file, which is called by the adt.bat file.  
**Descriptor.1.0.xsd** - The application schema file. |
| **RUNTIME** | The AIR runtime - The runtime is used by ADL to launch your AIR applications before they have been packaged or installed. |
| **SAMPLES** | This folder contains a sample application descriptor file, a sample of the seamless install feature (badge.swf), and the default AIR application icons; see "Distributing, Installing, and Running AIR applications" on page 360. |
| **SRC** | This folder contains the source files for the seamless install sample. |
| **TEMPLATES** | **descriptor-template.xml** - A template of the application descriptor file, which is required for each AIR application. For a detailed description of the application descriptor file, see "Setting AIR application properties" on page 116. |
Chapter 3: Introducing Adobe AIR

Adobe® AIR® is a cross-operating system runtime that allows you to leverage your existing web development skills (Adobe® Flash® CS3 Professional, Adobe® Flash® CS4 Professional, Adobe® Flex™, Adobe® ActionScript® 3.0, HTML, JavaScript®, Ajax) to build and deploy Rich Internet Applications (RIAs) to the desktop.

You can find more information about getting started with and using Adobe AIR at the Adobe AIR Developer Connection (http://www.adobe.com/devnet/air/).

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 the following technologies:

- 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.

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, which has the following 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 can be built faster by enabling you to leverage existing web technologies and design patterns and extend your web based applications to the desktop without learning traditional desktop development technologies or the complexity of native code.
- Application development is easier than using lower level languages such as C and C++. You do not need to manage the 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)
- HTML, CSS and JavaScript
Most Ajax frameworks
AIR 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.

What’s new in AIR 1.1
Adobe AIR 1.1 introduced the following new capabilities:

- Installation and other runtime dialog boxes have been translated into:
  - Brazilian Portuguese
  - Chinese (Traditional and Simplified)
  - French
  - German
  - Italian
  - Japanese
  - Korean
  - Russian
  - French
  - Spanish
- Support for building internationalized applications, including keyboard input for double-byte languages. See “Localizing AIR applications” on page 395.
  - Support for localizing the name and description attributes in the application descriptor file.
  - Support for localizing error messages, such as SQLError.detailID and SQLError.detailArguments, in the SQLite database.
  - Addition of Capabilities.languages property to obtain an array of preferred UI languages as set by the operating system.
  - HTML button labels and default menus, such as context menus and the Mac menu bar, have been localized to all supported languages.
- Support for certificate migration from a self-signed application to one that chains to a certificate of authority (CA).
- Addition of File.spaceAvailable API to obtain the amount of disk space available on a disk.
- Addition of NativeWindow.supportsTransparency property to determine whether a window can be drawn as transparent by the current operating system.

For more information about the AIR 1.1 release, see the Adobe AIR 1.1 Release Notes (http://www.adobe.com/go/learn_air_relnotes_en).
What’s new in AIR 1.5

Adobe AIR 1.5 introduces the following new features:

- Support for the following features of Flash Player 10.
  - Custom Filters and Effects
  - Enhanced Drawing API
  - Dynamic Sound Generation
  - Vector Data Type
  - Enhanced file upload and download APIs
  - Real Time Media Flow Protocol (RTMFP)
  - 3D Effects
  - Advanced Text Support
  - Color Management
  - Text Engine
  - Dynamic Streaming
  - Speex Audio Codec

For more information, see http://www.adobe.com/products/flashplayer/features/ for details of these features.

- Additional languages supported in the AIR 1.5 installer and other runtime dialog boxes: Czech, Dutch, Swedish, Turkish, Polish.

- Database encryption.
  Database files can be encrypted in AIR 1.5. All database content, including the metadata, can be encrypted so that the data is unreadable outside of the AIR application that encrypted it. This feature will allow a developer to encrypt, decrypt, and re-encrypt database files. See “Storing encrypted data” on page 290.

- The version of WebKit used by Adobe AIR has been updated and it now includes support for the SquirrelFish JavaScript interpreter.

- New XML signature validation APIs that can be used to help verify the integrity and signer identity of data or information. See “XML signature validation” on page 406.

For more information about the AIR 1.5 release, see the Adobe AIR 1.5 Release Notes (http://www.adobe.com/go/learn_air_relnotes_en).
Chapter 4: Finding AIR Resources

For more information on developing Adobe® AIR® applications, see the following resources:

<table>
<thead>
<tr>
<th>Source</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adobe AIR Quick Starts for HTML</td>
<td><a href="http://www.adobe.com/go/learn_air_html_qs_en">http://www.adobe.com/go/learn_air_html_qs_en</a></td>
</tr>
</tbody>
</table>

You can find articles, samples and presentations by both Adobe and community experts on the Adobe AIR Developer Connection at http://www.adobe.com/devnet/air/. You can also download Adobe AIR and related software from there.

You can find a section specifically for HTML and Ajax developers at http://www.adobe.com/devnet/air/ajax/.

Visit the Adobe Support website, at http://www.adobe.com/support/, to find troubleshooting information for your product and to learn about free and paid technical support options. Follow the Training link for access to Adobe Press books, a variety of training resources, Adobe software certification programs, and more.
Chapter 5: Creating your first HTML-based AIR application with the AIR SDK

For a quick, hands-on illustration of how Adobe® AIR® works, use these instructions to create and package a simple HTML-based AIR “Hello World” application.

To begin, you must have installed the runtime and set up the AIR SDK. You will use the AIR Debug Launcher (ADL) and the AIR Developer Tool (ADT) in this tutorial. ADL and ADT are command-line utility programs and can be found in the bin directory of the AIR SDK (see “Setting up HTML development tools” on page 3). This tutorial assumes that you are already familiar with running programs from the command line and know how to set up the necessary path environment variables for your operating system.

Create the project files

Every HTML-based AIR project must contain the following two files: an application descriptor file, which specifies the application metadata, and a top-level HTML page. In addition to these required files, this project includes a JavaScript code file, AIRAliases.js, that defines convenient alias variables for the AIR API classes.

To begin:
1. Create a directory named HelloWorld to contain the project files.
2. Create an XML file, named HelloWorld-app.xml.
3. Create an HTML file named HelloWorld.html.
4. Copy AIRAliases.js from the frameworks folder of the AIR SDK to the project directory.

Create the AIR application descriptor file

To begin building your AIR application, create an XML application descriptor file with the following structure:

```xml
<application>
  <id>…</id>
  <version>…</version>
  <filename>…</filename>
  <initialWindow>
    <content>…</content>
    <visible>…</visible>
    <width>…</width>
    <height>…</height>
  </initialWindow>
</application>
```

1. Open the HelloWorld-app.xml for editing.
2. Add the root <application> element, including the AIR namespace attribute:

   ```xml
   <application xmlns="http://ns.adobe.com/air/application/1.5">
   The last segment of the namespace, “1.5”, specifies the version of the runtime required by the application.
3 Add the `<id>` element:

```xml
<id>examples.html.HelloWorld</id>
```

The application id uniquely identifies your application along with the publisher id (which AIR derives from the certificate used to sign the application package). The recommended form is a dot-delimited, reverse-DNS-style string, such as "com.company.AppName". The application id is used for installation, access to the private application file-system storage directory, access to private encrypted storage, and interapplication communication.

4 Add the `<version>` element:

```xml
<version>0.1</version>
```

Helps users to determine which version of your application they are installing.

5 Add the `<filename>` element:

```xml
<filename>HelloWorld</filename>
```

The name used for the application executable, install directory, and other references to the application in the operating system.

6 Add the `<initialWindow>` element containing the following child elements to specify the properties for your initial application window:

```xml
<content>HelloWorld.html</content>
<visible>true</visible>
<width>400</width>
<height>200</height>
```

Identifies the root HTML file for AIR to load.

Makes the window visible immediately.

Sets the window width (in pixels).

Sets the window height.

7 Save the file. The completed application descriptor file should look like the following:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://ns.adobe.com/air/application/1.5">
  <id>examples.html.HelloWorld</id>
  <version>0.1</version>
  <filename>HelloWorld</filename>
  <initialWindow>
    <content>HelloWorld.html</content>
    <visible>true</visible>
    <width>400</width>
    <height>200</height>
  </initialWindow>
</application>
```

This example only sets a few of the possible application properties. For the full set of application properties, which allow you to specify such things as window chrome, window size, transparency, default installation directory, associated file types, and application icons, see “Setting AIR application properties” on page 116.

Create the application HTML page

You now need to create a simple HTML page to serve as the main file for the AIR application.

1 Open the HelloWorld.html file for editing. Add the following HTML code:
In the <head> section of the HTML, import the AIRAliases.js file:

```html
<script src="AIRAliases.js" type="text/javascript"></script>
```

AIR defines a property named runtime on the HTML window object. The runtime property provides access to the built-in AIR classes, using the fully qualified package name of the class. For example, to create an AIR File object you could add the following statement in JavaScript:

```javascript
var textFile = new runtime.flash.filesystem.File("app:/textfile.txt");
```

The AIRAliases.js file defines convenient aliases for the most useful AIR APIs. Using AIRAliases.js, you could shorten the reference to the File class to the following:

```javascript
var textFile = new air.File("app:/textfile.txt");
```

Below the AIRAliases script tag, add another script tag containing a JavaScript function to handle the onLoad event:

```html
<script type="text/javascript">
  function appLoad(){
    air.trace("Hello World");
  }
</script>
```

The appLoad() function simply calls the air.trace() function. The trace message print to the command console when you run the application using ADL. Trace statements can be very useful for debugging.

4 Save the file.

Your HelloWorld.html file should now look like the following:

```html
<html>
<head>
  <title>Hello World</title>
  <script type="text/javascript" src="AIRAliases.js"></script>
  <script type="text/javascript">
    function appLoad(){
      air.trace("Hello World");
    }
  </script>
</head>
<body onLoad="appLoad()">
  <h1>Hello World</h1>
</body>
</html>
```
Test the application

To run and test the application from the command line, use the AIR Debug Launcher (ADL) utility. The ADL executable can be found in the bin directory of the AIR SDK. If you haven’t already set up the AIR SDK, see “Setting up HTML development tools” on page 3.

❖ First, open a command console or shell. Change to the directory you created for this project. Then, run the following command:

```
adl HelloWorld-app.xml
```

An AIR window opens, displaying your application. Also, the console window displays the message resulting from the `air.trace()` call.

For more information, see “Using the AIR Debug Launcher (ADL)” on page 23.

Create the AIR installation file

When your application runs successfully, you can use the ADT utility to package the application into an AIR installation file. An AIR installation file is an archive file that contains all the application files, which you can distribute to your users. You must install Adobe AIR before installing a packaged AIR file.

To ensure application security, all AIR installation files must be digitally signed. For development purposes, you can generate a basic, self-signed certificate with ADT or another certificate generation tool. You can also buy a commercial code-signing certificate from a commercial certificate authority such as VeriSign or Thawte. When users install a self-signed AIR file, the publisher is displayed as “unknown” during the installation process. This is because a self-signed certificate only guarantees that the AIR file has not been changed since it was created. There is nothing to prevent someone from self-signing a masquerade AIR file and presenting it as your application. For publicly released AIR files, a verifiable, commercial certificate is strongly recommended. For an overview of AIR security issues, see “AIR security” on page 100.

Generate a self-signed certificate and key pair

❖ From the command prompt, enter the following command (the ADT executable is located in the bin directory of the AIR SDK):

```
adt --certificate --cn SelfSigned 1024-RSA sampleCert.pfx samplePassword
```

ADT generates a keystore file named `sampleCert.pfx` containing a certificate and the related private key.

This example uses the minimum number of attributes that can be set for a certificate. You can use any values for the parameters in italics. The key type must be either 1024-RSA or 2048-RSA (see “Digitally signing an AIR file” on page 369).

Create the AIR installation file

❖ From the command prompt, enter the following command (on a single line):
Creating your first HTML-based AIR application with the AIR SDK

You will be prompted for the keystore file password.

The HelloWorld.air argument is the AIR file that ADT produces. HelloWorld-app.xml is the application descriptor file. The subsequent arguments are the files used by your application. This example only uses two files, but you can include any number of files and directories.

After the AIR package is created, you can install and run the application by double-clicking the package file. You can also type the AIR filename as a command in a shell or command window.

**Next Steps**

In AIR, HTML and JavaScript code generally behaves the same as it would in a typical web browser. (In fact, AIR uses the same WebKit rendering engine used by the Safari web browser.) However, there are some important differences that you must understand when you develop HTML applications in AIR. For more information on these differences, and other important topics, see:
Chapter 6: Create your first HTML-based AIR application with Dreamweaver

For a quick, hands-on illustration of how Adobe® AIR® works, use these instructions to create and package a simple HTML-based AIR “Hello World” application using the Adobe® AIR® Extension for Dreamweaver.

If you haven’t already done so, download and install Adobe AIR, which is located here: www.adobe.com/go/air.

For instructions on installing the Adobe AIR Extension for Dreamweaver, see “Installing the AIR Extension for Dreamweaver” on page 3.

For an overview of the extension, including system requirements, see “Using the AIR Extension for Dreamweaver” on page 18.

Prepare the application files

Your Adobe AIR application must have a start page and all of its related pages defined in a Dreamweaver site. (For more information on Dreamweaver sites, see Dreamweaver Help.) To create a start page for a simple “Hello World” AIR application, follow these instructions:

1  Start Dreamweaver and make sure you have a site defined.
2  Open a new HTML page by selecting File > New, selecting HTML in the Page Type column, selecting None in the Layout column, and clicking Create.
3  In the new page, type Hello World!
   This example is extremely simple, but if you want you can style the text to your liking, add more content to the page, link other pages to this start page, and so on.
4  Save the page (File > Save) as hello_world.html. Make sure you save the file in a Dreamweaver site.

Create the Adobe AIR application

1  Make sure you have the hello_world.html page open in the Dreamweaver Document window. (See the previous section for instructions on creating it.)
2  Select Site > Air Application Settings.
   Most of the required settings in the AIR Application and Settings dialog box are auto-populated for you. You must, however, select the initial content (or start page) of your application.
3  Click the Browse button next to the Initial Content option, navigate to your hello_world.html page, and select it.
4  Next to the Digital signature option, click the Set button.
   A digital signature provides an assurance that the code for an application has not been altered or corrupted since its creation by the software author, and is required on all Adobe AIR applications.
5  In the Digital Signature dialog box, select Sign the AIR package with a digital certificate, and click the Create button. (If you already have access to a digital certificate, you can click the Browse button to select it instead.)
6 Complete the required fields in the Self-Signed Digital Certificate dialog box. You’ll need to enter your name, enter a password and confirm it, and enter a name for the digital certificate file. Dreamweaver saves the digital certificate in your site root.

7 Click OK to return to the Digital Signature dialog box.

8 In the Digital Signature dialog box, enter the password you specified for your digital certificate and click OK.

Your completed AIR Application and Installer Settings dialog box might look like this:

For further explanation about all of the dialog box options and how to edit them, see “Creating an AIR application in Dreamweaver” on page 18.

9 Click the Create AIR File button.

Dreamweaver creates the Adobe AIR application file and saves it in your site root folder. Dreamweaver also creates an application.xml file and saves it in the same place. This file serves as a manifest, defining various properties of the application.
Install the application on a desktop

Now that you've created the application file, you can install it on any desktop.

1. Move the Adobe AIR application file out of your Dreamweaver site and onto your desktop, or to another desktop.
   This step is optional. You can actually install the new application on your computer right from your Dreamweaver site directory if you prefer.

2. Double-click the application executable file (.air file) to install the application.

Preview the Adobe AIR application

You can preview pages that will be part of AIR applications at any time. That is, you don’t necessarily need to package the application before seeing what it will look like when it’s installed.

1. Make sure your hello_world.html page is open in the Dreamweaver Document window.

2. On the Document toolbar, click the Preview/Debug in Browser button, and then select Preview In AIR.
   You can also press Ctrl+Shift+F12 (Windows) or Cmd+Shift+F12 (Macintosh).
   When you preview this page, you are essentially seeing what a user would see as the start page of the application after they've installed the application on a desktop.
Chapter 7: Using the AIR Extension for Dreamweaver

The Adobe® AIR® Extension for Dreamweaver® lets you transform a web-based application into a desktop application. Users can then run the application on their desktops and, in some cases, without an Internet connection.

You can use the extension with Dreamweaver CS3 and Dreamweaver CS4. It is not compatible with Dreamweaver 8.

For information about installing the extension, see “Installing the AIR Extension for Dreamweaver” on page 3.

**Note:** Adobe AIR does not support Adobe InContext Editing. If you use the AIR Extension for Dreamweaver to export an application that contains InContext Editing regions, the InContext Editing functionality will not work.

Creating an AIR application in Dreamweaver

To create an HTML-based AIR application in Dreamweaver, you select an existing site to package as an AIR application.

1. Make sure that the web pages you want to package into an application are contained in a defined Dreamweaver site.
2. In Dreamweaver, open the home page of the set of pages you want to package.
3. Select Site > Air Application Settings.
4. Complete the AIR Application and Installer Settings dialog box, and then click Create AIR File.

For more information, see the dialog box options listed below.

The first time you create an Adobe AIR file, Dreamweaver creates an application.xml file in your site root folder. This file serves as a manifest, defining various properties of the application.

The following describes the options in the AIR Application and Installer Settings dialog box:

- **Application File Name** is the name used for the application executable file. By default, the extension uses the name of the Dreamweaver site to name the file. You can change the name if you prefer. However, the name must contain only valid characters for files or folder names. (That is, it can only contain ASCII characters, and cannot end with a period.) This setting is required.

- **Application Name** is the name that appears on installation screens when users install the application. Again, the extension specifies the name of the Dreamweaver site by default. This setting does not have character restrictions, and is not required.

- **Application ID** identifies your application with a unique ID. You can change the default ID if you prefer. Do not use spaces or special characters in the ID. The only valid characters are 0-9, a-z, A-Z, . (dot), and - (dash). This setting is required.

- **Version** specifies a version number for your application. This setting is required.

- **Initial Content** specifies the start page for your application. Click the Browse button to navigate to your start page and select it. The chosen file must reside inside the site root folder. This setting is required.

- **Description** lets you specify a description of the application to display when the user installs the application.
Copyright lets you specify a copyright that is displayed in the About information for Adobe AIR applications installed on the Macintosh. This information is not used for applications installed on Windows.

Window Style specifies the window style (or chrome) to use when the user runs the application on their computer. System chrome surrounds the application with the operating system standard window control. Custom chrome (opaque) eliminates the standard system chrome and lets you create a chrome of your own for the application. (You build the custom chrome directly in the packaged HTML page.) Custom chrome (transparent) is like Custom chrome (opaque), but adds transparent capabilities to the edges of the page, allowing for application windows that are not rectangular in shape.

Window Size specifies the dimensions of your application window when it opens.

Icon lets you select custom images for the application icons. (The default images are Adobe AIR images that come with the extension.) To use custom images, click the Select Icon Images button. Then, in the Icon Images dialog box that appears, click the folder for each icon size and select the image file you want to use. AIR only supports PNG files for application icon images.

Note: Selected custom images must reside in the application site, and their paths must be relative to the site root.

Associated File Types lets you associate file types with your application. For more information, see the section that follows.

Application Updates determines whether the Adobe AIR Application Installer or the application itself performs updates to new versions of Adobe AIR applications. The check box is selected by default, which causes the Adobe AIR Application Installer to perform updates. If you want your application to perform its own updates, deselect the checkbox. Keep in mind that if you deselect the checkbox, you then need to write an application that can perform updates.

Included Files specifies which files or folders to include in your application. You can add HTML and CSS files, image files, and JavaScript library files. Click the Plus (+) button to add files, and the folder icon to add folders. You should not include certain files such as _mmServerScripts, _notes, and so on. To delete a file or folder from your list, select the file or folder and click the Minus (-) button.

Digital Signature Click Set to sign your application with a digital signature. This setting is required. For more information, see the section that follows.

Program Menu Folder specifies a subdirectory in the Windows Start Menu where you want the application’s shortcut created. (Not applicable on Macintosh.)

Destination specifies where to save the new application installer (.air file). The default location is the site root. Click the Browse button to select a different location. The default file name is based on the site name with an .air extension added to it. This setting is required.
The following is an example of the dialog box with some basic options set:

![AIR Application and Installer Settings dialog box](image)

### Signing an application with a digital certificate

A digital signature provides an assurance that the code for an application has not been altered or corrupted since its creation by the software author. All Adobe AIR applications require a digital signature, and can’t be installed without one. You can sign your application with a purchased digital certificate, create your own certificate, or prepare an Adobe AIRI file (an Adobe AIR intermediate file) that you’ll sign at a later time.

1. In the AIR Application and Installer Settings dialog box, click the Set button next to the Digital Signature option.
2. In the Digital Signature dialog box, do one of the following:
   - To sign an application with a pre-purchased digital certificate, click the Browse button, select the certificate, enter the corresponding password, and click OK.
   - To create your own self-signed digital certificate, click the Create button and complete the dialog box. The certificate Type option refers to the level of security: 1024-RSA uses a 1024-bit key (less secure), and 2048-RSA uses a 2048-bit key (more secure). When you’re finished click OK. Then enter the corresponding password in the Digital Signature dialog box and click OK.
• Select Prepare an AIRI package that will be signed later and click OK. This option lets you create an AIR Intermediate (AIRI) application without a digital signature. A user is not able to install the application, however, until you add a digital signature.

**About Timestamping**

When you sign an Adobe AIR application with a digital certificate, the packaging tool queries the server of a timestamp authority to obtain an independently verifiable date and time of signing. The timestamp obtained is embedded in the AIR file. As long as the signing certificate is valid at the time of signing, the AIR file can be installed, even after the certificate has expired. On the other hand, if no timestamp is obtained, the AIR file ceases to be installable when the certificate expires or is revoked.

By default, the Adobe AIR Extension for Dreamweaver obtains a timestamp when creating an Adobe AIR application. You can, however, turn timestamping off by deselecting the Timestamp option in the Digital Signature dialog box. (You might want to do this, for example, if a timestamping service is unavailable.) Adobe recommends that all publically distributed AIR files include a timestamp.

The default timestamp authority used by the AIR packaging tools is Geotrust. For more information on timestamping and digital certificates, see

**Editing associated AIR file types**

You can associate different file types with your Adobe AIR application. For example, if you want file types with an .avf extension to open in Adobe AIR when a user double-clicks them, you can add the .avf extension to your list of associated file types.

1 In the AIR Application and Installer Settings dialog box, click the Edit list button next to the Associated File Types option.

2 In the Associated File Types dialog box, do one of the following:
   • Select a file type and click the minus (-) button to delete the file type.
   • Click the plus (+) button to add a file type.

   If you click the plus button to add a file type, the File Type Settings dialog box appears. Complete the dialog box and click OK to close it.

   Following is a list of options:
   - **Name** specifies the name of the file type that appears in the Associated File Types list. This option is required, and can only include alphanumeric ASCII characters (a-z, A-Z, 0-9) and dots (for example, adobe.VideoFile). The name must start with a letter. The maximum length is 38 characters.
   - **Extension** specifies the extension of the file type. Do not include a preceding dot. This option is required, and can only include alphanumeric ASCII characters (a-z, A-Z, 0-9). The maximum length is 38 characters.
   - **Description** lets you specify an optional description for the file type.
   - **Content Type** specifies the MIME type or media type for the file (for example text/html, image/gif, and so on).
   - **Icon File Locations** lets you select custom images for the associated file types. (The default images are Adobe AIR images that come with the extension.)
Editing AIR application settings

You can edit the settings for your Adobe AIR application at any time.
❖ Select Site > AIR Application Settings and make your changes.

Previewing a web page in an AIR application

You can preview an HTML page in Dreamweaver as it would appear in an Adobe AIR application. Previewing is useful when you want to see what a web page will look like in the application without having to create the entire application.
❖ On the Document toolbar, click the Preview/Debug in Browser button, and then select Preview In AIR.

You can also press Ctrl+Shift+F12 (Windows) or Cmd+Shift+F12 (Macintosh).

Using AIR code hinting and code coloring

The Adobe AIR Extension for Dreamweaver also adds code hinting and code coloring for Adobe AIR language elements in Code view in Dreamweaver.
❖ Open an HTML or JavaScript file in Code view and enter Adobe AIR code.

*Note: The code hinting mechanism only works inside `<script>` tags, or in .js files.*

For more information on the Adobe AIR language elements, see the developer documentation in the rest of this guide.

Accessing the Adobe AIR documentation

The Adobe AIR extension adds a Help menu item in Dreamweaver that lets you access Developing AIR Applications with HTML and Ajax.
❖ Select Help > Adobe AIR Help.

*More Help topics*
“Create your first HTML-based AIR application with Dreamweaver” on page 15
Chapter 8: Creating an AIR application using the command line tools

The Adobe® AIR® command line tools allow you to test and package Adobe AIR applications. You can also use these tools in automated build processes. The command line tools are included in both the Adobe® Flex™ and AIR SDKs.

Using the AIR Debug Launcher (ADL)

Use the AIR Debug Launcher (ADL) to run both SWF-based and HTML-based applications during development. Using ADL, you can run an application without first packaging and installing it. By default, ADL uses a runtime included with the SDK, which means you do not have to install the runtime separately to use ADL.

ADL prints trace statements and run-time errors to the standard output, but does not support breakpoints or other debugging features. If you’re developing a SWF-based application, use the Flash Debugger for complex debugging issues. You can connect to the Flash Debugger by starting the debugger program before running your application with ADL.

Launching an application with ADL

Use the following syntax:

```
```

- **-runtime runtime-directory** Specifies the directory containing the runtime to use. If not specified, the runtime directory in the same SDK as the ADL program is used. If you move ADL out of its SDK folder, then you must specify the runtime directory. On Windows and Linux, specify the directory containing the Adobe AIR directory. On Mac OS X, specify the directory containing Adobe AIR.framework.

- **-pubid publisher-id** Assigns the specified value as the publisher ID of the AIR application for this run. Specifying a temporary publisher ID allows you to test features of an AIR application, such as communicating over a local connection, that use the publisher ID to help uniquely identify an application. As of AIR 1.5.3, you can also specify the publisher ID in the application descriptor file (and should not use this parameter).

  **Note:** As of AIR 1.5.3, a Publisher ID is no longer automatically computed and assigned to an AIR application. You can specify a publisher ID when creating an update to an existing AIR application, but new applications do not need and should not specify a publisher ID.

- **-nodebug** Turns off debugging support. If used, the application process cannot connect to the Flash debugger and dialogs for unhandled exceptions are suppressed. (However, trace statements still print to the console window.)

  Turning off debugging allows your application to run a little faster and also emulates the execution mode of an installed application more closely.

- **application.xml** The application descriptor file. See “Setting AIR application properties” on page 116.

- **root-directory** Specifies 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.

Printing trace statements
To print trace statements to the console used to run ADL, add trace statements to your code with the air.trace() function:

```javascript
trace("debug message");
air.trace("debug message");
```

In JavaScript, you can use the alert() and confirm() functions to display debugging messages from your application. In addition, the line numbers for syntax errors as well as any uncaught JavaScript exceptions are printed to the console.

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, "tick" and "tock":

```
adl myApp-app.xml -- tick tock
```

Run an application using a specific runtime:

```
adl -runtime /AIRSDK/runtime myApp-app.xml
```

Run an application without debugging support:

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

Connecting to the Flash Debugger (FDB)
To debug an HTML-based AIR application with the Flash Debugger, start an FDB session and then launch the application using ADL.

1. Start FDB. The FDB program can be found in the `bin` directory of the Flex SDK.
   - The console displays the FDB prompt: `<fdb>`
   - The Flex SDK is available from [http://opensource.adobe.com](http://opensource.adobe.com).

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.xml
   ```

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

5. Type: `continue [Enter]`

   If an AIR application is SWF-based, the debugger only controls the execution of ActionScript code. If the AIR application is HTML-based, then the debugger only controls the execution of JavaScript code.

   To run ADL without connecting to the debugger, include the `-nodebug` option:
adl myApp.xml -nodebug

For basic information on FDB commands, execute the help command:

<fdb>help [Enter]

For details on the FDB commands, see Using the command-line debugger commands in the Flex documentation.

**ADL exit and error codes**

The following table describes the exit codes printed by ADL:

<table>
<thead>
<tr>
<th>Exit code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful launch. ADL exits after the AIR application exits.</td>
</tr>
<tr>
<td>1</td>
<td>Successful invocation of an already running AIR application. ADL exits immediately.</td>
</tr>
<tr>
<td>2</td>
<td>Usage error. The arguments supplied to ADL are incorrect.</td>
</tr>
<tr>
<td>3</td>
<td>The runtime cannot be found.</td>
</tr>
<tr>
<td>4</td>
<td>The runtime cannot be started. Often, this occurs because the version or patch level specified in the application does not match the version or patch level of the runtime.</td>
</tr>
<tr>
<td>5</td>
<td>An error of unknown cause occurred.</td>
</tr>
<tr>
<td>6</td>
<td>The application descriptor file cannot be found.</td>
</tr>
<tr>
<td>7</td>
<td>The contents of the application descriptor are not valid. This error usually indicates that the XML is not well formed.</td>
</tr>
<tr>
<td>8</td>
<td>The main application content file (specified in the <code>&lt;content&gt;</code> element of the application descriptor file) cannot be found.</td>
</tr>
<tr>
<td>9</td>
<td>The main application content file is not a valid SWF or HTML file.</td>
</tr>
</tbody>
</table>

**Packaging an AIR installation file using the AIR Developer Tool (ADT)**

You create an AIR installation file for both your SWF-based and HTML-based AIR applications using the AIR Developer Tool (ADT). (If you are using the Adobe® AIR® Extension for Dreamweaver® to create your application, you can also use the Create AIR File command on the AIR Application and Installer Settings dialog to build the AIR package. See “Using the AIR Extension for Dreamweaver” on page 18.)

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 HTML development tools” on page 3 for information on configuring your system to run the ADT tool.

**Packaging an AIR installation file**

Every AIR application must, at a minimum, have an application descriptor file and a main SWF or HTML file. Any other installed application assets must be packaged in the AIR file as well.
All AIR package files must be signed using a digital certificate. The AIR installer uses the signature to verify that your application file has not been altered since you signed it. You can use a code signing certificate from a certification authority or a self-signed certificate. A certificate issued by a trusted certification authority provides users of your application some assurance of your identity as publisher. A self-signed certificate cannot be used to verify your identity as the signer. This drawback also weakens the assurance that the package hasn’t been altered, because a legitimate installation file could be substituted with a forgery before it reaches the user.

You can package and sign an AIR file in a single step using the ADT -package command. You can also create an intermediate, unsigned package with the -prepare command, and sign the intermediate package with the -sign command in a separate step.

When signing the installation package, ADT automatically contacts a time-stamp authority server to verify the time. The time-stamp information is included in the AIR file. An AIR file that includes a verified time stamp can be installed at any point in the future. If ADT cannot connect to the time-stamp server, then packaging is canceled. You can override the time-stamping option, but without a time stamp, an AIR application ceases to be installable after the certificate used to sign the installation file expires.

If you are creating a package to update an existing AIR application, the package must be signed with the same certificate as the original application. If the original certificate has been renewed or has expired within the last 180 days, or if you want to change to a new certificate, you can apply a migration signature. A migration signature involves signing the application AIR file with both the new and the old certificate. Use the -migrate command to apply the migration signature as described in “Signing an AIR file to change the application certificate” on page 35.

**Important:** There is a strict 180 day grace period for applying a migration signature after the original certificate expires. Without a migration signature, existing users must uninstall their existing application before installing your new version. The grace period only applies to applications that specify AIR version 1.5.3, or above, in the application descriptor namespace. There is no grace period when targeting earlier versions of the AIR runtime.

Prior to AIR 1.1, migration signatures were not supported. You must package an application with an SDK of version 1.1 or later to apply a migration signature.

**Publisher IDs**

As of AIR 1.5.3, publisher IDs are deprecated. New applications (originally published with AIR 1.5.3 or later) do not need and should not specify a publisher ID.

When updating applications published with earlier versions of AIR, you must specify the original publisher ID in the application descriptor file. Otherwise, the installed version of your application and the update version are treated as different applications. If you use a different ID or omit the publisherID tag, a user must uninstall the earlier version before installing the new version.

To determine the original publisher ID, find the publisherid file in the META-INF/AIR subdirectory where the original application is installed. The string within this file is the publisher ID. Your application descriptor must specify the AIR 1.5.3 runtime (or later) in the namespace declaration of the application descriptor file in order to specify the publisher ID manually.

For applications published prior to AIR 1.5.3 — or that are published with the AIR 1.5.3 SDK, while specifying an earlier version of AIR in the application descriptor namespace — a publisher ID is computed based on the signing certificate. This ID is used, along with the application ID, to determine the identity of an application. The publisher ID, when present, is used for the following purposes:

- Verifying that an AIR file is an update rather than a new application to install
- As part of the encryption key for the encrypted local store
- As part of the path for the application storage directory
• As part of the connection string for local connections
• As part of the identity string used to invoke an application with the AIR in-browser API
• As part of the OSID (used when creating custom install/uninstall programs)

Before AIR 1.5.3, the publisher ID of an application could change if you signed an application update with migration signature using a new or renewed certificate. When a publisher ID changes, the behavior of any AIR features relying on the ID also changes. For example, data in the existing encrypted local store can no longer be accessed and any Flash or AIR instances that create a local connection to the application must use the new ID in the connection string.

In AIR 1.5.3, or later, the publisher ID is not based on the signing certificate and is only assigned if the publisherID tag is included in the application descriptor. An application cannot be updated if the publisher ID specified for the update AIR package does not match its current publisher ID.

Package and sign an AIR file in one step
❖ Use the -package command with the following syntax (on a single command line):

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

**SIGNING_OPTIONS** The signing options identify the keystore containing the private key and certificate used to sign the AIR file. To sign an AIR application with a self-signed certificate generated by ADT, the options to use are:

- `storetype pkcs12 -keystore certificate.p12`

In this example, `certificate.p12` is the name of the keystore file. (ADT prompts you for the password since it is not supplied on the command line.) The signing options are fully described in “ADT command line signing options” on page 32.

- **air_file** The name of the AIR file that is created.
- **app_xml** The path to the application descriptor file. The path can be specified relative to the current directory or as an absolute path. (The application descriptor file is renamed as “application.xml” in the AIR file.)
- **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.) 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:** Wild cards cannot be used in the file_or_dir arguments following the -C option. (Command shells expand the wildcards before passing the arguments to ADT, which causes ADT to look for files in the wrong location.) You can, however, still use the dot character, “.”, to stand for the current directory. For example, “-C assets .” copies everything in the assets directory, including any subdirectories, to the root level of the application package.

- **-C dir** Changes the working directory to the value of 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 processes the files and directories 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. Therefore, specifying `-C release/bin lib/feature.swf` places the file `release/bin/lib/feature.swf` in the lib subdirectory of the root application folder.

- `e file dir` Places the specified file into the specified package directory.
Note: The <content> element of the application descriptor file must specify the final location of the main application file within the application package directory tree.

ADT Examples
Package specific application files in the current directory:

```
adt –package -storetype pkcs12 -keystore cert.p12 myApp.air myApp.xml myApp.html AIRAliases.js image.gif
```

Package all files and subdirectories in the current working directory:

```
```

Note: The keystore file contains the private key used to sign your application. Never include the signing certificate inside the AIR package! If you use wildcards in the ADT command, place the keystore file in a different location so that it is not included in the package. In this example the keystore file, cert.p12, resides in the parent directory.

Package only the main files and an images subdirectory:

```
adt –package -storetype pkcs12 -keystore cert.p12 myApp.air myApp.xml myApp.html AIRAliases.js images
```

Package an HTML-based application and all files in the HTML, scripts, and images subdirectories:

```
adt –package -storetype pkcs12 -keystore cert.p12 myApp.air myApp.xml index.html AIRAliases.js html scripts images
```

Package the application.xml file and main HTML file located in a working directory (src):

```
```

Package assets from more than one place in your build file system. In this example, the application assets are located in the following folders before packaging:

```
/devRoot
 /myApp
   /release
     /bin
       myApp.xml
       myApp.html
   /artwork
     /myApp
       /images
         image-1.png
         ...
         image-n.png
   /libraries
     /release
       /libs
         lib-1.js
         ...
         lib-n.js
       AIRAliases.js
```

Running the following ADT command from the /devRoot/myApp directory:

```
adt –package -storetype pkcs12 -keystore cert.p12 myApp.air release/bin/myApp.xml –C release/bin myApp.swf
 –C release/bin myApp.html
 –C ../artwork/myApp images
 –C ../libraries/release libs
```
Results in the following package structure:

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

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

```
java -jar {AIRSDK}/lib/ADT.jar -package -storetype pkcs12 -keystore cert.p12 myApp.air
```

```
java -jar {AIRSDK}/lib/ADT.jar -package -storetype pkcs12 -keystore cert.p12 myApp.air
```

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

```
java com.adobe.air.ADT -package -storetype pkcs12 -keystore cert.p12 myApp.air myApp.xml
```

```
java com.adobe.air.ADT -package -storetype pkcs12 -keystore cert.p12 myApp.air myApp.xml
```

**ADT error messages**

The following tables list the possible errors that can be reported by the ADT program and the probable causes:

**Application descriptor validation errors**
Creating an AIR application using the command line tools

### Application icon errors

<table>
<thead>
<tr>
<th>Error code</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Application descriptor cannot be parsed</td>
<td>Check the application descriptor file for XML syntax errors such as unclosed tags.</td>
</tr>
<tr>
<td>101</td>
<td>Namespace is missing</td>
<td>Add the missing namespace.</td>
</tr>
<tr>
<td>102</td>
<td>Invalid namespace</td>
<td>Check the namespace spelling.</td>
</tr>
<tr>
<td>103</td>
<td>Unexpected element or attribute</td>
<td>Remove offending elements and attributes. Custom values are not allowed in the descriptor file. Check the spelling of element and attribute names. Make sure that elements are placed within the correct parent element and that attributes are used with the correct elements.</td>
</tr>
<tr>
<td>104</td>
<td>Missing element or attribute</td>
<td>Add the required element or attribute.</td>
</tr>
<tr>
<td>105</td>
<td>Element or attribute contains an invalid value</td>
<td>Correct the offending value.</td>
</tr>
<tr>
<td>106</td>
<td>Illegal window attribute combination</td>
<td>Some window settings, such as transparency = true and systemChrome = standard cannot be used together. Change one of the incompatible settings.</td>
</tr>
<tr>
<td>107</td>
<td>Window minimum size is larger than the window maximum size</td>
<td>Change either the minimum or the maximum size setting.</td>
</tr>
</tbody>
</table>

See “Setting AIR application properties” on page 116 for more information about the namespaces, elements, attributes, and their valid values.

### Application file errors

<table>
<thead>
<tr>
<th>Error code</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Icon file cannot be opened</td>
<td>Check that the file exists at the specified path.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use another application to ensure that the file can be opened.</td>
</tr>
<tr>
<td>201</td>
<td>Icon is the wrong size</td>
<td>Icon size (in pixels) must match the XML tag. For example, given the application descriptor element: &lt;image32x32&gt;icon.png&lt;/image32x32&gt; The image in icon.png must be exactly 32x32 pixels.</td>
</tr>
<tr>
<td>202</td>
<td>Icon file contains an unsupported image format</td>
<td>Only the PNG format is supported. Convert images in other formats before packaging your application.</td>
</tr>
</tbody>
</table>
### Exit codes for other errors

<table>
<thead>
<tr>
<th>Exit code</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Usage error</td>
<td>Check the command line arguments for errors</td>
</tr>
<tr>
<td>5</td>
<td>Unknown error</td>
<td>This error indicates a situation that cannot be explained by common error conditions. Possible root causes include incompatibility between ADT and the Java Runtime Environment, corrupt ADT or JRE installations, and programming errors within ADT.</td>
</tr>
<tr>
<td>6</td>
<td>Could not write to output directory</td>
<td>Make sure that the specified (or implied) output directory is accessible and that the containing drive is has sufficient disk space.</td>
</tr>
<tr>
<td>7</td>
<td>Could not access certificate</td>
<td>Make sure that the path to the keystore is specified correctly. Check that the certificate within the keystore can be accessed. The Java 1.6 Keytool utility can be used to help troubleshoot certificate access issues.</td>
</tr>
<tr>
<td>8</td>
<td>Invalid certificate</td>
<td>The certificate file is malformed, modified, expired, or revoked.</td>
</tr>
<tr>
<td>9</td>
<td>Could not sign AIR file</td>
<td>Verify the signing options passed to ADT.</td>
</tr>
</tbody>
</table>
ADT command line signing options

ADT uses the Java Cryptography Architecture (JCA) to access private keys and certificates for signing AIR applications. The signing options identify the keystore and the private key and certificate within that keystore.

The keystore must include both the private key and the associated certificate chain. The certificate chain is used to establish the publisher ID for the application. If the signing certificate chains to a trusted certificate on a computer, then the common name of the certificate is displayed as the publisher name on the AIR installation dialog.

ADT requires that the certificate conform to the x509v3 standard (RFC3280) and include the Extended Key Usage extension with the proper values for code signing. Constraints within the certificate are respected and could preclude the use of some certificates for signing AIR applications.

Note: ADT uses the Java runtime environment proxy settings, when appropriate, for connecting to Internet resources for checking certificate revocation lists and obtaining time-stamps. If you encounter problems connecting to Internet resources when using ADT and your network requires specific proxy settings, you may need to configure the JRE proxy settings.

Specifying AIR signing options

To specify the ADT signing options for the -package and -prepare commands, use the following syntax:

```
[-alias aliasName] [-storetype type] [-keystore path] [-storepass password1] [-keypass password2] [-providerName className] [-tsa url]
```

- **-alias aliasName** —The alias of a key in the keystore. Specifying an alias is not necessary when a keystore only contains a single certificate. If no alias is specified, ADT uses the first key in the keystore.

Not all keystore management applications allow an alias to be assigned to certificates. When using the Windows system keystore for example, use the distinguished name of the certificate as the alias. You can use the Java Keytool utility to list the available certificates so that you can determine the alias. For example, running the command:

```
keytool -list -storetype Windows-MY
```

produces output like the following for a certificate:

```
CN=TestingCert,OU=QE,O=Adobe,C=US, PrivateKeyEntry,
```

To reference this certificate on the ADT command line, set the alias to:
On Mac OS X, the alias of a certificate in the Keychain is the name displayed in the Keychain Access application.

**-storetype type** — The type of keystore, determined by the keystore implementation. The default keystore implementation included with most installations of Java supports the JKS and PKCS12 types. Java 5.0 includes support for the PKCS11 type, for accessing keystores on hardware tokens, and Keychain type, for accessing the Mac OS X keychain. Java 6.0 includes support for the MSCAPI type (on Windows). If other JCA providers have been installed and configured, additional keystore types might be available. If no keystore type is specified, the default type for the default JCA provider is used.

**-keystore path** — The path to the keystore file for file-based store types.

**-storepass password1** — The password required to access the keystore. If not specified, ADT prompts for the password.

**-keypass password2** — The password required to access the private key that is used to sign the AIR application. If not specified, ADT prompts for the password.

**-providerName className** — The JCA provider for the specified keystore type. If not specified, then ADT uses the default provider for that type of keystore.

**-tsa url** — Specifies the URL of an RFC3161-compliant timestamp server to time-stamp the digital signature. If no URL is specified, a default time-stamp server provided by Geotrust is used. When the signature of an AIR application is time-stamped, the application can still be installed after the signing certificate expires, because the timestamp verifies that the certificate was valid at the time of signing.

If ADT cannot connect to the time-stamp server, then signing is canceled and no package is produced. Specify -tsa none to disable time-stamping. However, an AIR application packaged without a timestamp ceases to be installable after the signing certificate expires.

**Note:** The signing options are like the equivalent options of the Java Keytool utility. You can use the Keytool utility to examine and manage keystores on Windows. The Apple® security utility can also be used for this purpose on Mac OS X.

### Signing option examples

**Signing with a .p12 file:**

```
-storetype pkcs12 -keystore cert.p12
```

**Signing with the default Java keystore:**

```
-alias AIRcert -storetype jks
```

**Signing with a specific Java keystore:**

```
-alias AIRcert -storetype jks -keystore certStore.keystore
```
Signing with the Mac OS X keychain:
-alias AIRcert -storetype KeychainStore -providerName Apple

Signing with the Windows system keystore:
-alias cn=AIRCert -storeype Windows-MY

Signing with a hardware token (refer to the token manufacturer’s instructions on configuring Java to use the token and for the correct providerName value):
-alias AIRCert -storetype pkcs11 -providerName tokenProviderName

Signing without embedding a timestamp:
-storetype pkcs12 -keystore cert.p12 -tsa none

Creating an unsigned AIR intermediate file with ADT

Use the -prepare command to create an unsigned AIR intermediate file. An AIR intermediate file must be signed with the ADT -sign command to produce a valid AIR installation file.

The -prepare command takes the same flags and parameters as the -package command (except for the signing options). The only difference is that the output file is not signed. The intermediate file is generated with the filename extension: .airi.

To sign an AIR intermediate file, use the ADT -sign command. (See Signing an AIR intermediate file with ADT.)

ADT Example
adt –prepare unsignedMyApp.airi myApp.xml myApp.swf components.swc

adt –prepare unsignedMyApp.airi myApp.xml myApp.html AIRAliases.js image.gif

Signing an AIR intermediate file with ADT

To sign an AIR intermediate file with ADT, use the -sign command. The sign command only works with AIR intermediate files (extension .airi). An AIR file cannot be signed a second time.

To create an AIR intermediate file, use the adt -prepare command. (See “Creating an unsigned AIR intermediate file with ADT” on page 34.)

Sign an AIRI file
❖ Use the ADT -sign command with following syntax:
   adt -sign SIGNING_OPTIONsairi_fileair_file

   SIGNING_OPTIONS The signing options identify the private key and certificate with which to sign the AIR file. These options are described in “ADT command line signing options” on page 32.

   airi_file The path to the unsigned AIR intermediate file to be signed.

   air_file The name of the AIR file to be created.

ADT Example
adt -sign -storetype pkcs12 -keystore cert.p12 unsignedMyApp.airi myApp.air

For more information, see “Digitally signing an AIR file” on page 369.
Signing an AIR file to change the application certificate

To publish an update for an existing AIR application while using a new or renewed signing certificate, use the ADT -migrate command to apply a certificate migration signature. A migration signature is a second signature applied to an AIR file using the original certificate. The migration signature validates that an application update was produced by the owners of the original certificate.

In order to apply a migration signature, the original certificate must still be valid or have expired within the last 180 days. Once the certificate has expired and the 180 day grace period has elapsed, a migration signature cannot be applied. Users of your application will have to uninstall the existing version before they can install the updated version. The migration signature is time stamped, by default, so AIR updates signed with a migration signature will remain valid even after the certificate expires.

Note: The 180 day grace period only applies to applications specifying AIR version 1.5.3, or higher, in the application descriptor namespace.

To migrate the application to use a new or renewed certificate:

1. Create an update to your application
2. Package and sign the update AIR file with the new certificate
3. Sign the AIR file again with the original certificate using the -migrate command

An AIR file signed with the -migrate command can be used both to install a new version of the application and to update any previous versions, including those signed with the old certificate.

Note: When updating an application published for a version of AIR earlier than 1.5.3, you must specify the original publisher ID in the application descriptor. Otherwise, users of your application must uninstall the earlier version before installing the update.

Migrate an AIR application to use a new certificate

❖ Use the ADT -migrate command with following syntax:

```
adt -migrate SIGNING_OPTIONS air_file_in air_file_out
```

**SIGNING_OPTIONS** The signing options identify the private key and certificate with which to sign the AIR file. These options must identify the original signing certificate and are described in “ADT command line signing options” on page 32.

- **air_file_in** The AIR file for the update, signed with the new certificate.
- **air_file_out** The AIR file to create.

**ADT Example**

```
adt -migrate -storetype pkcs12 -keystore cert.p12 myApp.air myApp.air
```

For more information, see “Digitally signing an AIR file” on page 369.

**Note:** The -migrate command was added to ADT in the AIR 1.1 release.
Creating a self-signed certificate with ADT

Self-signed certificates allow you to produce a valid AIR installation file, but only provide limited security assurances to your users since the authenticity of self-signed certificates cannot be verified. When a self-signed AIR file is installed, the publisher information is displayed to the user as Unknown. A certificate generated by ADT is valid for five years.

If you create an update for an AIR application that was signed with a self-generated certificate, you must use the same certificate to sign both the original and update AIR files. The certificates that ADT produces are always unique, even if the same parameters are used. Thus, if you want to self-sign updates with an ADT-generated certificate, preserve the original certificate in a safe location. In addition, you will be unable to produce an updated AIR file after the original ADT-generated certificate expires. (You can publish new applications with a different certificate, but not new versions of the same application.)

Important: Because of the limitations of self-signed certificates, Adobe strongly recommends using a commercial certificate issued by a reputable certification authority for signing publicly released AIR applications.

The certificate and associated private key generated by ADT are stored in a PKCS12-type keystore file. The password specified is set on the key itself, not the keystore.

Generating a digital ID certificate for self-signing AIR files

❖ Use the ADT -certificate command (on a single command line):

```
adt -certificate -cn name [-ou org_unit] [-o org_name] [-c country] key_type pfx_file password
```

- **-cn name** The string assigned as the common name of the new certificate.
- **-ou org_unit** A string assigned as the organizational unit issuing the certificate. (Optional.)
- **-o org_name** A string assigned as the organization issuing the certificate. (Optional.)
- **-c country** A two-letter ISO-3166 country code. A certificate is not generated if an invalid code is supplied. (Optional.)
- **key_type** The type of key to use for the certificate, either “1024-RSA” or “2048-RSA”.
- **pfx_file** The path for the certificate file to be generated.
- **password** The password for the new certificate. The password is required when signing AIR files with this certificate.

Certificate generation examples

```
adt -certificate -cn SelfSign -ou QE -o "Example, Co" -c US 2048-RSA newcert.p12 33#wnetx3tl
adt -certificate -cn ADigitalID 1024-RSA SigningCert.p12 33#wnetx3tl
```

To use these certificates to sign AIR files, you use the following signing options with the ADT -package or -prepare commands:

```
-storetype pkcs12 -keystore newcert.p12 -keypass 33#wnetx3tl
-storetype pkcs12 -keystore SigningCert.p12 -keypass 33#wnetx3tl
```

Using Apache Ant with the SDK tools

This topic provides examples of using the Apache Ant build tool to 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 simple project structure is used with all files stored in a single directory.

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:

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

The second set of properties is project specific. These properties assume a naming convention in which the application descriptor and AIR files are named based on the root source file. Other conventions are easily supported.

```xml
<property name="APP_NAME" value="ExampleApplication"/>
<property name="APP_ROOT" value="."/>
<property name="APP_DESCRIPTOR" value="${APP_ROOT}/${APP_NAME}-app.xml"/>
<property name="AIR_NAME" value="${APP_NAME}.air"/>
<property name="STORETYPE" value="pkcs12"/>
<property name="KEYSTORE" value="ExampleCert.p12"/>
```

Invoking ADL to test an application

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

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

Invoking ADT to package an application

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

```xml
<target name="package">
  <java jar="${ADT.JAR}" fork="true" failonerror="true">
    <arg value="-package"/>
    <arg value="-storetype"/>
    <arg value="-keystore"/>
    <arg value="-storetype"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
    <arg value="-keystore"/>
Using Ant for more complex projects

The directory structure of a typical application is more complex than 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.

release This sample project stores application source files and other assets like icon files within a src directory. The build script creates a release directory to store the final AIR package.

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

Testing The second argument passed to ADL specifies the root directory of the AIR application. To specify the application root directory, the following line is added to the testing task:

    <arg value="$\{debug\}"/>

Packaging Packaging files from subdirectories that should not be part of the final package structure requires using the -C directive to change the ADT working directory. When you use the -C directive, files and directories in the new working directory are copied to the root level of the AIR package file. Thus, -C build file.png copies file.png to the root of the application directory. Likewise, -C assets icons copies the icon folder to the root level, and copies all the files and directories within the icons folder as well. For example, the following sequence of lines in the package task adds the icons directory directly to the root level of the application package file:

    <arg value="-C"/>
    <arg value="$\{assets\}"/>
    <arg value="icons"/>

Note: If you need to move many resources and assets into different relative locations, it is typically easier to marshall them into a temporary directory using Ant tasks than it is to build a complex argument list for ADT. Once your resources are organized, a simple ADT argument list can be used to package them.

<pre>
<project>
    <!-- SDK properties -->
    <property name="SDK_HOME" value="C:/AIRSDK"/>
    <property name="ADL" value="$\{SDK_HOME}/bin/adl.exe"/>
    <property name="ADT.JAR" value="$\{SDK_HOME}/lib/adt.jar"/>
    <!-- Project properties -->
    <property name="PROJ_ROOT_DIR" value="."/>
    <property name="APP_NAME" value="ExampleApplication"/>
    <property name="APP_ROOT_DIR" value="$\{PROJ_ROOT_DIR}/src/html"/>
    <property name="APP_ROOT_FILE" value="$\{APP_NAME}.html"/>
    <property name="APP_DESCRIPTOR" value="$\{PROJ_ROOT_DIR}/${APP_NAME}-app.xml"/>
    <property name="AIR_NAME" value="$\{APP_NAME}.air"/>
    <property name="release" location="$\{PROJ_ROOT_DIR}/release"/>
    <property name="assets" location="$\{PROJ_ROOT_DIR}/src/assets"/>
    <property name="STORETYPE" value="pkcs12"/>
    <property name="KEYSTORE" value="ExampleCert.p12"/>

    <target name="init" depends="clean">
        <mkdir dir="$\{release\}"/>
    </target>

    <target name="test">
        <exec executable="$\{ADL}\">
            <arg value="$\{APP_DESCRIPTOR\}"/>
            <arg value="$\{APP_ROOT_DIR\}"/>
        </exec>
    </target>
</project>
</pre>
DEVELOPING ADOBE AIR 1.5 APPLICATIONS WITH HTML AND AJAX

Creating an AIR application using the command line tools

```xml
</exec>
</target>

<target name="package" depends="init">
  <java jar="${ADT.JAR}" fork="true" failonerror="true">
    <arg value="-package"/>
    <arg value="-storetype"/>
    <arg value="${STORETYPE}"/>
    <arg value="-keystore"/>
    <arg value="${KEYSTORE}"/>
    <arg value="${release}/${AIR_NAME}"/>
    <arg value="${APP_DESCRIPTOR}"/>
    <arg value="-C"/>
    <arg value="${APP_ROOT_DIR}"/>
    <arg value="${APP_ROOT_FILE}"/>
    <arg value="-C"/>
    <arg value="${assets}"/>
    <arg value="icons"/>
  </java>
</target>

<target name="clean" description="clean up">
  <delete dir="${release}"/>
</target>
</project>
```
Chapter 9: Debugging with the AIR HTML Introspector

The Adobe® AIR® SDK includes an AIRIntrospector.js JavaScript file that you can include in your application to help debug HTML-based applications.

About the AIR Introspector

The Adobe AIR HTML/JavaScript Application Introspector (called the AIR HTML Introspector) provides useful features to assist HTML-based application development and debugging:

- It includes an introspector tool that allows you to point to a user interface element in the application and see its markup and DOM properties.
- It includes a console for sending objects references for introspection, and you can adjust property values and execute JavaScript code. You can also serialize objects to the console, which limits you from editing the data. You can also copy and save text from the console.
- It includes a tree view for DOM properties and functions.
- It lets you edit the attributes and text nodes for DOM elements.
- It lists links, CSS styles, images, and JavaScript files loaded in your application.
- It lets you view to the initial HTML source and the current markup source for the user interface.
- It lets you access files in the application directory. (This feature is only available for the AIR HTML Introspector console opened for application sandbox. Not available for the consoles open for non-application sandbox content.)
- It includes a viewer for XMLHttpRequest objects and their properties, including `responseText` and `responseXML` properties (when available).
- You can search for matching text in the source code and files.

Loading the AIR Introspector code

The AIR Introspector code is included in a JavaScript file, AIRIntrospector.js, that is included in the frameworks directory of the AIR SDK. To use the AIR Introspector in your application, copy the AIRIntrospector.js to your application project directory and load the file via a script tag in the main HTML file in your application:

```html
<script type="text/javascript" src="AIRIntrospector.js"></script>
```

Also include the file in every HTML file that corresponds to different native windows in your application.

**Important:** Include the AIRIntrospector.js file only when developing and debugging the application. Remove it in the packaged AIR application that you distribute.

The AIRIntrospector.js file defines a class, Console, which you can access from JavaScript code by calling `air.Introspector.Console`.

**Note:** Code using the AIR Introspector must be in the application security sandbox (in a file in the application directory).
Inspecting an object in the Console tab

The Console class defines five methods: log(), warn(), info(), error(), and dump().

The log(), warn(), info(), and error() methods all let you send an object to the Console tab. The most basic of these methods is the log() method. The following code sends a simple object, represented by the test variable, to the Console tab:

```javascript
var test = "hello";
air.Introspector.Console.log(test);
```

However, it is more useful to send a complex object to the Console tab. For example, the following HTML page includes a button (btn1) that calls a function that sends the button object itself to the Console tab:

```html
<html>
  <head>
    <script type="text/javascript" src="scripts/AIRIntrospector.js"></script>
    <script type="text/javascript">
      function logBtn()
      {
        var button1 = document.getElementById("btn1");
        air.Introspector.Console.log(button1);
      }
    </script>
  </head>
  <body>
    <p>Click to view the button object in the Console.</p>
    <input type="button" id="btn1" onclick="logBtn()"
          value="Log" />
  </body>
</html>
```
When you click the button, the Console tab displays the btn1 object, and you can expand the tree view of the object to inspect its properties:

![AIR Introspector](image)

You can edit a property of the object by clicking the listing to the right of the property name and modifying the text listing.

The `info()`, `error()`, and `warn()` methods are just like the `log()` method. However, when you call these methods, the Console displays an icon at the beginning of the line:

<table>
<thead>
<tr>
<th>Method</th>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>info()</code></td>
<td>![info icon]</td>
</tr>
<tr>
<td><code>error()</code></td>
<td>![error icon]</td>
</tr>
<tr>
<td><code>warn()</code></td>
<td>![warning icon]</td>
</tr>
</tbody>
</table>

The `log()`, `warn()`, `info()`, and `error()` methods send a reference only to an actual object, so the properties available are the ones at the moment of viewing. If you want to serialize the actual object, use the `dump()` method. The method has two parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dumpObject</code></td>
<td>The object to be serialized.</td>
</tr>
<tr>
<td><code>levels</code></td>
<td>The maximum number of levels to be examined in the object tree (in addition to the root level). The default value is 1 (meaning that one level beyond the root level of the tree is shown). This parameter is optional.</td>
</tr>
</tbody>
</table>

Calling the `dump()` method serializes an object before sending it to the Console tab, so that you cannot edit the objects properties. For example, consider the following code:
var testObject = new Object();
testObject.foo = "foo";
testObject.bar = 234;
air.Introspector.Console.dump(testObject);

When you execute this code, the Console displays the testObject object and its properties, but you cannot edit the property values in the Console.

### Configuring the AIR Introspector

You can configure the console by setting properties of the global AIRIntrospectorConfig variable. For example, the following JavaScript code configures the AIR Introspector to wrap columns at 100 characters:

```javascript
var AIRIntrospectorConfig = new Object();
AIRIntrospectorConfig.wrapColumns = 100;
```

Be sure to set the properties of the AIRIntrospectorConfig variable before loading the AIRIntrospector.js file (via a script tag).

There are eight properties of the AIRIntrospectorConfig variable:

<table>
<thead>
<tr>
<th>Property</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>closeIntrospectorOnExit</td>
<td>true</td>
<td>Sets the Inspector window to close when all other windows of the application are closed.</td>
</tr>
<tr>
<td>debuggerKey</td>
<td>123 (the F12 key)</td>
<td>The key code for the keyboard shortcut to show and hide the AIR Introspector window.</td>
</tr>
<tr>
<td>debugRuntimeObjects</td>
<td>true</td>
<td>Sets the Introspector to expand runtime objects in addition to objects defined in JavaScript.</td>
</tr>
<tr>
<td>flashTabLabels</td>
<td>true</td>
<td>Sets the Console and XMLHttpRequest tabs to flash, indicating when a change occurs in them (for example, when text is logged in these tabs).</td>
</tr>
<tr>
<td>introspectorKey</td>
<td>122 (the F11 key)</td>
<td>The key code for the keyboard shortcut to open the Inspect panel.</td>
</tr>
<tr>
<td>showTimestamp</td>
<td>true</td>
<td>Sets the Console tab to display timestamps at the beginning of each line.</td>
</tr>
<tr>
<td>showSender</td>
<td>true</td>
<td>Sets the Console tab to display information on the object sending the message at the beginning of each line.</td>
</tr>
<tr>
<td>wrapColumns</td>
<td>2000</td>
<td>The number of columns at which source files are wrapped.</td>
</tr>
</tbody>
</table>

### AIR Introspector interface

To open the AIR introspector window when debugging the application, press the F12 key or call one of the methods of the Console class (see “Inspecting an object in the Console tab” on page 41). You can configure the hot key to be a key other than the F12 key; see “Configuring the AIR Introspector” on page 43.
The AIR Introspector window has six tabs—Console, HTML, DOM, Assets, Source, and XHR—as shown in the following illustration:

The Console tab
The Console tab displays values of properties passed as parameters to one of the methods of the air.Introspector.Console class. For details, see "Inspecting an object in the Console tab" on page 41.

- To clear the console, right-click the text and select Clear Console.
- To save text in the Console tab to a file, right-click the Console tab and select Save Console To File.
To save text in the Console tab to the clipboard, right-click the Console tab and select Save Console To Clipboard.

To copy only selected text to the clipboard, right-click the text and select Copy.

To save text in the Console class to a file, right-click the Console tab and select Save Console To File.

To search for matching text displayed in the tab, click CTRL+F on Windows or Command+F on Mac OS. (Tree nodes that are not visible are not searched.)

The HTML tab

The HTML tab lets you view the entire HTML DOM in a tree structure. Click an element to view its properties on the right-hand side of the tab. Click the + and - icons to expand and collapse a node in the tree.

You can edit any attribute or text element in the HTML tab and the edited value is reflected in the application.

Click the Inspect button (to the left of the list of tabs in the AIR Introspector window). You can click any element on the HTML page of the main window and the associated DOM object is displayed in the HTML tab. When the main window has focus, you can also press the keyboard shortcut to toggle the Inspect button on and off. The keyboard shortcut is F11 by default. You can configure the keyboard shortcut to be a key other than the F11 key; see “Configuring the AIR Introspector” on page 43.

Click the Refresh Active Window button (at the top of the AIR Introspector window) to refresh the data displayed in the HTML tab.

Click CTRL+F on Windows or Command+F on Mac OS to search for matching text displayed in the tab. (Tree nodes that are not visible are not searched.)
The DOM tab

The DOM tab shows the window object in a tree structure. You can edit any string and numeric properties and the edited value is reflected in the application.

Click the Refresh Active Window button (at the top of the AIR Introspector window) to refresh the data displayed in the DOM tab.

Click CTRL+F on Windows or Command+F on Mac OS to search for matching text displayed in the tab. (Tree nodes that are not visible are not searched.)
The Assets tab

The Assets tab lets you check the links, images, CSS, and JavaScript files loaded in the native window. Expanding one of these nodes shows the content of the file or displays the actual image used.

Click the Refresh Active Window button (at the top of the AIR Introspector window) to refresh the data displayed in the Assets tab.

Click CTRL+F on Windows or Command+F on Mac OS to search for matching text displayed in the tab. (Tree nodes that are not visible are not searched.)

The Source tab

The Source tab includes three sections:

- Actual source—Shows the HTML source of the page loaded as the root content when the application started.
- Parsed source—Shows the current markup that makes up the application UI, which can be different from the actual source, since the application generates markup code on the fly using Ajax techniques.
• Application files—Lists the files in the application directory. This listing is only available for the AIR Introspector when launched from content in the application security sandbox. In this section, you can view the content of text files or view images.

Click the Refresh Active Window button (at the top of the AIR Introspector window) to refresh the data displayed in the Source tab.

Click CTRL+F on Windows or Command+F on Mac OS to search for matching text displayed in the tab. (Tree nodes that are not visible are not searched.)
The XHR tab

The XHR tab intercepts all XMLHttpRequest communication in the application and logs the information. This lets you view the XMLHttpRequest properties including `responseText` and `responseXML` (when available) in a tree view.

Click CTRL+F on Windows or Command+F on Mac OS to search for matching text displayed in the tab. (Tree nodes that are not visible are not searched.)

Using the AIR Introspector with content in a non-application sandbox

You can load content from the application directory into an iframe or frame that is mapped to a non-application sandbox (see “HTML security” on page 106). You can use the AIR introspector with such content, but observe the following rules:

- The AIRIntrospector.js file must be included in both the application sandbox and in the non-application sandbox (the iframe) content.
- Do not overwrite the `parentSandboxBridge` property; the AIR Introspector code uses this property. Add properties as needed. So instead of writing the following:

  ```javascript
  parentSandboxBridge = mytrace: function(str) {runtime.trace(str)} ;
  ```

  Use syntax such as the following:

  ```javascript
  parentSandboxBridge.mytrace = function(str) {runtime.trace(str)} ;
  ```
- From the non-application sandbox content, you cannot open the AIR Introspector by pressing the F12 key or by calling one of methods in the `air.Introspector.Console` class. You can open the Introspector window only by clicking the Open Introspector button. The button is added by default at the upper-right corner of the iframe or frame. (Due to security restrictions imposed to non-application sandbox content, a new window can be opened only as a result of a user gesture, such as clicking a button.)
- You can open separate AIR Introspector windows for the application sandbox and for the non-application sandbox. You can differentiate the two using the title displayed in the AIR Introspector windows.
- The Source tab doesn’t display application files when the AIR Introspector is run from a non-application sandbox.
- The AIR Introspector can only look at code in the sandbox from which it was opened.
Chapter 10: Programming in HTML and JavaScript

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

Creating an HTML-based AIR application

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

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

The basic steps to creating an AIR application are:

1. Create the application descriptor file. The content element identifies the root page of the application, which is loaded automatically when your application is launched. (See "Setting AIR application properties" on page 116 for more information.)

2. Create the application pages and code.

3. Test the application using the ADL utility or Dreamweaver.

4. Package the application into an AIR installation file with the ADT utility or Dreamweaver.

For a walk-through of these steps, see "Creating your first HTML-based AIR application with the AIR SDK" on page 10 or "Create your first HTML-based AIR application with Dreamweaver" on page 15.
An example application and security implications

In “Creating your first HTML-based AIR application with the AIR SDK” on page 10, the Hello World example is intentionally simple. It only calls one AIR-specific API (the `air.trace()` method). The following HTML code uses a more advanced AIR API; it uses the filesystem APIs to list the files and directories in the user’s desktop directory.

Here’s the HTML code for the application:

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

You also must set up an application descriptor file and test the application using the AIR Debug Launcher (ADL) application. (See “Creating your first HTML-based AIR application with the AIR SDK” on page 10).

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

The `getDesktopFileList()` method uses the File class, which is defined in the runtime APIs. The first `<script>` tag in the application loads the `AIRAliases.js` file (supplied with the AIR SDK), which lets you easily access the AIR APIs. (For example, the example code accesses the AIR File class using the syntax `air.File`.) For details, see “Using the `AIRAliases.js` file” on page 58.
The `File.desktopDirectory` property is a `File` object (a type of object defined by the runtime). A `File` object is a reference to a file or directory on the user’s computer. The `File.desktopDirectory` property is a reference to the user’s desktop directory. The `getDirectoryListing()` method is defined for any `File` object and returns an array of `File` objects. The `File.desktopDirectory.getDirectoryListing()` method returns an array of `File` objects representing files and directories on the user’s desktop.

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

### Important security rules when using HTML in AIR applications

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

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

To properly structure content to work in an AIR application, you must take the rules for the security restrictions on content from different sources into account. Content from different sources is placed in separate security classifications, called sandboxes (see “Sandboxes” on page 104). By default, content installed with the application is installed in a sandbox known as the application sandbox, and this grants it access to the AIR APIs. The application sandbox is generally the most secure sandbox, with restrictions designed to prevent the execution of untrusted code.

The runtime allows you to load content installed with your application into a sandbox other than the application sandbox. Content in non-application sandboxes operates in a security environment similar to that of a typical web browser. For example, code in non-application sandboxes can use `eval()` and related methods (but at the same time is not allowed to access the AIR APIs). The runtime includes ways to have content in different sandboxes communicate securely (without exposing AIR APIs to non-application content, for example). For details, see “Cross-scripting content in different security sandboxes” on page 63.

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

To avoid this error, follow the coding practices described in the next section, “Avoiding security-related JavaScript errors” on page 53.

For more information, see “HTML security” on page 106.

### Avoiding security-related JavaScript errors

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

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

- `eval()` function
- `setTimeout()` and `setInterval()`
- Function constructor

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

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

**Note:** In some restricted cases, evaluation of strings is permitted. See “Code restrictions for content in different sandboxes” on page 108 for more information.

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

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

Mapping application content to a different sandbox

In most cases, you can rewrite or restructure an application to avoid security-related JavaScript errors. However, when rewriting or restructuring is not possible, you can load the application content into a different sandbox using the technique described in “Loading application content into a non-application sandbox” on page 64. If that content also must access AIR APIs, you can create a sandbox bridge, as described in “Setting up a sandbox bridge interface” on page 65.

`eval()` function

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

Assigning properties to an object

Instead of parsing a string to build the property accessor:

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

access properties with bracket notation:

```javascript
  obj[propName] = val;
```
Creating a function with variables available in context

Replace statements such as the following:

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

with:

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

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

Consider a hypothetical JavaScript class defined with the following code:

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

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

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

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

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

To create the instance, use:
try{
    var Parser = getter(constructorClassName);
    var a = new Parser();
}catch(e){
    alert(e);
}

setTimeout() and setInterval()

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

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

with:

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

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

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

with the following:

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

Function constructor

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

javascript: URLs

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

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

with:

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

Event callbacks assigned through onevent attributes in innerHTML and outerHTML statements

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

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

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

Replace statements such as:

```
document.getElementById('container').innerHTML =
    '<a href="#" onclick="code()">Click Me</a>';
```
Loading JavaScript files from outside the application installation directory

Loading script files from outside the application sandbox is not permitted. No security error is generated. All script files that run in the application sandbox must be installed in the application directory. To use external scripts in a page, you must map the page to a different sandbox. See “Loading application content into a non-application sandbox” on page 64.

document.write() and document.writeln()

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

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

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

Dynamically created script elements

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

Accessing AIR API classes from JavaScript

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

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

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

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

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

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

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

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

**Note:** For more information, see “ActionScript 3.0 classes, packages, and namespaces” on page 128 and “ActionScript basics for JavaScript developers” on page 126.

### Using the AIRAliases.js file

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

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

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

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

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

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

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

```html
<script src="AIRAliases.js"></script>
```
Adjust the path in the src reference, as needed.

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

### About URLs in AIR

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

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

For more information about using URL schemes in AIR, see “Using AIR URL schemes in URLs” on page 344.

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

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

For information about URLRequest objects see “URL requests and networking” on page 342.

### Embedding SWF content in HTML

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

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

The following example illustrates the use of the HTML object tag to display a SWF file within HTML content. The SWF file is loaded from the application directory, but you can use any of the URL schemes supported by AIR. (The location from which the SWF file is loaded determines the security sandbox in which AIR places the content.)

```html
<object type="application/x-shockwave-flash" width="100%" height="100%">
  <param name="movie" value="/SWFFile.swf"></param>
</object>
```
You can also use a script to load content dynamically. The following example creates an object node to display the SWF file specified in the urlString parameter. The example adds the node as a child of the page element with the ID specified by the elementID parameter:

```html
<script>
function showSWF(urlString, elementID){
    var displayContainer = document.getElementById(elementID);
    displayContainer.appendChild(createSWFObject(urlString,650,650));
}

function createSWFObject(urlString, width, height){
    var SWFObject = document.createElement("object");
    SWFObject.setAttribute("type","application/x-shockwave-flash");
    SWFObject.setAttribute("width","100%"ера");
    SWFObject.setAttribute("height","100%"ера");
    var movieParam = document.createElement("param");
    movieParam.setAttribute("name","movie");
    movieParam.setAttribute("value",urlString);
    SWFObject.appendChild(movieParam);
    return SWFObject;
}
</script>

Using ActionScript libraries within an HTML page

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

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

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

If the SWF content is compiled as a Flash Player 10 or AIR 1.5 SWF, you must set the XML namespace of the application descriptor file to the AIR 1.5 namespace. For more information, see "Defining properties in the application descriptor file" on page 117.

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

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

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

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

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

**Note:** To compile an ActionScript SWF library for use as part of an HTML page in AIR, use the acompc compiler. The acompc utility is part of the Flex 3 SDK and is described in the Flex 3 SDK documentation.
Accessing the HTML DOM and JavaScript objects from an imported ActionScript file

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

For example, consider the following HTML page:

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

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

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

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

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

```actionscript
public function modifyDOM(window:*):void {
    window.document.getElementById("p1").innerHTML = "Bye";
    window.foo = 66;
}
```
Converting Date and RegExp objects

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

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

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

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

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

Manipulating an HTML stylesheet from ActionScript

Once the HTMLLoader 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
<html>
<style>
.style1A { font-family:Arial; font-size:12px }
.style1B { font-family:Arial; font-size:24px }
</style>
<body>
<p class="style1A">
   Style 1A
</p>
<p class="style1B">
   Style 1B
</p>
<p class="style2">
   Style 2
</p>
</body>
</html>
```

After an HTMLLoader 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:HTMLLoader = new HTMLLoader();
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

Keep in mind that code can add styles to the page after the HTMLLoader object dispatches the complete event.

Cross-scripting content in different security sandboxes

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

AIR security sandboxes and JavaScript code

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

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

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

At the same time, application content mapped to another sandbox loses its access to the AIR APIs, so the sandbox mapping technique cannot be used to expose AIR functionality to code executed outside the application sandbox.
Another cross-scripting technique lets you create an interface called a sandbox bridge between content in a non-application sandbox and its parent document in the application sandbox. The bridge allows the child content to access properties and methods defined by the parent, the parent to access properties and methods defined by the child, or both.

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

For more information, see “HTML frame and iframe elements” on page 77, “HTML security” on page 106, and “The XMLHttpRequest object” on page 71.

**Loading application content into a non-application sandbox**

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

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

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

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

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

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

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

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

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

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

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

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

Establishing a child sandbox bridge

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

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

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

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

Establishing a parent sandbox bridge

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

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

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

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

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

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

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

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

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

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

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

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

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

If you use the HTMLLoader class createRootWindow() function to create a window, the new window is not a child of the document from which createRootWindow() is called. Thus, you cannot create a sandbox bridge from the calling window to non-application content loaded into the new window. Instead, you must use load an intermediate page in the new window that itself uses frames to load the child document. You can then establish the bridge from the parent document of the new window to the child document loaded into the frame.
Chapter 11: About the HTML environment

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

Important: New versions of the Adobe AIR runtime may include updated versions of WebKit. A WebKit update in a new version of AIR may result in unexpected changes in a deployed AIR application. These changes may affect the behavior or appearance of HTML content in an application. For example, improvements or corrections in WebKit rendering may change the layout of elements in an application’s user interface. For this reason, it is highly recommended that you provide an update mechanism in your application. Should you need to update your application due to a change in the WebKit version included in AIR, the AIR update mechanism can prompt the user to install the new version of your application. For more information, see “Updating AIR applications” on page 376.

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

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

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

HTML content in AIR does not display SWF or PDF content if alpha, scaling, or transparency settings are applied. For more information, see Considerations when loading SWF or PDF content in an HTML page and “Window transparency” on page 136.

Overview of the HTML environment

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

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

![Diagram of JavaScript environment and AIR run-time environment]

The JavaScript environment has its own Document and Window objects. JavaScript code can interact with the AIR run-time environment through the runtime, nativeWindow, and htmlLoader properties. ActionScript code can interact with the JavaScript environment through the window property of an HTMLLoader object, which is a reference to the JavaScript Window object. In addition, both ActionScript and JavaScript objects can listen for events dispatched by both AIR and JavaScript objects.

The runtime property provides access to AIR API classes, allowing you to create new AIR objects as well as access class (also called static) members. To access an AIR API, you add the name of the class, with package, to the runtime property. For example, to create a File object, you would use the statement:

```javascript
var file = new window.runtime.filesystem.File();
```

Note: The AIR SDK provides a JavaScript file, AIRAliases.js, that defines more convenient aliases for the most commonly used AIR classes. When you import this file, you can use the shorter form air.Class instead of window.runtime.package.Class. For example, you could create the File object with `new air.File()`.

The NativeWindow object provides properties for controlling the desktop window. From within an HTML page, you can access the containing NativeWindow object with the window.nativeWindow property.
The HTMLLoader object provides properties, methods, and events for controlling how content is loaded and rendered. From within an HTML page, you can access the parent HTMLLoader object with the `window.htmlLoader` property.

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

**About security**

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

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

Another way to allow application and non-application content to cross-script each other, and the only way to give non-application content access to AIR APIs, is to create a sandbox bridge. A parent-to-child bridge allows content in a child frame, iframe, or window to access designated methods and properties defined in the application sandbox. Conversely, a child-to-parent bridge allows application content to access designated methods and properties defined in the sandbox of the child. Sandbox bridges are established by setting the `parentSandboxBridge` and `childSandboxBridge` properties of the window object. For more information, see “HTML security” on page 106 and “HTML frame and iframe elements” on page 77.

**About plug-ins and embedded objects**

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

No other WebKit plug-ins are supported in AIR.

**More Help topics**  
“HTML security” on page 106  
“HTML Sandboxes” on page 70  
“HTML frame and iframe elements” on page 77  
“JavaScript Window object” on page 75  
“The XMLHttpRequest object” on page 71  
“Adding PDF content” on page 293
**AIR and WebKit extensions**

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

Only the AIR additions and the most noteworthy WebKit extensions are covered here; for additional documentation on non-standard HTML, CSS, and JavaScript, see [www.webkit.org](http://www.webkit.org) and [developer.apple.com](http://developer.apple.com). For standards information, see the [W3C website](http://www.w3.org). Mozilla also provides a valuable general reference on HTML, CSS, and DOM topics (of course, the WebKit and Mozilla engines are not identical).

**Note:** AIR does not support the following standard and extended WebKit features: the JavaScript Window object `print()` method; plug-ins, except Acrobat or Adobe Reader 8.1+; Scalable Vector Graphics (SVG), the CSS `opacity` property.

**JavaScript in AIR**

AIR makes several changes to the typical behavior of common JavaScript objects. Many of these changes are made to make it easier to write secure applications in AIR. At the same time, these differences in behavior mean that some common JavaScript coding patterns, and existing web applications using those patterns, might not always execute as expected in AIR. For information on correcting these types of issues, see “Avoiding security-related JavaScript errors” on page 53.

**HTML Sandboxes**

AIR places content into isolated sandboxes according to the origin of the content. The sandbox rules are consistent with the same-origin policy implemented by most web browsers, as well as the rules for sandboxes implemented by the Adobe Flash Player. In addition, AIR provides a new application sandbox type to contain and protect application content. See “Sandboxes” on page 104 for more information on the types of sandboxes you may encounter when developing AIR applications.

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

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

To allow an application page to interact freely with content in a remote sandbox, the page can be mapped to the same domain as the remote content. For example, if you write an application that displays map data from an Internet service, the page of your application that loads and displays content from the service could be mapped to the service domain. The attributes for mapping pages into a remote sandbox and domain are new attributes added to the frame and iframe HTML elements.
To allow content in a non-application sandbox to safely use AIR features, you can set up a parent sandbox bridge. To allow application content to safely call methods and access properties of content in other sandboxes, you can set up a child sandbox bridge. Safety here means that remote content cannot accidentally get references to objects, properties, or methods that are not explicitly exposed. Only simple data types, functions, and anonymous objects can be passed across the bridge. However, you must still avoid explicitly exposing potentially dangerous functions. If, for example, you exposed an interface that allowed remote content to read and write files anywhere on a user’s system, then you might be giving remote content the means to do considerable harm to your users.

**JavaScript eval() function**

Use of the `eval()` function is restricted within the application sandbox once a page has finished loading. Some uses are permitted so that JSON-formatted data can be safely parsed, but any evaluation that results in executable statements results in an error. “Code restrictions for content in different sandboxes” on page 108 describes the allowed uses of the `eval()` function.

**Function constructors**

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

**Loading external scripts**

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

**The XMLHttpRequest object**

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

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

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

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

By default, AIR blocks cross-domain XMLHttpRequests in non-application sandboxes. A parent window in the application sandbox can choose to allow cross-domain requests in a child frame containing content in a non-application sandbox by setting `allowCrossDomainXHR`, an attribute added by AIR, to `true` in the containing frame or `iframe` element:
<iframe id="mashup"
    src="http://www.example.com/map.html"
    allowCrossDomainXHR="true"
</iframe>

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

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

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

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

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

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

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

For more information on sandbox mapping see “HTML frame and iframe elements” on page 77 and “HTML security” on page 106.

**The Canvas object**

The Canvas object defines an API for drawing geometric shapes such as lines, arcs, ellipses, and polygons. To use the canvas API, you first add a canvas element to the document and then draw into it using the JavaScript Canvas API. In most other respects, the Canvas object behaves like an image.

The following example draws a triangle using a Canvas object:
<html>
<body>
<canvas id="triangleCanvas" style="width:40px; height:40px;">"</canvas>
<script>
var canvas = document.getElementById("triangleCanvas");
var context = canvas.getContext("2d");
context.lineWidth = 3;
context.strokeStyle = ";
context.beginPath();
context.moveTo(5,5);
context.lineTo(35,5);
context.lineTo(20,35);
context.lineTo(5,5);
context.lineTo(6,5);
context.stroke();
</script>
</body>
</html>

For more documentation on the Canvas API, see the [Safari JavaScript Reference](https://developer.apple.com) from Apple. Note that the WebKit project recently began changing the Canvas API to standardize on the [HTML 5 Working Draft](https://www.w3.org) proposed by the Web Hypertext Application Technology Working Group (WHATWG) and W3C. As a result, some of the documentation in the Safari JavaScript Reference may be inconsistent with the version of the canvas present in AIR.

**Cookies**

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

**The Clipboard object**

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

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

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

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

The valid data MIME types are:
**About the HTML environment**

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

For more information on using the clipboard, see “Copy and paste” on page 222 and Using the Pasteboard from JavaScript (Apple Developer Center).

### Drag and Drop

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

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

The `dataTransfer` object has the following important members:

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

For more information on adding support for drag-and-drop to an AIR application see “Drag and drop” on page 213 and Using the Drag-and-Drop from JavaScript (Apple Developer Center).
innerHTML and outerHTML properties
AIR places security restrictions on the use of the `innerHTML` and `outerHTML` properties for content running in the application sandbox. Before the page load event, as well as during the execution of any load event handlers, use of the `innerHTML` and `outerHTML` properties is unrestricted. However, once the page has loaded, you can only use `innerHTML` or `outerHTML` properties to add static content to the document. Any statement in the string assigned to `innerHTML` or `outerHTML` that evaluates to executable code is ignored. For example, if you include an event callback attribute in an element definition, the event listener is not added. Likewise, embedded `<script>` tags are not evaluated. For more information, see the “HTML security” on page 106.

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

Document.designMode property
Set the `document.designMode` property to a value of `on` to make all elements in the document editable. Built-in editor support includes text editing, copy, paste, and drag-and-drop. Setting `designMode` to `on` is equivalent to setting the `contentEditable` property of the body element to `true`. You can use the `contentEditable` property on most HTML elements to define which sections of a document are editable. See “HTML contentEditable attribute” on page 80 for additional information.

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

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

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

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

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

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

```javascript
var preferencesFile = new window.runtime.flash.filesystem.File();
```
The `AIRAliases.js` file, provided in the AIR SDK, contains alias definitions that allow you to shorten such references. For example, when `AIRAliases.js` is imported into a page, a File object can be created with the following statement:

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

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

See “Using the AIRAliases.js file” on page 58.

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

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

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

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

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

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

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

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

**Window.parentSandboxBridge and Window.childSandboxBridge properties** The `parentSandboxBridge` and `childSandboxBridge` properties allow you to define an interface between a parent and a child frame. For more information, see “Cross-scripting content in different security sandboxes” on page 63.

**Window.setTimeout() and Window.setInterval() functions** AIR places security restrictions on use of the `setTimeout()` and `setInterval()` functions within the application sandbox. You cannot define the code to be executed as a string when calling `setTimeout()` or `setInterval()`. You must use a function reference. For more information, see “setTimeout() and setInterval()” on page 56.

**Window.open() function** When called by code running in a non-application sandbox, the `open()` method only opens a window when called as a result of user interaction (such as a mouse click or keypress). In addition, the window title is prefixed with the application title (to prevent windows opened by remote content from impersonating windows opened by the application). For more information, see the “Restrictions on calling the JavaScript window.open() method” on page 111.

**air.NativeApplication object**

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

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

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

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

The NativeApplication object can only be accessed from within the application sandbox. Interacting with the operating system “Working with runtime and operating system information” on page 337 describes the NativeApplication object in detail.

**The JavaScript URL scheme**

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

**Extensions to HTML**

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

“HTML frame and iframe elements” on page 77

“HTML Canvas element” on page 79

“HTML element event handlers” on page 79

**HTML frame and iframe elements**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**Place child.html in a remote sandbox, using the dominitialize event to establish a sandbox bridge:**
About the HTML environment

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

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

For more information, see “Cross-scripting content in different security sandboxes” on page 63 and “HTML security” on page 106.

HTML Canvas element

Defines a drawing area for use with the WebKit Canvas API. Graphics commands cannot be specified in the tag itself. To draw into the canvas, call the canvas drawing methods through JavaScript.

```html
<canvas id="drawingAtrium" style="width:300px; height:300px;"></canvas>
```

For more information, see “The Canvas object” on page 72.

HTML element event handlers

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

<table>
<thead>
<tr>
<th>Callback attribute name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oncontextmenu</td>
<td>Called when a context menu is invoked, such as through a right-click or command-click on selected text.</td>
</tr>
<tr>
<td>oncopy</td>
<td>Called when a selection in an element is copied.</td>
</tr>
<tr>
<td>oncut</td>
<td>Called when a selection in an element is cut.</td>
</tr>
<tr>
<td>ondominitialize</td>
<td>Called when the DOM of a document loaded in a frame or iframe is created, but before any DOM elements are created or scripts parsed.</td>
</tr>
<tr>
<td>ondrag</td>
<td>Called when an element is dragged.</td>
</tr>
</tbody>
</table>
About the HTML environment

**HTML contentEditable attribute**

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

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

Note: If you set the `document.designMode` property to `on`, then all elements in the document are editable, regardless of the setting of `contentEditable` for an individual element. However, setting `designMode` to `off`, does not disable editing of elements for which `contentEditable` is `true`. See “Document.designMode property” on page 75 for additional information.

**Extensions to CSS**

WebKit supports several extended CSS properties. The following table lists the extended properties for which support is established. Additional non-standard properties are available in WebKit, but are not fully supported in AIR, either because they are still under development in WebKit, or because they are experimental features that may be removed in the future.

<table>
<thead>
<tr>
<th>Callback attribute name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ondragend</td>
<td>Called when a drag is released.</td>
</tr>
<tr>
<td>ondragenter</td>
<td>Called when a drag gesture enters the bounds of an element.</td>
</tr>
<tr>
<td>ondragleave</td>
<td>Called when a drag gesture leaves the bounds of an element.</td>
</tr>
<tr>
<td>ondragover</td>
<td>Called continuously while a drag gesture is within the bounds of an element.</td>
</tr>
<tr>
<td>ondragstart</td>
<td>Called when a drag gesture begins.</td>
</tr>
<tr>
<td>ondrop</td>
<td>Called when a drag gesture is released while over an element.</td>
</tr>
<tr>
<td>onerror</td>
<td>Called when an error occurs while loading an element.</td>
</tr>
<tr>
<td>oninput</td>
<td>Called when text is entered into a form element.</td>
</tr>
<tr>
<td>onpaste</td>
<td>Called when an item is pasted into an element.</td>
</tr>
<tr>
<td>onscroll</td>
<td>Called when the content of a scrollable element is scrolled.</td>
</tr>
<tr>
<td>onsearch</td>
<td>Called when an element is copied (?) Apple docs correct?)</td>
</tr>
<tr>
<td>onselectstart</td>
<td>Called when a selection begins.</td>
</tr>
</tbody>
</table>
### CSS property name

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-webkit-border-horizontal-spacing</td>
<td>Non-negative unit of length</td>
<td>Specifies the horizontal component of the border spacing.</td>
</tr>
<tr>
<td>-webkit-border-vertical-spacing</td>
<td>Non-negative unit of length</td>
<td>Specifies the vertical component of the border spacing.</td>
</tr>
<tr>
<td>-webkit-line-break</td>
<td>after-white-space, normal</td>
<td>Specifies the line break rule to use for Chinese, Japanese, and Korean (CJK) text.</td>
</tr>
<tr>
<td>-webkit-margin-bottom-collapse</td>
<td>collapse, discard, separate</td>
<td>Defines how the bottom margin of a table cell collapses.</td>
</tr>
<tr>
<td>-webkit-margin-collapse</td>
<td>collapse, discard, separate</td>
<td>Defines how the top and bottom margins of a table cell collapses.</td>
</tr>
<tr>
<td>-webkit-margin-start</td>
<td>Any unit of length</td>
<td>The width of the starting margin.</td>
</tr>
<tr>
<td>-webkit-margin-top-collapse</td>
<td>collapse, discard, separate</td>
<td>Defines how the top margin of a table cell collapses.</td>
</tr>
<tr>
<td>-webkit-nbsp-mode</td>
<td>normal, space</td>
<td>Defines the behavior of non-breaking spaces within the enclosed content.</td>
</tr>
<tr>
<td>-webkit-padding-start</td>
<td>Any unit of length</td>
<td>Specifies the width of the starting padding.</td>
</tr>
<tr>
<td>-webkit-rtl-ordering</td>
<td>logical, visual</td>
<td>Overrides the default handling of mixed left-to-right and right-to-left text.</td>
</tr>
<tr>
<td>-webkit-text-fill-color</td>
<td>Any named color or numeric color value</td>
<td>Specifies the text fill color.</td>
</tr>
<tr>
<td>-webkit-text-security</td>
<td>circle, disc, none, square</td>
<td>Specifies the replacement shape to use in a password input field.</td>
</tr>
<tr>
<td>-webkit-user-drag</td>
<td>• auto — Default behavior</td>
<td>Overrides the automatic drag behavior.</td>
</tr>
<tr>
<td></td>
<td>• element — The entire element is dragged</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• none — The element cannot be dragged</td>
<td></td>
</tr>
<tr>
<td>-webkit-user-modify</td>
<td>read-only, read-write, read-write-plaintext-only</td>
<td>Specifies whether the content of an element can be edited.</td>
</tr>
<tr>
<td>-webkit-user-select</td>
<td>• auto — Default behavior</td>
<td>Specifies whether a user can select the content of an element.</td>
</tr>
<tr>
<td></td>
<td>• none — The element cannot be selected</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• text — Only text in the element can be selected</td>
<td></td>
</tr>
</tbody>
</table>

For more information, see the Apple Safari CSS Reference (http://developer.apple.com/documentation/AppleApplications/Reference/SafariCSSRef/).
Chapter 12: Handling HTML-related events

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

HTMLLoader events

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

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>htmlDOMInitialize</td>
<td>Dispatched when the HTML document is created, but before any scripts are parsed or DOM nodes are added to the page.</td>
</tr>
<tr>
<td>complete</td>
<td>Dispatched when the HTML DOM has been created in response to a load operation, immediately after the onload event in the HTML page.</td>
</tr>
<tr>
<td>htmlBoundsChanged</td>
<td>Dispatched when one or both of the contentWidth and contentHeight properties have changed.</td>
</tr>
<tr>
<td>locationChange</td>
<td>Dispatched when the location property of the HTMLLoader has changed.</td>
</tr>
<tr>
<td>scroll</td>
<td>Dispatched anytime the HTML engine changes the scroll position. Scroll events can be because of navigation to anchor links (# links) in the page or because of calls to the window.scrollTo() method. Entering text in a text input or text area can also cause a scroll event.</td>
</tr>
<tr>
<td>uncaughtScriptException</td>
<td>Dispatched when a JavaScript exception occurs in the HTMLLoader and the exception is not caught in JavaScript code.</td>
</tr>
</tbody>
</table>

You can also register an ActionScript function for a JavaScript event (such as onClick). For details, see Handling DOM events with ActionScript.

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

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

- Defining an on event handler within an HTML element opening tag, as in:
  ```html
div id="myDiv" onclick="myHandler()"
  ```

- Callback function properties, such as:
  ```javascript
document.getElementById("myDiv").onclick
  ```

- Event listeners that you register using the addEventListener() method, as in:
  ```javascript
document.getElementById("myDiv").addEventListener("click", clickHandler)
  ```
However, since runtime objects do not appear in the DOM, you can only add event listeners by calling the addEventListener() method of an AIR object.

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

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

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

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

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

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

**Default behaviors**

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

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

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

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

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

The event flow

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

Adobe AIR event objects

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

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

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

Understanding Event class properties

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

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

Event object types

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

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

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

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

```javascript
myFileStream.addEventListener(Event.COMPLETE, htmlRenderHandler);
```
myFileStream.addEventListener("complete", htmlRenderHandler);

**Default behavior information**

Your code can check whether the default behavior for any given event object can be prevented by accessing the `cancelable` property. The `cancelable` property holds a Boolean value that indicates whether a default behavior can be prevented. You can prevent, or cancel, the default behavior associated with a small number of events using the `preventDefault()` method. For more information, see “Canceling default event behavior” on page 85.

**Understanding Event class methods**

There are three categories of Event class methods:

- Utility methods, which can create copies of an event object or convert it to a string.
- Event flow methods, which remove event objects from the event flow (primarily of use when using ActionScript 3.0 in SWF content for the runtime—see “The event flow” on page 84).
- Default behavior methods, which prevent default behavior or check whether it has been prevented.

**Event class utility methods**

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

**Canceling default event behavior**

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

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

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

**Subclasses of the Event class**

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

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

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

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

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

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

Creating an event handler function

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

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

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

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

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

```
addEventListner(Event.MOVE, moveHandler);
```

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

Removing event listeners

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

Removing event listeners in HTML pages that navigate

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

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

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

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

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

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

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

```
var html = runtime.flash.html.HTMLLoader.createRootWindow(true);
html.addEventListener('complete', htmlCompleteListener);
function htmlCompleteListener()
{
  html.removeEventListener(complete, arguments.callee)
  // handler code..
}
html.load(new runtime.flash.net.URLRequest("second.html"));
```
Removing unneeded event listeners also allows the system garbage collector to reclaim any memory associated with those listeners.

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

**Error events without listeners**
Exceptions, rather than events, are the primary mechanism for error handling in the runtime classes. However, exception handling does not work for asynchronous operations such as loading files. If an error occurs during an asynchronous operation, the runtime dispatches an error event object. If you do not create a listener for the error event, the AIR Debug Launcher presents a dialog box with information about the error.

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

An error event does not cause an application to stop executing. It manifests only as a dialog box on the AIR Debug Launcher. It does not manifest at all in the installed AIR application running in the runtime.
Chapter 13: Scripting the HTML Container

The HTMLLoader class serves as the container for HTML content in Adobe® AIR®. The class provides many properties and methods for controlling the behavior and appearance of the object on the ActionScript® 3.0 display list. In addition, the class defines properties and methods for such tasks as loading and interacting with HTML content and managing history.

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

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

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

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

Display properties of HTMLLoader objects

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

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

Basic display properties

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

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

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

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

Transparency of HTMLLoader content

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

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

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

Scaling HTMLLoader content

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

Considerations when loading SWF or PDF content in an HTML page

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

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

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

Note: The runtime cannot display SWF or PDF content in transparent windows.

For more information on loading these types of media in an HTMLLoader, see “Embedding SWF content in HTML” on page 59 and “Adding PDF content” on page 293.
Advanced display properties

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

The advanced display properties include:

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

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

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

**Note:** Display object classes, such as Sprite and BlurFilter, are not commonly used in HTML-based applications. They are not listed in the Adobe AIR Language Reference for HTML Developers (http://www.adobe.com/go/learn_air_html_jslr) nor aliased in the AIRAliases.js file. For documentation about these classes, you can refer to the Flex 3 Language Reference.

Accessing the HTML history list

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

<table>
<thead>
<tr>
<th>Class member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>historyLength</td>
<td>The overall length of the history list, including back and forward entries.</td>
</tr>
<tr>
<td>historyPosition</td>
<td>The current position in the history list. History items before this position represent “back” navigation, and items after this position represent “forward” navigation.</td>
</tr>
<tr>
<td>getHistoryAt()</td>
<td>Returns the URLRequest object corresponding to the history entry at the specified position in the history list.</td>
</tr>
</tbody>
</table>
Items in the history list are stored as objects of type HistoryListItem. The HistoryListItem class has the following properties:

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

### Setting the user agent used when loading HTML content

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

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

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

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

### Setting the character encoding to use for HTML content

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

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

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

```javascript
window.htmlLoader.textEncodingOverride = "ISO-8859-1";
```
Specify the character encoding for the HTMLLoader content to use when an HTML page does not specify a setting with the `textEncodingFallback` property of the HTMLLoader object:

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

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

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

### Defining browser-like user interfaces for HTML content

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

**Important:** You can only create a custom implementation of the HTMLHost class using ActionScript. You can import and use a compiled ActionScript (SWF) file containing a custom implementation in an HTML page. See “Using ActionScript libraries within an HTML page” on page 60 for more information about importing ActionScript libraries into HTML.

### About extending the HTMLHost class

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

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

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

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

When you create an HTML window in AIR with the HTMLLoader class `createRootWindow()` method, an HTMLHost instance supporting the default behaviors is assigned automatically. You can change the host object behavior by assigning a different HTMLHost implementation to the `htmlHost` property of the HTMLLoader, or you can assign `null` to disable the features entirely.
Note: AIR assigns a default HTMLHost object to the initial window created for an HTML-based AIR application and any windows created by the default implementation of the JavaScript window.open() method.

Example: Extending the HTMLHost class

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

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

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

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

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

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

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

            htmlControl.htmlHost = new HTMLHostImplementation();

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

            return htmlControl;
        }
    }
}
```
4 Compile the class into a SWF file using the acompc component compiler.

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

**Note:** The acompc compiler is included with the Flex 3 SDK (but not the AIR SDK, which is targeted for HTML developers who do not generally need to compile SWF files.) Instructions for using acompc are provided in the Developing Adobe AIR Applications with Adobe Flex 3.

5 Open the Host.zip file and extract the Library.swf file inside.

6 Rename Library.swf to HTMLHostLibrary.swf. This SWF file is the library to import into the HTML page.

7 Import the library into the HTML page using a `<script>` tag:

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

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

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

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

```html
<html>
<head>
    <title>HTMLHost Example</title>
    <script src="HTMLHostLibrary.swf" type="application/x-shockwave-flash"></script>
    <script language="javascript">
    window.htmlLoader.htmlHost = new window.runtime.HTMLHostImplementation();
    function test(){
        window.open('child.html', 'Child', 'fullscreen');
    }
    </script>
</head>
<body>
    <button onClick="test()">Create Window</button>
</body>
</html>
```

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

**Handling changes to the window.location property**

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

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

**Note:** You can use the `htmlLoader` property of the HTMLHost object to reference the current HTMLLoader object.
Handling JavaScript calls to `window.moveBy()`, `window.moveTo()`, `window.resizeTo()`, `window.resizeBy()`

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

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

Handling JavaScript calls to `window.open()`

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

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

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

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

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

<table>
<thead>
<tr>
<th><code>HTMLWindowCreateOptions</code> property</th>
<th>Corresponding setting in the <code>features</code> string in the JavaScript call to <code>window.open()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fullscreen</code></td>
<td><code>fullscreen</code></td>
</tr>
<tr>
<td><code>height</code></td>
<td><code>height</code></td>
</tr>
<tr>
<td><code>locationBarVisible</code></td>
<td><code>location</code></td>
</tr>
<tr>
<td><code>menuBarVisible</code></td>
<td><code>menubar</code></td>
</tr>
<tr>
<td><code>resizable</code></td>
<td><code>resizable</code></td>
</tr>
<tr>
<td><code>scrollBarsVisible</code></td>
<td><code>scrollbars</code></td>
</tr>
</tbody>
</table>
The HTMLLoader class does not implement all the features that can be specified in the feature string. Your application must provide scroll bars, location bars, menu bars, status bars, and toolbars when appropriate.

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

### Handling JavaScript calls to `window.close()`

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

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

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

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

### Handling changes of the `window.status` property

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

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

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

The HTMLLoader object does not provide a status bar.

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

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

```actionscript
override public function updateTitle():void {
    status bar visible
    toolbar visible
    width
    x (left or screenX)
    y (top of screenY)
    HTMLWindowCreateOptions property
    Corresponding setting in the features string in the JavaScript call to `window.open()`
    statusBarVisible    status
    toolBarVisible      toolbar
    width                width
    x (left or screenX) left
    y (top of screenY)  top
```
override public function updateTitle(title:String):void
{
    htmlLoader.stage.nativeWindow.title = title + " - Sample";
}

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

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

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

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

```java
override public function windowBlur():void
{
    htmlLoader.alpha = 0.5;
}

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

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

**Creating windows with scrolling HTML content**

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

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

For example, the following code uses the `HTMLLoader.createRootWindow()` method to create a window with HTMLLoader content that uses scrollbars:
var initOptions = new air.NativeWindowInitOptions();
var bounds = new air.Rectangle(10, 10, 600, 400);
var html2 = air.HTMLLoader.createRootWindow(true, initOptions, true, bounds);
var urlReq2 = new air.URLRequest("http://www.example.com");
html2.load(urlReq2);
html2.stage.nativeWindow.activate();

Note: Windows created by calling createRootWindow() directly in JavaScript remain independent from the opening HTML window. The JavaScript Window opener and parent properties, for example, are null. However, if you call createRootWindow() indirectly by overriding the HTMLHost createWindow() method to call createRootWindow(), then opener and parent do reference the opening HTML window.
Chapter 14: AIR security

This topic discusses security issues you should consider when developing an AIR application.

AIR security basics

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

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

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

Installation and updates

AIR applications are distributed via AIR installer files which use the air extension. When Adobe AIR is installed and an AIR installer file is opened, the runtime administers the installation process.

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

Runtime installation location

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

The runtime is installed to the following location on a user's computer:

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

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

The runtime can be installed in two ways: using the seamless install feature (installing directly from a web browser) or via a manual install. For more information, see “Distributing, Installing, and Running AIR applications” on page 360.
Seamless install (runtime and application)

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

Manual install

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

For more information on this process, see “Distributing, Installing, and Running AIR applications” on page 360

Application installation flow

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

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

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

Application destination

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

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

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

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

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

An AIR application is run by:

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

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

AIR application storage

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

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

You can access the application storage directory via the air.File.applicationStorageDirectory property. You can access its contents using the resolvePath() method of the File class. For details, see “Working with the file system” on page 186.

Updating Adobe AIR

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

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

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

Note: Developers can specify the version of an application by setting the version property of the application descriptor file. 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.” It is up to the developer to maintain meaningful versioning. For details, see “Defining properties in the application descriptor file” on page 117.

Uninstalling an AIR application

A user can uninstall an AIR application:

- On Windows: Using the Add/Remove Programs panel to remove the application.
- On Mac OS: Deleting the app file from the install location.

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

Uninstalling Adobe AIR

AIR can be uninstalled:

- On Windows: by running Add/Remove Programs from the Control Panel, selecting Adobe AIR and selecting “Remove”.
- On Mac OS: by running the Adobe AIR Uninstaller application in the Applications directory.

Windows registry settings for administrators

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

<table>
<thead>
<tr>
<th>Registry setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppInstallDisabled</td>
<td>Specifies that AIR application installation and uninstallation are allowed. Set to 0 for “allowed,” set to 1 for “disallowed.”</td>
</tr>
<tr>
<td>UntrustedAppInstallDisabled</td>
<td>Specifies that installation of untrusted AIR applications (applications that do not includes a trusted certificate) is allowed (see “Digitally signing an AIR file” on page 369). Set to 0 for “allowed,” set to 1 for “disallowed.”</td>
</tr>
<tr>
<td>UpdateDisabled</td>
<td>Specifies that updating the runtime is allowed, either as a background task or as part of an explicit installation. Set to 0 for “allowed,” set to 1 for “disallowed.”</td>
</tr>
</tbody>
</table>
Sandboxes

AIR provides a comprehensive security architecture that defines permissions accordingly to each file in an AIR application, both internal and external. Permissions are granted to files according to their origin, and are assigned into logical security groupings called sandboxes.

The AIR security model is based on the Flash Player security model. This security model categorizes each item of loaded content into a security sandbox based on the content’s origin. There are sandboxes for content loaded from the local file system and those for content loaded from a network domain. For details, see the Flash Player 9 Security white paper (http://www.adobe.com/go/fp9_0_security) or the Flash Player 10 Security white paper (http://www.adobe.com/go/fp10_0_security_en).

About the AIR application sandboxes

The runtime security model of sandboxes is composed of the Flash Player security model with the addition of the application sandbox. Files that are not in the application sandbox have security restrictions similar to those specified by the Flash Player security model.

The runtime uses these security sandboxes to define the range of data that code may access and the operations it may execute. To maintain local security, the files in each sandbox are isolated from the files of other sandboxes. For example, a SWF file loaded into an AIR application from an external Internet URL is placed into a remote sandbox, and does not by default have permission to script into files that reside in the application directory, which are assigned to the application sandbox.

The following table describes each type of sandbox:

<table>
<thead>
<tr>
<th>Sandbox</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>application</td>
<td>The file resides in the application directory and operates with the full set of AIR privileges.</td>
</tr>
<tr>
<td>remote</td>
<td>The file is from an Internet URL, and operates under domain-based sandbox rules analogous to the rules that apply to remote files in Flash Player. (There are separate remote sandboxes for each network domain, such as <a href="http://www.example.com">http://www.example.com</a> and <a href="https://foo.example.org">https://foo.example.org</a>.)</td>
</tr>
<tr>
<td>local-trusted</td>
<td>The file is a local file and the user has designated it as trusted, using either the Settings Manager or a Flash Player trust configuration file. The file can both read from local data sources and communicate with the Internet, but does not have the full set of AIR privileges.</td>
</tr>
<tr>
<td>local-with-networking</td>
<td>The file is a local SWF file published with a networking designation, but has not been explicitly trusted by the user. The file can communicate with the Internet but cannot read from local data sources. This sandbox is only available to SWF content.</td>
</tr>
<tr>
<td>local-with-filesystem</td>
<td>The file is a local scripting file that was not published with a networking designation and has not been explicitly trusted by the user. This includes JavaScript files that have not been trusted. The file can read from local data sources but cannot communicate with the Internet.</td>
</tr>
</tbody>
</table>

This topic focuses primarily on the application sandbox and its relationship to other sandboxes in the AIR application. Developers that use content assigned to other sandboxes should read further documentation on the Flash Player security model. See the “Flash Player Security” chapter in the Programming ActionScript 3.0 (http://www.adobe.com/go/flashcs4_prog_as3_security_en) documentation and the Flash Player 9 Security white paper (http://www.adobe.com/go/fp9_0_security) or the Flash Player 10 Security white paper (http://www.adobe.com/go/fp10_0_security_en).
The application sandbox
When an application is installed, all files included within an AIR installer file are installed onto the user’s computer into an application directory. Developers can reference this directory in code through the app:/ URL scheme (see “Using AIR URL schemes in URLs” on page 344). All files within the application directory tree are assigned to the application sandbox when the application is run. Content in the application sandbox is blessed with the full privileges available to an AIR application, including interaction with the local file system.

Many AIR applications use only these locally installed files to run the application. However, AIR applications are not restricted to just the files within the application directory — they can load any type of file from any source. This includes files local to the user’s computer as well as files from available external sources, such as those on a local network or on the Internet. File type has no impact on security restrictions; loaded HTML files have the same security privileges as loaded SWF files from the same source.

Content in the application security sandbox has access to AIR APIs that content in other sandboxes are prevented from using. For example, the air.NativeApplication.nativeApplication.applicationDescriptor property, which returns the contents of the application descriptor file for the application, is restricted to content in the application security sandbox. Another example of a restricted API is the FileStream class, which contains methods for reading and writing to the local file system.

For HTML content (in an HTMLLoader object), all AIR JavaScript APIs (those that are available via the window.runtime property, or via the air object when using the AIRAliases.js file) are available to content in the application security sandbox. HTML content in another sandbox does not have access to the window.runtime property, so this content cannot access the AIR APIs.

JavaScript and HTML restrictions
For HTML content in the application security sandbox, there are limitations on using APIs that can dynamically transform strings into executable code after the code is loaded. This is to prevent the application from inadvertently injecting (and executing) code from non-application sources (such as potentially insecure network domains). An example is the use of the eval() function. For details, see “Code restrictions for content in different sandboxes” on page 108.

Restrictions on img tags in ActionScript text field content
To prevent possible phishing attacks, img tags in HTML content in ActionScript TextField objects are ignored in SWF content in the application security sandbox.

Restrictions on asfunction
Content in the application sandbox cannot use the asfunction protocol in HTML content in ActionScript 2.0 text fields.

No access to the cross-domain persistent cache
SWF content in the application sandbox cannot use the cross-domain cache, a feature that was added to Flash Player 9 Update 3. This feature lets Flash Player persistently cache Adobe platform component content and reuse it in loaded SWF content on demand (eliminating the need to reload the content multiple times).
Privileges of content in non-application sandboxes

Files loaded from a network or Internet location are assigned to the remote sandbox. Files loaded from outside the application directory are assigned to either the local-with-filesystem, local-with-networking, or the local-trusted sandbox; this depends on how the file was created and if the user has explicitly trusted the file through the Flash Player Global Settings Manager. For details, see http://www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager.html.

JavaScript and HTML restrictions

Unlike content in the application security sandbox, JavaScript content in a non-application security sandbox can call the eval() function to execute dynamically generated code at any time. However, there are restrictions to JavaScript in a non-application security sandbox. These include:

- JavaScript code in a non-application sandbox does not have access to the window.runtime object, and as such this code cannot execute AIR APIs.
- By default, content in a non-application security sandbox cannot use XMLHttpRequest calls to load data from other domains other than the domain calling the request. However, application code can grant non-application content permission to do so by setting an allowCrossdomainXHR attribute in the containing frame or iframe. For more information, see “Scripting between content in different domains” on page 111.
- There are restrictions on calling the JavaScript window.open() method. For details, see "Restrictions on calling the JavaScript window.open() method" on page 111.

For details, see “Code restrictions for content in different sandboxes” on page 108.

Restrictions on loading CSS, frame, iframe, and img elements

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

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

HTML security

The runtime enforces rules and provides mechanisms for overcoming possible security vulnerabilities in HTML and JavaScript. The same rules are enforced whether your application is primarily written in JavaScript or whether you load the HTML and JavaScript content into a SWF-based application. Content in the application sandbox and the non-application security sandbox (see “Sandboxes” on page 104) have different privileges. When loading content into an iframe or frame, the runtime provides a secure sandbox bridge mechanism that allows content in the frame or iframe to communicate securely with content in the application security sandbox.

This topic describes the AIR HTML security architecture and how to use iframes, frames, and the sandbox bridge to set up your application.

For more information, see “Avoiding security-related JavaScript errors” on page 53.

Overview on configuring your HTML-based application

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

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

For a complete list of the restrictions on JavaScript in the application sandbox, see “Code restrictions for content in
different sandboxes” on page 108.

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

**Setting up a parent-child sandbox relationship**

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

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

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

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

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

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

- childSandboxBridge — The childSandboxBridge property allows the child frame to expose an interface to content
  in the parent frame. To expose an interface, you set the childSandbox property to a function or object in the child
  frame. You can then access the object or function from content in the parent frame. The following example shows how
  a script running in a child frame can expose an object containing a function and a property to its parent:
var interface = {}; 
interface.calculatePrice = function(){
    return .45 + 1.20;
}
interface.storeID = "abc"
window.childSandboxBridge = interface;

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

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

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

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

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

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

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

**Code restrictions for content in different sandboxes**

As discussed in the introduction to this topic, “HTML security” on page 106, the runtime enforces rules and provides mechanisms for overcoming possible security vulnerabilities in HTML and JavaScript. This topic lists those restrictions. If code attempts to call these restricted APIs, the runtime throws an error with the message “Adobe AIR runtime security violation for JavaScript code in the application security sandbox.”

For more information, see “Avoiding security-related JavaScript errors” on page 53.

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

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

For example, if your application uses string data from a remote source to write to the innerHTML property of a DOM element, the string could include executable (JavaScript) code that could perform insecure operations. However, while the content is loading, there is no risk of inserting remote strings into the DOM.
One restriction is in the use of the JavaScript `eval()` function. Once code in the application sandbox is loaded and after processing of the `onload` event handler, you can only use the `eval()` function in limited ways. The following rules apply to the use of the `eval()` function after code is loaded from the application security sandbox:

- Expressions involving literals are allowed. For example:
  ```javascript
  eval("null");
  eval("3 + .14");
  eval("'foo'");
  ```
- Object literals are allowed, as in the following:
  ```javascript
  { prop1: val1, prop2: val2 }
  ```
- Object literal setter/getters are prohibited, as in the following:
  ```javascript
  { get prop1() { ... }, set prop1(v) { ... } }
  ```
- Array literals are allowed, as in the following:
  ```javascript
  [ val1, val2, val3 ]
  ```
- Expressions involving property reads are prohibited, as in the following:
  ```javascript
  a.b.c
  ```
- Function invocation is prohibited.
- Function definitions are prohibited.
- Setting any property is prohibited.
- Function literals are prohibited.

However, while the code is loading, before the `onload` event, and during execution the `onload` event handler function, these restrictions do not apply to content in the application security sandbox. For example, after code is loaded, the following code results in the runtime throwing an exception:

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

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

Ways to generate dynamic code are the following:

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

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

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

For other Ajax frameworks and JavaScript code libraries, check to see if the code in the framework or library works within these restrictions on dynamically generated code. If they do not, include any content that uses the framework or library in a non-application security sandbox. For details, see “Privileges of content in non-application sandboxes” on page 106 and “Scripting between application and non-application content” on page 113. Adobe maintains a list of Ajax frameworks known to support the application security sandbox, at http://www.adobe.com/products/air/develop/ajax/features/.

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

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

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

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

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

You can expose runtime functionality to content in a non-application sandbox by using a script bridge. For details, see and “Scripting between application and non-application content” on page 113.

**Restrictions on using XMLHttpRequest calls**

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

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

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

For more information, see “Scripting between content in different domains” on page 111.
Restrictions on loading CSS, frame, iframe, and img elements (for content in non-application sandboxes)

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

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

Restrictions on calling the JavaScript window.open() method

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

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

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

Content in the local-with-filesystem, local-with-networking, or local-trusted sandboxes (see “Sandboxes” on page 104) can only use the window.open() method to open content in local sandboxes. It cannot use window.open() to open content from the application or remote sandboxes.

Errors when calling restricted code

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

For more information, see “Avoiding security-related JavaScript errors” on page 53.

Sandbox protection when loading HTML content from a string

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

Scripting between content in different domains

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

About the AIR sandbox bridge

Normally, content from other domains cannot call scripts in other domains.
There are still cases where the main AIR application requires content from a remote domain to have controlled access to scripts in the main AIR application, or vice versa. To accomplish this, the runtime provides a sandbox bridge mechanism, which serves as a gateway between the two sandboxes. A sandbox bridge can provide explicit interaction between remote and application security sandboxes.

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

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

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

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

**Sandbox bridge example (HTML)**

In HTML content, the `parentSandboxBridge` and `childSandboxBridge` properties are added to the JavaScript window object of a child document. For an example of how to set up bridge functions in HTML content, see “Setting up a sandbox bridge interface” on page 65.

**Limiting API exposure**

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

**More Help topics**

“Cross-scripting content in different security sandboxes” on page 63

“The application sandbox” on page 105

“Privileges of content in non-application sandboxes” on page 106

**Writing to disk**

Applications running in a web browser have only limited interaction with the user’s local file system. Web browsers implement security policies that ensure that a user’s computer cannot be compromised as a result of loading web content. For example, SWF files running through Flash Player in a browser cannot directly interact with files already on a user’s computer. Shared objects and cookies can be written to a user’s computer for the purpose of maintaining user preferences and other data, but this is the limit of file system interaction. Because AIR applications are natively installed, they have a different security contract, one which includes the capability to read and write across the local file system.
This freedom comes with high responsibility for developers. Accidental application insecurities jeopardize not only the functionality of the application, but also the integrity of the user's computer. For this reason, developers should read “Best security practices for developers” on page 114.

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

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

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

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

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

If an application is designed to interact with existing files in the user's file system, be sure to read “Best security practices for developers” on page 114.

### Working securely with untrusted content

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

### Scripting between application and non-application content

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

The benefit of sandbox bridges is best illustrated through example. Suppose an AIR music store application wants to provide an API to advertisers who want to create their own SWF files, with which the store application can then communicate. The store wants to provide advertisers with methods to look up artists and CDs from the store, but also wants to isolate some methods and properties from the third-party SWF file for security reasons.
A sandbox bridge can provide this functionality. By default, content loaded externally into an AIR application at runtime does not have access to any methods or properties in the main application. With a custom sandbox bridge implementation, a developer can provide services to the remote content without exposing these methods or properties. Consider the sandbox bridge as a pathway between trusted and untrusted content, providing communication between loader and loadee content without exposing object references.

For more information on how to securely use sandbox bridges, see "Scripting between content in different domains" on page 111.

**Best security practices for developers**

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

**Risk from importing files into the application security sandbox**

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

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

**Risk from using an external source to determine paths**

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

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

**Risk from using, storing, or transmitting insecure credentials**

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

- If credentials must be stored locally, encrypt the credentials when writing to the local file system. The runtime provides an encrypted storage unique to each installed application, via the EncryptedLocalStore class. For details, see “Storing encrypted data” on page 290.
- Do not transmit unencrypted user credentials to a network source unless that source is trusted.
• Never specify a default password in credential creation — let users create their own. Users who leave the default unchanged expose their credentials to an attacker who already knows the default password.

Risk from a downgrade attack
During application install, the runtime checks to ensure that a version of the application is not currently installed. If an application is already installed, the runtime compares the version string against the version that is being installed. If this string is different, the user can choose to upgrade their installation. The runtime does not guarantee that the newly installed version is newer than the older version, only that it is different. An attacker can distribute an older version to the user to circumvent a security weakness. For this reason, the developer is advised to make version checks when the application is run. It is a good idea to have applications check the network for required updates. That way, even if an attacker gets the user to run an old version, that old version will recognize that it needs to be updated. Also, using a clear versioning scheme for your application makes it more difficult to trick users into installing a downgraded version. For details on providing application versions, see "Defining properties in the application descriptor file" on page 117.

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

For more information about digitally signing AIR applications, see "Digitally signing an AIR file" on page 369 and "Creating an AIR application using the command line tools" on page 23.
Chapter 15: Setting AIR application properties

Aside from all the files and other assets that make up an AIR application, each AIR application requires an application descriptor file. The application descriptor file is an XML file which defines the basic properties of the application.

When developing AIR applications, you must create an application descriptor file for each Adobe® AIR® project. A sample descriptor file, descriptor-sample.xml, can be found in the samples directory of your Adobe® AIR™ installation.

The application descriptor file structure

The application descriptor file contains properties that affect the entire application, such as its name, version, copyright, and so on. Any filename can be used for the application descriptor file. When you package the application with ADT, the application descriptor file is renamed application.xml and placed within a special directory inside the AIR package. When you create an AIR file using the default settings in Flash CS3 or Flash CS4, the application descriptor file is renamed to application.xml and placed inside a special directory in the AIR package.

Here's an example application descriptor file:

```xml
<?xml version="1.0" encoding="utf-8"?>
<application xmlns="http://ns.adobe.com/air/application/1.5">
  <id>HelloWorld</id>
  <version>2.0</version>
  <filename>Hello World</filename>
  <name>Example Co. AIR Hello World</name>
  <description>
    <text xml:lang="en">This is a example.</text>
    <text xml:lang="fr">C’est un exemple.</text>
    <text xml:lang="es">Esto es un ejemplo.</text>
  </description>
  <copyright>Copyright (c) 2006 Example Co.</copyright>
  <initialWindow>
    <title>Hello World</title>
    <content>
      HelloWorld-debug.html
    </content>
    <systemChrome>none</systemChrome>
    <transparent>true</transparent>
    <visible>true</visible>
    <minimizable>true</minimizable>
    <maximizable>false</maximizable>
    <resizable>false</resizable>
    <width>640</width>
    <height>480</height>
    <minSize>320 240</minSize>
    <maxSize>1280 960</maxSize>
  </initialWindow>
  <installFolder>Example Co/Hello World</installFolder>
  <programMenuFolder>Example Co</programMenuFolder>
</application>
```
Setting AIR application properties

Defining properties in the application descriptor file

Use the XML elements and attributes in the application descriptor to define the following types of properties for your AIR application:

- Required AIR runtime version
- Application identity
- Installation and program menu folders
- Initial content and window properties
- Application icon files
- Whether your application provides a custom update UI
- Whether your application can be invoked by SWF content running in the user’s browser
- File type associations

Specifying the required AIR version

The attributes of the root element of an application descriptor file, application, specifies the required AIR runtime version:

```xml
<application xmlns="http://ns.adobe.com/air/application/1.5.3"/>
```

**xmlns** The AIR namespace, which you must define as the default XML namespace. The namespace changes with each major release of AIR. The last segment of the namespace, such as “1.5.3” indicates the runtime version required by the application. Be sure to set the namespace to AIR 1.5.3 (“http://ns.adobe.com/air/application/1.5.3”) if your application uses any new AIR 1.5.3 features.
For SWF-based applications, the AIR runtime version specified in the application descriptor determines the maximum SWF version that can be loaded as the initial content of the application. Applications that specify AIR 1.0 or AIR 1.1 can only use SWF9 (Flash Player 9) files as initial content — even when run using the AIR 1.5 runtime. Applications that specify AIR 1.5, or higher, can use either SWF9 or SWF10 (Flash Player 10) files as initial content. The SWF version determines which version of the AIR and Flash Player APIs are available. If a SWF9 file is used as the initial content of an AIR 1.5 application, that application will only have access to the AIR 1.1 and Flash Player 9 APIs. Furthermore, behavior changes made to existing APIs in AIR 1.5 or Flash Player 10 will not be effective. (Important security-related changes to APIs are an exception to this principle and can be applied retroactively in present or future patches of the runtime.)

For HTML-based applications, the runtime version specified in the application descriptor alone determines which version of the AIR and Flash Player APIs are available to the application. The HTML, CSS, and JavaScript behaviors are always determined by the version of Webkit used in the installed AIR runtime, not by the application descriptor.

When an AIR application loads SWF content, the version of the AIR and Flash Player APIs available to that content depends on how the content is loaded. The following table shows how the API version is determined based on the loading method:

<table>
<thead>
<tr>
<th>How the content is loaded</th>
<th>How the API version is determined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial content, SWF-based application</td>
<td>SWF version of the loaded file</td>
</tr>
<tr>
<td>Initial content, HTML-based application</td>
<td>Application descriptor namespace</td>
</tr>
<tr>
<td>SWF loaded by SWF content</td>
<td>Version of the loading content</td>
</tr>
<tr>
<td>SWF library loaded by HTML content using &lt;script&gt; tag</td>
<td>Application descriptor namespace</td>
</tr>
<tr>
<td>SWF loaded by HTML content using AIR or Flash Player APIs (such as flash.display.Loader)</td>
<td>Application descriptor namespace</td>
</tr>
<tr>
<td>SWF loaded by HTML content using &lt;object&gt; or &lt;embed&gt; tags (or the equivalent JavaScript APIs)</td>
<td>SWF version of the loaded file</td>
</tr>
</tbody>
</table>

When loading a SWF file of a different version than the loading content, you can run into the two problems:

- Loading SWF10 content by SWF9 (or earlier) — References to AIR 1.5 and Flash Player 10 APIs in the loaded content will be unresolved
- Loading SWF9 (or earlier) content by SWF10 — APIs changed in AIR 1.5 and Flash Player 10 may behave in ways that the loaded content does not expect.

`minimumPatchLevel` Deprecated — do not use.

**Defining the application identity**

The following elements define the id, publisherID, version, name, filename, description, and copyright information:
<id>TestApp</id>
<publisherID>48B5E02D9FB21E6389F73B8D17023320485DF8CE.1</publisherID>
<version>2.0</version>
<filename>TestApp</filename>
<name>
  <text xml:lang="en">Hello AIR</text>
  <text xml:lang="fr">Bonjour AIR</text>
  <text xml:lang="es">Hola AIR</text>
</name>
<description>An MP3 player.</description>
<copyright>Copyright (c) 2008 YourCompany, Inc.</copyright>

**id** An identifier string for the application, known as the application ID. The attribute value is restricted to the following characters:
- 0–9
- a–z
- A–Z
- . (dot)
- - (hyphen)

The value must contain 1 to 212 characters. This element is required.

The id string uniquely identifies the application.

**publisherID** (Optional) Specifies the publisher ID to use when creating an update for an application published with AIR 1.5.2, or earlier. If the application to be updated has no publisher ID, omit the publisherID tag.

In order for an AIR file to update an existing version of an application, the application and publisher IDs of both the installed version and the update version must match. If the IDs are not the same, the user must uninstall the earlier version before they can install the new version.

Set this value to the existing publisher ID when publishing an update for an AIR application published for AIR 1.5.2 or earlier — or which targets AIR 1.5.2, or earlier, in the application descriptor namespace. The publisher ID for such applications is computed from the code signing certificate. You can find the ID value in the `META-INF/AIR/publisherid` file inside the installation directory of an application.

Publisher IDs are no longer computed or assigned automatically as of AIR 1.5.3. New applications do not need and should not specify a publisher ID.

The publisher ID tag was added in AIR 1.5.3. For more information about publisher IDs, see “About AIR publisher identifiers” on page 370.

**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", "4.4", "0.5", "4.9", "1.3.4a". This element is required.

**filename** The string to use as a filename of the application (without extension) when the application is installed. The application file launches the AIR application in the runtime. If no name value is provided, the filename is also used as the name of the installation folder. This element is required.

The filename property can contain any Unicode (UTF-8) character except the following, which are prohibited from use as filenames on various file systems:
The `filename` value cannot end in a period.

**name** (Optional, but recommended) The title displayed by the AIR application installer.

If you specify a single text node (instead of multiple `text` elements), the AIR application installer uses this name, regardless of the system language:

```xml
<name>Test Application</name>
```

The AIR 1.0 application descriptor schema allows only one simple text node to be defined for the name (not multiple `text` elements).

In AIR 1.1 (or above), you can specify multiple languages in the `name` element. For example, the following specifies the `name` in three languages (English, French, and Spanish):

```xml
<name>
  <text xml:lang="en">Hello AIR</text>
  <text xml:lang="fr">Bonjour AIR</text>
  <text xml:lang="es">Hola AIR</text>
</name>
```

The `xml:lang` attribute for each text element specifies a language code, as defined in [RFC4646](http://www.ietf.org/rfc/rfc4646.txt).

The AIR application installer uses the name that most closely matches the user interface language of the user’s operating system. For example, consider an installation in which the `name` element of the application descriptor file includes a value for the en (English) locale. The AIR application installer uses the en name if the operating system identifies en (English) as the user interface language. It also uses the en name if the system user interface language is en-US (U.S. English). However, if the user interface language is en-US and the application descriptor file defines both en-US and en-GB names, then the AIR application installer uses the en-US value. If the application defines no name that matches the system user interface languages, the AIR application installer uses the first `name` value defined in the application descriptor file.

If no `name` element is specified, the AIR application installer displays the `filename` as the application name.

The `name` element only defines the application title used in the AIR application installer. The AIR application installer supports multiple languages: Traditional Chinese, Simplified Chinese, Czech, Dutch, English, French, German, Italian, Japanese, Korean, Polish, Brazilian Portuguese, Russian, Spanish, Swedish, and Turkish. The AIR application installer selects its displayed language (for text other than the application title and description) based on the system user interface language. This language selection is independent of the settings in the application descriptor file.

### Character Set

<table>
<thead>
<tr>
<th>Character</th>
<th>Hexadecimal Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>various</td>
<td>0x00 – x1F</td>
</tr>
<tr>
<td>*</td>
<td>x2A</td>
</tr>
<tr>
<td>+</td>
<td>x22</td>
</tr>
<tr>
<td>:</td>
<td>x3A</td>
</tr>
<tr>
<td>&gt;</td>
<td>x3C</td>
</tr>
<tr>
<td>&lt;</td>
<td>x3E</td>
</tr>
<tr>
<td>?</td>
<td>x3F</td>
</tr>
<tr>
<td>\</td>
<td>x5C</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `filename` value cannot end in a period.
The name element does not define the locales available for the running, installed application. For details on developing multi-language applications, see “Localizing AIR applications” on page 395.

**description** (Optional) The description of the application, displayed in the AIR application installer.

If you specify a single text node (not multiple text elements), the AIR application installer uses this description, regardless of the system language:

```xml
<description>This is a sample AIR application.</description>
```

The AIR 1.0 application descriptor schema allows only one simple text node to be defined for the name (not multiple text elements).

In AIR 1.1 (or above), you can specify multiple languages in the description element. For example, the following specifies a description in three languages (English, French, and Spanish):

```xml
<description>
  <text xml:lang="en">This is a example.</text>
  <text xml:lang="fr">C'est un exemple.</text>
  <text xml:lang="es">Esto es un ejemplo.</text>
</description>
```

The xml:lang attribute for each text element specifies a language code, as defined in RFC4646 (http://www.ietf.org/rfc/rfc4646.txt).

The AIR application installer uses the description that most closely matches the user interface language of the user’s operating system. For example, consider an installation in which the description element of the application descriptor file includes a value the en (English) locale. The AIR application installer uses the en name if the user’s system identifies en (English) as the user interface language. It also uses the en name if the system user interface language is en-US (U.S. English). However, if system user interface language is en-US and the application descriptor file defines both en-US and en-GB names, then the AIR application installer uses the en-US value. If the application defines no name that matches the system user interface language, the AIR application installer uses the first description value defined in the application descriptor file.

For more information on developing multi-language applications, see “Localizing AIR applications” on page 395.

**copyright** (Optional) The copyright information for the AIR application. On Mac OS, the copyright text appears in the About dialog box for the installed application. On Mac OS, the copyright information is also used in the NSHumanReadableCopyright field in the Info.plist file for the application.

### Defining the installation folder and program menu folder

The installation and program menu folders are defined with the following property settings:

```xml
<installFolder>Acme</installFolder>
<programMenuFolder>Acme/Applications</programMenuFolder>
```

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

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

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

```xml
<installFolder>Acme/Power Tools</installFolder>
```
The `installFolder` property can contain any Unicode (UTF-8) character except those that are prohibited from use as folder names on various file systems (see the `filename` property above for the list of exceptions).

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.

`programMenuFolder` (Optional) Identifies the location in which to place shortcuts to the application in the All Programs menu of the Windows operating system or in the Applications menu on Linux. (This setting is currently ignored on other operating systems.) The restrictions on the characters that are allowed in the value of the property are the same as those for the `installFolder` property. Do not use a forward slash (/) character as the last character of this value.

### Defining the properties of the initial application window

When an AIR application is loaded, the runtime uses the values in the `initialWindow` element to create the initial window for the application. The runtime then loads the SWF or HTML file specified in the `content` element into the window.

Here is an example of the `initialWindow` element:

```xml
<initialWindow>
  <content>AIRTunes.html</content>
  <title>AIR Tunes</title>
  <systemChrome>none</systemChrome>
  <transparent>true</transparent>
  <visible>true</visible>
  <minimizable>true</minimizable>
  <maximizable>true</maximizable>
  <resizable>true</resizable>
  <width>400</width>
  <height>600</height>
  <x>150</x>
  <y>150</y>
  <minSize>300 300</minSize>
  <maxSize>800 800</maxSize>
</initialWindow>
```

The child elements of the `initialWindow` element set the properties of the window into which the root content file is loaded.

- **content** The value specified for the `content` element is the URL for the main content file of the application. This may be either a SWF file or an HTML file. The URL is specified relative to the root of the application installation folder. (When running an AIR application with ADL, the URL is relative to the folder containing the application descriptor file. You can use the `root-dir` parameter of ADL to specify a different root directory.)

  **Note:** Because the value of the `content` element is treated as a URL, characters in the name of the content file must be URL encoded according to the rules defined in RFC 1738. Space characters, for example, must be encoded as `%20`.  

- **title** (Optional) The window title.

- **systemChrome** (Optional) If you set this attribute to `standard`, the standard system chrome supplied by the operating system is displayed. If you set it to `none`, no system chrome is displayed. The system chrome setting cannot be changed at run time.

- **transparent** (Optional) Set to "true" if you want the application window to support alpha blending. A window with transparency may draw more slowly and require more memory. The transparent setting cannot be changed at run time.
**Important:** You can only set `transparent` to `true` when `systemChrome` is `none`.

**visible** (Optional) Set to `true` if you want the main window to be visible as soon as it is created. The default value is `false`.

You may want to leave the main window hidden initially, so that changes to the window’s position, the window’s size, and the layout of its contents are not shown. You can then display the window by calling the `activate()` method of the window or by setting the `visible` property to `true`. For details, see “Working with native windows” on page 132.

**x, y, width, height** (Optional) The initial bounds of the main window of the application. If you do not set these values, the window size is determined by the settings in the root SWF file or, in the case of HTML, by the operating system. The maximum values for `width` and `height` are each 2880.

**minSize, maxSize** (Optional) The minimum and maximum sizes of the window. If you do not set these values, they are determined by the operating system.

**minimizable, maximizable, resizable** (Optional) Specifies whether the window can be minimized, maximized, and resized. By default, these settings default to `true`.

**Note:** On operating systems, such as Mac OS X, for which maximizing windows is a resizing operation, both `maximizable` and `resizable` must be set to `false` to prevent the window from being zoomed or resized.

### 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 displays a default icon.

The path specified is relative to the application root directory. Icon files must be in the PNG format. You can specify all of the following icon sizes:

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

If an element for a given size is present, the image in the file must be exactly the size specified. If all sizes are not provided, the closest size is 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.

For best results, provide an image for each of the available sizes. In addition, make sure that the icons look presentable in both 16- and 32-bit color modes.

### Providing a custom user interface for application updates

AIR installs and updates applications using the default installation dialogs. However, you can provide your own user interface for updating an application. To indicate that your application should handle the update process itself, set the `customUpdateUI` element to `true`:

```xml
<customUpdateUI>true</customUpdateUI>
```
When the installed version of your application has the `customUpdateUI` element set to `true` and the user then double-clicks the AIR file for a new version or installs an update of the application using the seamless install feature, 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. (The update AIR file must be signed with the same certificate as the installed application for an upgrade to proceed.)

**Note:** The `customUpdateUI` mechanism only comes into play when the application is already installed and the user double-clicks the AIR installation file containing an update or installs an update of the application using the seamless install feature. You can download and start an update through your own application logic, displaying your custom UI as necessary, whether or not `customUpdateUI` is `true`.

For more information, see “Updating AIR applications” on page 376.

### Allowing browser invocation of the application

If you specify the following setting, the installed AIR application can be launched via the browser invocation feature (by the user clicking a link in a page in a web browser):

```xml
<allowBrowserInvocation>true</allowBrowserInvocation>
```

The default value is `false`.

If you set this value to `true`, be sure to consider security implications, described in “Browser invocation” on page 332.

For more information, see “Installing and running AIR applications from a web page” on page 361.

### Declaring file type associations

The `fileTypes` element allows you to declare the file types with which an AIR application can be associated. When an AIR application is installed, any declared file type is registered with the operating system and, if these file types are not already associated with another application, they are associated with the AIR application. To override an existing association between a file type and another application, use the `NativeApplication.setAsDefaultApplication()` method at run time (preferably with the user’s permission).

**Note:** The runtime methods can only manage associations for the file types declared in the application descriptor.

```xml
<fileTypes>
  <fileType>
    <name>adobe.VideoFile</name>
    <extension>avf</extension>
    <description>Adobe Video File</description>
    <contentType>application/vnd.adobe.video-file</contentType>
    <icon>
      <image16x16>icons/AIRApp_16.png</image16x16>
      <image32x32>icons/AIRApp_32.png</image32x32>
      <image48x48>icons/AIRApp_48.png</image48x48>
      <image128x128>icons/AIRApp_128.png</image128x128>
    </icon>
  </fileType>
</fileTypes>
```

The `fileTypes` element is optional. It may contain any number of `fileType` elements.
The `name` and `extension` elements are required for each `fileType` declaration that you include. The same name can be used for multiple extensions. The extension uniquely identifies the file type. (Note that the extension is specified without the preceding period.) The `description` element is optional and is displayed to the user by the operating system user interface. The `contentType` is required in AIR 1.5 (it was optional in AIR 1.0 and 1.1). The property helps the operating system to locate the best application to open a file under some circumstances. The value should be the MIME type of the file content. Note that the value is ignored on Linux if the file type is already registered and has an assigned MIME type.

Icons can be specified for the file extension, using the same format as the application icon element. The icon files must also be included in the AIR installation file (they are not packaged automatically).

When a file type is associated 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 `InvokeEvent` object to the running instance. Otherwise, AIR will launch the application first. In both cases, the path to the file can be retrieved from the `InvokeEvent` object dispatched by the `NativeApplication` object. You can use this path to open the file.

For more information, see “Managing file associations” on page 337 and “Capturing command line arguments” on page 329.
Chapter 16: ActionScript basics for JavaScript developers

Adobe® ActionScript® 3.0 is a programming language like JavaScript—both are based on ECMAScript. ActionScript 3.0 was released with Adobe® Flash® Player 9 and you can therefore develop rich Internet applications with it in Adobe® Flash® CS3 Professional, Adobe® Flash® CS4 Professional, and Adobe® Flex™ 3.

The current version of ActionScript 3.0 was available only when developing SWF content for Flash Player 9 in the browser. It is now also available for developing SWF content running in Adobe® AIR®.

The Adobe AIR Language Reference for HTML Developers includes documentation for those classes that are useful in JavaScript code in an HTML-based application. It’s a subset of the entire set of classes in the runtime. Other classes in the runtime are useful in developing SWF-based applications (the DisplayObject class for example, which defines the structure of visual content). If you need to use these classes in JavaScript, refer to the following ActionScript documentation:

• Programming Adobe ActionScript 3.0
• The Flex 3 Language Reference. (Only the top-level classes and functions in the flash package are available to HTML content running in AIR. The classes in the mx package are available only in Flex-based SWF applications.)

Differences between ActionScript and JavaScript: an overview

ActionScript, like JavaScript, is based on the ECMAScript language specification; therefore, the two languages have a common core syntax. For example, the following code works the same in JavaScript and in ActionScript:

```javascript
var str1 = "hello";
var str2 = " world.";
var str = reverseString(str1 + str2);

function reverseString(s) {
    var newString = "";
    var i;
    for (i = s.length - 1; i >= 0; i--) {
        newString += s.charAt(i);
    }
    return newString;
}
```

However, there are differences in the syntax and workings of the two languages. For example, the preceding code example can be written as the following in ActionScript 3.0 (in a SWF file):

```actionscript
function reverseString(s:String):String {
    var newString:String = "";
    for (var i:int = s.length - 1; i >= 0; i--) {
        newString += s.charAt(i);
    }
    return newString;
}
```
The version of JavaScript supported in HTML content in Adobe AIR is JavaScript 1.7. The differences between JavaScript 1.7 and ActionScript 3.0 are described throughout this topic.

The runtime includes some built-in classes that provide advanced capabilities. At runtime, JavaScript in an HTML page can access those classes. The same runtime classes are available both to ActionScript (in a SWF file) and JavaScript (in an HTML file running in a browser). However, the current API documentation for these classes (which are not included in the Adobe AIR Language Reference for HTML Developer) describes them using ActionScript syntax. In other words, for some of the advanced capabilities of the runtime, refer to the Adobe AIR ActionScript 3.0 Language Reference. Understanding the basics of ActionScript helps you understand how to use these runtime classes in JavaScript.

For example, the following JavaScript code plays sound from an MP3 file:

```javascript
var file = air.File.userDirectory.resolve("My Music/test.mp3");
var sound = air.Sound(file);
sound.play();
```

Each of these lines of code calls runtime functionality from JavaScript.

In a SWF file, ActionScript code can access these runtime capabilities as in the following code:

```actionscript
var file:File = File.userDirectory.resolve("My Music/test.mp3");
var sound = new Sound(file);
sound.play();
```

### ActionScript 3.0 data types

ActionScript 3.0 is a strongly typed language. That means that you can assign a data type to a variable. For example, the first line of the previous example could be written as the following:

```actionscript
var str1:String = "hello";
```

Here, the `str1` variable is declared to be of type String. All subsequent assignments to the `str1` variable assign String values to the variable.

You can assign types to variables, parameters of functions, and return types of functions. Therefore, the function declaration in the previous example looks like the following in ActionScript:

```actionscript
function reverseString(s:String):String {
    var newString:String = "";
    for (var i:int = s.length - 1; i >= 0; i--) {
        newString += s.charAt(i);
    }
    return newString;
}
```

**Note:** The `s` parameter and the return value of the function are both assigned the type String.

Although assigning types is optional in ActionScript, there are advantages to declaring types for objects:

- Typed objects allow for type checking of data at not only at run-time, but also at compile time if you use strict mode, which helps identify errors. (Strict mode is a compiler option.)
- Using typed objects creates applications that are more efficient.

For this reason, the examples in the ActionScript documentation use data types. Often, you can convert sample ActionScript code to JavaScript by simply removing the type declarations (such as `":String"`).
**Data types corresponding to custom classes**

An ActionScript 3.0 object can have a data type that corresponds to the top-level classes, such as String, Number, or Date.

In ActionScript 3.0, you can define custom classes. Each custom class also defines a data type. This means that an ActionScript variable, function parameter, or function return can have a type annotation defined by that class. For more information, see “Custom ActionScript 3.0 classes” on page 128.

**The void data type**

The void data type is used as the return value for a function that, in fact, returns no value (a function that does not include a return statement).

**The * data type**

Use of the asterisk character (*) as a data type is the same as not assigning a data type. For example, the following function includes a parameter, n, and a return value that are both not given a data type:

```actionscript
function exampleFunction(n:*):* {
    trace("hi, ", n);
}
```

Use of the * as a data type is not defining a data type at all. You use the asterisk in ActionScript 3.0 code to be explicit that no data type is defined.

**ActionScript 3.0 classes, packages, and namespaces**

ActionScript 3.0 includes capabilities related to classes that are not found in JavaScript 1.7.

**Runtime classes**

The runtime includes built-in classes, many of which are also included in standard JavaScript, such as the Array, Date, Math, and String classes (and others). However, the runtime also includes classes that are not found in standard JavaScript; classes that have a variety of uses, from playing rich media (such as sounds) to interacting with sockets.

Most runtime classes are in the flash package, or one of the packages contained by the flash package. Packages are a means to organize ActionScript 3.0 classes (see “ActionScript 3.0 packages” on page 129.

**Custom ActionScript 3.0 classes**

ActionScript 3.0 allows developers to create their own custom classes. For example, the following code defines a custom class named ExampleClass:

```actionscript
public class ExampleClass {
    public var x:Number;
    public function ExampleClass(input:Number):void {
        x = input;
    }
    public function greet():void {
        trace("The value of x is: ", x);
    }
}
```
This class has the following members:

- A constructor method, `ExampleClass()`, which lets you instantiate new objects of the `ExampleClass` type.
- A public property, `x` (of type `Number`), which you can get and set for objects of type `ExampleClass`.
- A public method, `greet()`, which you can call on objects of type `ExampleClass`.

  In this example, the `x` property and the `greet()` method are in the `public` namespace, which makes them accessible from objects and classes outside of the class.

### ActionScript 3.0 Packages

Packages provide the means to arrange ActionScript 3.0 classes. For example, many classes related to working with files and directories on the computer on which an AIR application is installed are included in the `flash.filesystem` package. In this case, `flash` is one package that contains another package, `filesystem`. And that package may contain other classes or packages. In fact, the `flash.filesystem` package contains the following classes: `File`, `FileMode`, and `FileStream`. To reference the `File` class in ActionScript, you can write the following:

```actionscript
flash.filesystem.File
```

Both built-in and custom classes can be arranged in packages.

When referencing an ActionScript package from JavaScript, use the special `runtime` object. For example, the following code instantiates a new ActionScript `File` object in JavaScript:

```javascript
var myFile = new air.flash.filesystem.File();
```

Here, the `File()` method is the constructor function corresponding to the class of the same name (`File`).

### ActionScript 3.0 Namespaces

In ActionScript 3.0, namespaces define the scope for which properties and functions in classes can be accessed.

Only those properties and methods in the `public` namespace are available in JavaScript.

For example, the `File` class (in the `flash.filesystem` package) includes `public` properties and methods, such as `userDirectory` and `resolve()`. Both are available as properties of a JavaScript variable that instantiates a `File` object (via the `runtime.flash.filesystem.File()` constructor method).

There are four predefined namespaces:

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Any code that instantiates an object of a certain type can access the public properties and methods in the class that defines that type. Also, any code can access the public static properties and methods of a public class.</td>
</tr>
<tr>
<td>private</td>
<td>Properties and methods designated as private are only available to code within the class. They cannot be accessed as properties or methods of an object defined by that class. Properties and methods in the private namespace are not available in JavaScript.</td>
</tr>
<tr>
<td>protected</td>
<td>Properties and methods designated as protected are only available to code in the class definition and to classes that inherit that class. Properties and methods in the protected namespace are not available in JavaScript.</td>
</tr>
<tr>
<td>internal</td>
<td>Properties and methods designated as internal are available to any caller within the same package. Classes, properties, and methods belong to the internal namespace by default.</td>
</tr>
</tbody>
</table>

Additionally, custom classes can use other namespaces that are not available to JavaScript code.
Required parameters and default values in ActionScript 3.0 functions

In both ActionScript 3.0 and JavaScript, functions can include parameters. In ActionScript 3.0, parameters can be required or optional; whereas in JavaScript, parameters are always optional.

The following ActionScript 3.0 code defines a function for which the one parameter, n, is required:

```actionscript
define(n:Number):Number {
    return n*n*n;
}
```

The following ActionScript 3.0 code defines a function for which the n parameter is required, and for which the p parameter is optional, with a default value of 1:

```actionscript
define(n:Number, p:Number = 1):Number {
    return Math.pow(n, 1/p);
}
```

An ActionScript 3.0 function can also receive any number of arguments, represented by `...rest` syntax at the end of a list of parameters, as in the following:

```actionscript
define(average(..., args) : Number{
    var sum:Number = 0;
    for (var i:int = 0; i < args.length; i++) {
        sum += args[i];
    }
    return (sum / args.length);
}
```

ActionScript 3.0 event listeners

In ActionScript 3.0 programming, all events are handled using `event listeners`. An event listener is a function. When an object dispatches an event, the event listener responds to the event. The event, which is an ActionScript object, is passed to the event listener as a parameter of the function, which differs from the DOM event model used in JavaScript.

For example, when you call the `load()` method of a Sound object (to load an MP3 file), the Sound object attempts to load the sound and then dispatch any of the following events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>complete</td>
<td>When the data has loaded successfully.</td>
</tr>
<tr>
<td>id3</td>
<td>When MP3 ID3 data is available.</td>
</tr>
<tr>
<td>ioError</td>
<td>When an input/output error occurs that causes a load operation to fail.</td>
</tr>
<tr>
<td>open</td>
<td>When the load operation starts.</td>
</tr>
<tr>
<td>progress</td>
<td>When data is received as a load operation progresses.</td>
</tr>
</tbody>
</table>

Any class that can dispatch events either extends the EventDispatcher class or implements the IEventDispatcher interface. (An ActionScript 3.0 interface is a data type used to define a set of methods that can be implemented by a class.) In each class listing for these classes in the ActionScript Language Reference, there is a list of events that the class can dispatch.
You can register an event listener function to handle any of these events, using the `addEventListener()` method of the object that dispatches the event. For example, in the case of a Sound object, you can register for the `progress` and `complete` events, as shown in the following ActionScript code:

```actionscript
class MySoundClass {
    constructor() {
        var sound:Sound = new Sound();
        var urlReq:URLRequest = new URLRequest("test.mp3");
        sound.load(urlReq);
        sound.addEventListener(ProgressEvent.PROGRESS, progressHandler);
        sound.addEventListener(Event.COMPLETE, completeHandler);

        function progressHandler(progressEvent):void {
            trace("Progress " + progressEvent.bytesTotal + " bytes out of " + progressEvent.bytesTotal);
        }

        function completeHandler(completeEvent):void {
            trace("Sound loaded.");
        }
    }
}
```

In HTML content running in AIR, you can register a JavaScript function as the event listener, as shown in the following code (which assumes that the HTML document includes a `textarea` object named `progressTextArea`):

```javascript
var sound = new runtime.flash.media.Sound();
var urlReq = new runtime.flash.net.URLRequest("test.mp3");
sound.load(urlReq);
sound.addEventListener(runtime.flash.events.ProgressEvent.PROGRESS, progressHandler);
sound.addEventListener(runtime.flash.events.Event.COMPLETE, completeHandler);

function progressHandler(progressEvent) {
    document.progressTextArea.value += "Progress " + progressEvent.bytesTotal + " bytes out of " + progressEvent.bytesTotal;
}

function completeHandler(completeEvent) {
    document.progressTextArea.value += "Sound loaded.";
}
```
Chapter 17: Working with native windows

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

Additional online information about native windows

You can find more information about the native window API and working with native windows from these sources:

Quick Starts (Adobe AIR Developer Connection)
- Customizing the look and feel of a window

Language Reference
- NativeWindow
- NativeWindowInitOptions

Adobe Developer Connection Articles and Samples
- Adobe AIR Developer Connection for HTML and Ajax (search for 'AIR windows')

AIR window basics

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

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

Windows in AIR

AIR supports three distinct APIs for working with windows:

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

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

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

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

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

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

For more information about the window properties specified in the application descriptor, see "The application descriptor file structure" on page 116.

Native window classes
The native window API contains the following classes:
Native window event flow

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

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

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

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

b. A `resize` event is dispatched if the window size changed because of the maximize operation.

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

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

For detailed information about the window API classes, methods, properties, and events, see the Adobe AIR Language Reference for HTML Developers (http://www.adobe.com/go/learn_air_html_jslr).
For general information about using the Flash display list, see the “Display Programming” section of the Programming Adobe ActionScript 3.0 (http://www.adobe.com/go/learn_fl_cs4_programmingAS3_en) reference.

Properties controlling native window style and behavior
The following properties control the basic appearance and behavior of a window:

- **type**
- **systemChrome**
- **transparent**

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

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

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

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

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

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

System chrome is managed by the system. Your application has no direct access to the controls themselves, but can react to the events dispatched when the controls are used. When you use standard chrome for a window, the transparent property must be set to false and the type property must be normal or utility.
Custom chrome
When you create a window with no system chrome, then you must add your own chrome controls to handle the
interactions between a user and the window. You are also free to make transparent, non-rectangular windows.

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

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

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

Transparency cannot be used with windows that have system chrome. In addition, SWF content and PDF content in
HTML does not display in transparent windows. For more information, see “Considerations when loading SWF or
PDF content in an HTML page” on page 90.

On some operating systems, transparency might not be supported because of hardware or software configuration, or
user display options. When transparency is not supported, the application is composited against a black background.
In these cases, any transparent areas of the application display as an opaque black.

The static `NativeWindow.supportsTransparency` property reports whether window transparency is available. If
this property tests `false`, for example, you could display a warning dialog to the user, or display a fallback, rectangular,
non-transparent user interface. Note that transparency is always supported by the Mac and Windows operating
systems. Support on Linux operating systems requires a compositing window manager, but even when a compositing
window manager is active, transparency can be unavailable because of user display options or hardware configuration.

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

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

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

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

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

A visual window catalog
The following table illustrates the visual effects of different combinations of window property settings on the Mac OS
X, Windows, and Linux operating systems:
## Working with native windows

<table>
<thead>
<tr>
<th>Window settings</th>
<th>Mac OS X</th>
<th>Microsoft Windows</th>
<th>Linux*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: normal</td>
<td><img src="image1" alt="Mac OS X window" /></td>
<td><img src="image2" alt="Microsoft Windows window" /></td>
<td><img src="image3" alt="Linux window" /></td>
</tr>
<tr>
<td>SystemChrome: standard</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transparent: false</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type: utility</td>
<td><img src="image4" alt="Mac OS X window" /></td>
<td><img src="image5" alt="Microsoft Windows window" /></td>
<td><img src="image6" alt="Linux window" /></td>
</tr>
<tr>
<td>SystemChrome: standard</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transparent: false</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type: Any</td>
<td><img src="image7" alt="Mac OS X window" /></td>
<td><img src="image8" alt="Microsoft Windows window" /></td>
<td><img src="image9" alt="Linux window" /></td>
</tr>
<tr>
<td>SystemChrome: none</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transparent: false</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type: Any</td>
<td><img src="image10" alt="Mac OS X window" /></td>
<td><img src="image11" alt="Microsoft Windows window" /></td>
<td><img src="image12" alt="Linux window" /></td>
</tr>
<tr>
<td>SystemChrome: none</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transparent: true</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mxWindowedApplication or mx:Window</td>
<td><img src="image13" alt="Mac OS X window" /></td>
<td><img src="image14" alt="Microsoft Windows window" /></td>
<td><img src="image15" alt="Linux window" /></td>
</tr>
<tr>
<td>Type: Any</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SystemChrome: none</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transparent: true</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Creating windows

AIR automatically creates the first window for an application, but you can create any additional windows you need. To create a native window, use the NativeWindow constructor method. To create an HTML window, either use the HTMLLoader createRootWindow() method or, from an HTML document, call the JavaScript window.open() method.

Specifying window initialization properties

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>systemChrome</td>
<td>standard</td>
</tr>
<tr>
<td>type</td>
<td>normal</td>
</tr>
<tr>
<td>transparent</td>
<td>false</td>
</tr>
<tr>
<td>maximizable</td>
<td>true</td>
</tr>
<tr>
<td>minimizable</td>
<td>true</td>
</tr>
<tr>
<td>resizable</td>
<td>true</td>
</tr>
</tbody>
</table>

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

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

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

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

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

Note: You cannot set the initialization properties for a window created with the JavaScript window.open() function. You can, however, override how these windows are created by implementing your own HTMLHost class. See “Handling JavaScript calls to window.open()” on page 96 for more information.
Creating the initial application window

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

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

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

```html
<html>
<head>
  <script language="javascript" type="text/javascript" src="AIRAliases.js"></script>
  <script language="javascript">
    window.onload=init;
    function init(){
      window.nativeWindow.activate();
    }
  </script>
</head>
<body></body>
</html>
```

Creating a NativeWindow

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

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

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

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

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

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

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

```javascript
var maxOSSize = air.NativeWindow.systemMaxSize;
var minOSSize = air.NativeWindow.systemMinSize;
```
Creating an HTML window

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

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

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

**Note:** You can extend the `HTMLHost` class in ActionScript to customize the window created with the JavaScript `window.open()` function. See “About extending the `HTMLHost` class” on page 93.

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

```javascript
var options = new air.NativeWindowInitOptions();
options.systemChrome = "none";
options.type = "lightweight";

var windowBounds = new air.Rectangle(200,250,300,400);
newHTMLLoader = air.HTMLLoader.createRootWindow(true, options, true, windowBounds);
newHTMLLoader.load(new air.URLRequest("xmpl.html"));
```

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

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

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

Adding content to a window

How you add content to an AIR window depends on the type of window. HTML lets you declaratively define the basic content of the window in a text file. You can load a variety of resources from separate application files. HTML and Flash content can be created on the fly and added to a window dynamically.

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

To allow safe cross-scripting, you can use a sandbox bridge to provide a limited interface between application content and non-application content. In HTML content, you can also map pages of your application into a non-application sandbox to allow the code on that page to cross-script external content. See “AIR security” on page 100.
Loading a SWF file or image
You can load Flash SWF files or images into the display list of a native window using the `flash.display.Loader` class:

```ActionScript
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);
    }
}
```

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

```ActionScript
//newWindow is a NativeWindow instance
var htmlView:HTMLLoader = new HTMLLoader();
htmlView.width = 300;
htmlView.height = 500;
//set the stage so display objects are added to the top-left and not scaled
newWindow.stage.align = "TL";
newWindow.stage.scaleMode = "noScale";
newWindow.stage.addChild( htmlView );

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

**Note:** SWF content or PDF content in an HTML file is not displayed if the window uses transparency (that is the `transparent` property of the window is `true`) or if the HTMLLoader control is scaled.

Adding SWF content as an overlay on an HTML window
Because HTML windows are contained within a NativeWindow instance, you can add Flash display objects both above and below the HTML layer in the display list.

To add a display object above the HTML layer, use the `addChild()` method of the `window.nativeWindow.stage` property. The `addChild()` method adds content layered above any existing content in the window.
To add a display object below the HTML layer, use the `addChildAt()` method of the `window.nativeWindow.stage` property, passing in a value of zero for the `index` parameter. Placing an object at the zero index moves existing content, including the HTML display, up one layer and insert the new content at the bottom. For content layered underneath the HTML page to be visible, you must set the `paintsDefaultBackground` property of the `HTMLLoader` object to `false`. In addition, any elements of the page that set a background color, will not be transparent. If, for example, you set a background color for the body element of the page, none of the page will be transparent.

The following example illustrates how to add a Flash display objects as overlays and underlays to an HTML page. The example creates two simple shape objects, adds one below the HTML content and one above. The example also updates the shape position based on the `enterFrame` event.

```html
<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;
        }
    }

</script>
</head>
<body>
</body>
</html>
```
function init(){
  //turn off the default HTML background
  window.htmlLoader.paintsDefaultBackground = false;
  var bottom = new Bouncer(60,0xff2233);
  var top = new Bouncer(30,0x2441ff);

  //listen for the enterFrame event
  window.htmlLoader.addEventListener("enterFrame",function(evt){
    bottom.update();
    top.update();
  });

  //add the bouncing shapes to the window stage
  window.nativeWindow.stage.addChildAt(bottom.shape,0);
  window.nativeWindow.stage.addChild(top.shape);
}
</script>
<body onload="init();">
<h1>de Finibus Bonorum et Malorum</h1>
<p>Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo.</p>
<p>This paragraph has a background color.</p>
<p>At vero eos et accusamus et iusto odio dignissimos ducimus qui blanditiis praesentium voluptatum deleniti atque corrupti quos dolores et quas molestias excepturi sint occaecati cupiditate non provident, similique sunt in culpa qui officia deserunt mollitia animi, id est laborum et dolorum fuga.</p>
</body>
</html>

Note: To access the runtime, nativeWindow and htmlLoader properties of the JavaScript Window object, the HTML page must be loaded from the application directory. This will always be the case for the root page in an HTML-based application, but may not be true for other content. In addition, documents loaded into frames even within the application sandbox do not receive these properties, but can access those of the parent document.

Example: Creating a native window

The following example illustrates how to create a native window:
function createNativeWindow() { 
   //create the init options 
   var options = new air.NativeWindowInitOptions();
   options.transparent = false;
   options.systemChrome = air.NativeWindowSystemChrome.STANDARD;
   options.type = air.NativeWindowType.NORMAL;

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

   //activate and show the new window 
   newWindow.activate();
}

Managing windows

You use the properties and methods of the NativeWindow class to manage the appearance, behavior, and life cycle of desktop 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 native window constructor used to create the window:
  var nativeWin = new air.NativeWindow(initOptions);

- The stage property of a display object in the window:
  var nativeWin = window.htmlLoader.stage.nativeWindow;

- The target property of a native window event dispatched by the window:
  function onNativeWindowEvent(event) 
  { 
      var nativeWin = event.target;
  }

- The nativeWindow property of an HTML page displayed in the window:
  var nativeWin = window.nativeWindow;

- The activeWindow and openedWindows properties of the NativeApplication object:
  var win = NativeApplication.nativeApplication.activeWindow;
  var firstWindow = NativeApplication.nativeApplication.openedWindows[0];

NativeApplication.nativeApplication.activeWindow references the active window of an application (but returns null if the active window is not a window of this AIR application). The NativeApplication.nativeApplication.openedWindows array contains all of the windows in an AIR application that have not been closed.
Activating, showing, and hiding windows

To activate a window, call the `NativeWindow.activate()` method. Activating a window brings the window to the front, gives it keyboard and mouse focus, and, if necessary, makes it visible by restoring the window or setting the `visible` property to `true`. Activating a window does not change the ordering of other windows in the application. Calling the `activate()` method causes the window to dispatch an `activate` event.

To show a hidden window without activating it, set the `visible` property to `true`. This brings the window to the front, but will not assign the focus to the window.

To hide a window from view, set its `visible` property to `false`. Hiding a window suppresses the display of both the window, any related taskbar icons, and, on Mac OS X, the entry in the Windows menu.

**Note:** On Mac OS X, it is not possible to completely hide a minimized window that has an icon in the window portion of the dock. If the `visible` property is set to `false` on a minimized window, the dock icon for the window is still displayed. If the user clicks the icon, the window is restored to a visible state and displayed.

Changing the window display order

AIR provides several methods for directly changing the display order of windows. You can move a window to the front of the display order or to the back; you can move a window above another window or behind it. At the same time, the user can reorder windows by activating them.

You can keep a window in front of other windows by setting its `alwaysInFront` property to `true`. If more than one window has this setting, then the display order of these windows is sorted among each other, but they are always sorted above windows which have `alwaysInFront` set to false. Windows in the top-most group are also displayed above windows in other applications, even when the AIR application is not active. Because this behavior can be disruptive to a user, setting `alwaysInFront` to `true` should only be done when necessary and appropriate. Examples of justified uses include:

- Temporary pop-up windows for controls such as tool tips, pop-up lists, custom menus, or combo boxes. Because these windows should close when they lose focus, the annoyance of blocking a user from viewing another window can be avoided.
- Extremely urgent error messages and alerts. When an irrevocable change may occur if the user does not respond in a timely manner, it may be justified to push an alert window to the forefront. However, most errors and alerts can be handled in the normal window display order.
- Short-lived toast-style windows.

**Note:** AIR does not enforce proper use of the `alwaysInFront` property. However, if your application disrupts a user’s workflow, it is likely to be consigned to that same user’s trash can.

The `NativeWindow` class provides the following properties and methods for setting the display order of a window relative to other windows:

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>alwaysInFront</code> property</td>
<td>Specifies whether the window is displayed in the top-most group of windows.</td>
</tr>
<tr>
<td></td>
<td>In almost all cases, <code>false</code> is the best setting. Changing the value from <code>false</code> to <code>true</code> brings the window to the front of all windows (but does not activate it). Changing the value from <code>true</code> to <code>false</code> orders the window behind windows remaining in the top-most group, but still in front of other windows. Setting the property to its current value for a window does not change the window display order.</td>
</tr>
<tr>
<td><code>orderToFront()</code></td>
<td>Brings the window to the front.</td>
</tr>
<tr>
<td><code>orderInFrontOf()</code></td>
<td>Brings the window directly in front of a particular window.</td>
</tr>
</tbody>
</table>
Note: If a window is hidden (visible is false) or minimized, then calling the display order methods has no effect.

On the Linux operating system, different window managers enforce different rules regarding the window display order:

- On some window managers, utility windows are always displayed in front of normal windows.
- On some window managers, a full screen window with alwaysInFront set to true is always displayed in front of other windows that also have alwaysInFront set to true.

### Closing a window

To close a window, use the `NativeWindow.close()` method.

Closing a window unloads the contents of the window, but if other objects have references to this content, the content objects will not be destroyed. The `NativeWindow.close()` method executes asynchronously, the application that is contained in the window continues to run during the closing process. The close method dispatches a close event when the close operation is complete. The NativeWindow object is still technically valid, but accessing most properties and methods on a closed window generates an IllegalOperationError. You cannot reopen a closed window. Check the `closed` property of a window to test whether a window has been closed. To simply hide a window from view, set the `NativeWindow.visible` property to false.

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

### 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 does not close.

When a window does not use system chrome, notification events for intended changes are not automatically dispatched before the change is made. Hence, if you call the methods for closing a window, changing the window state, or set any of the window bounds properties, the change cannot be canceled. To notify components in your application before a window change is made, your application logic can dispatch the relevant notification event using the `dispatchEvent()` method of the window.

```javascript
function onCloseCommand(event){
    var closingEvent = new air.Event(air.Event.CLOSING,true,true);
    dispatchEvent(closingEvent);
    if(!closingEvent.isDefaultPrevented()){ 
        win.close();
    }
}
```

The `dispatchEvent()` method returns `false` if the event `preventDefault()` method is called by a listener. However, it can also return `false` for other reasons, so it is better to explicitly use the `isDefaultPrevented()` method to test whether the change should be canceled.
Maximizing, minimizing, and restoring a window

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

window.nativeWindow.maximize();

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

window.nativeWindow.minimize();

To restore the window (that is, return it to the size that it was before it was either minimized or maximized), use the NativeWindow restore() method.

window.nativeWindow.restore();

Note: The behavior that results from maximizing an AIR window is different from the Mac OS X standard behavior. Rather than toggling between an application-defined "standard" size and the last size set by the user, AIR windows toggle between the size last set by the application or user and the full usable area of the screen.

On the Linux operating system, different window managers enforce different rules regarding setting the window display state:

- On some window managers, utility windows cannot be maximized.
- If a maximum size is set for the window, then some windows do not allow a window to be maximized. Some other window managers set the display state to maximized, but do not resize the window. In either of these cases, no display state change event is dispatched.
- Some window managers do not honor the window maximizable or minimizable settings.

Note: On Linux, window properties are changed asynchronously. If you change the display state in one line of your program, and read the value in the next, the value read will still reflect the old setting. On all platforms, the NativeWindow object dispatches the displayStateChange event when the display state changes. If you need to take some action based on the new state of the window, always do so in a displayStateChange event handler. See "Listening for window events" on page 150.

Example: Minimizing, maximizing, restoring and closing a window

The following short HTML page demonstrates the NativeWindow maximize(), minimize(), restore(), and close() methods:
Resizing and moving a window

When a window uses system chrome, the chrome provides drag controls for resizing the window and moving around the desktop. If a window does not use system chrome you must add your own controls to allow the user to resize and move the window.

Note: To resize or move a window, you must first obtain a reference to the NativeWindow instance. For information about how to obtain a window reference, see “Getting a NativeWindow instance” on page 144.

Resizing a window

To allow a user to resize a window interactively, use the NativeWindow startResize() method. When this method is called from a mouseDown event, the resizing operation is driven by the mouse and completes when the operating system receives a mouseUp event. When calling startResize(), you pass in an argument that specifies the edge or corner from which to resize the window.

To set the window size programmatically, set the width, height, or bounds properties of the window to the desired dimensions. When you set the bounds, the window size and position can all be changed at the same time. However, the order that the changes occur is not guaranteed. Some Linux window managers do not allow windows to extend outside the bounds of the desktop screen. In these cases, the final window size may be limited because of the order in which the properties are set, even though the net affect of the changes would otherwise have resulted in a legal window. For example, if you change both the height and y position of a window near the bottom of the screen, then the full height change might not occur when the height change is applied before the y position change.
Note: On Linux, window properties are changed asynchronously. If you resize a window in one line of your program, and read the dimensions in the next, they will still reflect the old settings. On all platforms, the NativeWindow object dispatches the resize event when the window resizes. If you need to take some action, such as laying out controls in the window, based on the new size or state of the window, always do so in a resize event handler. See “Listening for window events” on page 150.

Moving a window
To move a window without resizing it, use the NativeWindow startMove() method. Like the startResize() method, when the startMove() method is called from a mouseDown event, the move process is mouse-driven and completes when the operating system receives a mouseUp event.

For more information about the startResize() and startMove() methods, see the Adobe AIR Language Reference for HTML Developers (http://www.adobe.com/go/learn_air_html_jslr).

To move a window programmatically, set the x, y, or bounds properties of the window to the desired position. When you set the bounds, the window size and position can both be changed at the same time.

Note: On Linux, window properties are changed asynchronously. If you move a window in one line of your program, and read the position in the next, the value read will still reflect the old setting. On all platforms, the NativeWindow object dispatches the move event when the position changes. If you need to take some action based on the new position of the window, always do so in a move event handler. See “Listening for window events” on page 150.

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

```html
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<script src="AIRAliases.js"></script>
<script type="text/javascript">
function onResize(type){
    nativeWindow.startResize(type);
}

function onNativeMove(){
    nativeWindow.startMove();
}
</script>
<style type="text/css" media="screen">
    .drag {
        width:200px;
        height:200px;
        margin:0px auto;
        padding:15px;
        border:1px dashed #333;
        background-color:#eee;
    }

    .resize {
        background-color:#FF0000;
        padding:10px;
    }

    .left {
        float:left;
    }
</style>
</head>
```

```
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:

```javascript
window.nativeWindow.addEventListener(air.Event.CLOSING, onClosingEvent);
```

When an event is dispatched, the `target` property references the window sending the event.

Most window events have two related messages. The first message signals that a window change is imminent (and can be canceled), while the second message signals that the change has occurred. For example, when a user clicks the close button of a window, the closing event message is dispatched. If no listeners cancel the event, the window closes and the close event is dispatched to any listeners.

Typically, the warning events, such as `closing`, are only dispatched when system chrome has been used to trigger an event. Calling the window `close()` method, for example, does not automatically dispatch the `closing` event—only the `close` event is dispatched. You can, however, construct a closing event object and dispatch it using the window `dispatchEvent()` method.

The window events that dispatch an Event object are:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>activate</td>
<td>Dispatched when the window receives focus.</td>
</tr>
<tr>
<td>deactivate</td>
<td>Dispatched when the window loses focus.</td>
</tr>
<tr>
<td>closing</td>
<td>Dispatched when the window is about to close. This only occurs automatically when the system chrome close button is pressed or, on Mac OS X, when the Quit command is invoked.</td>
</tr>
<tr>
<td>close</td>
<td>Dispatched when the window has closed.</td>
</tr>
</tbody>
</table>

The window events that dispatch an NativeWindowBoundsEvent object are:
For NativeWindowBoundsEvent events, you can use the beforeBounds and afterBounds properties to determine the window bounds before and after the impending or completed change.

The window events that dispatch a NativeWindowStateEvent object are:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>moving</td>
<td>Dispatched immediately before the top-left corner of the window changes position, either as a result of moving, resizing or changing the window display state.</td>
</tr>
<tr>
<td>move</td>
<td>Dispatched after the top-left corner has changed position.</td>
</tr>
<tr>
<td>resizing</td>
<td>Dispatched immediately before the window width or height changes either as a result of resizing or a display state change.</td>
</tr>
<tr>
<td>resize</td>
<td>Dispatched after the window has changed size.</td>
</tr>
</tbody>
</table>

For NativeWindowStateEvent events, you can use the beforeDisplayState and afterDisplayState properties to determine the window display state before and after the impending or completed change.

On some Linux window managers, a display state change event is not dispatched when a window with a maximum size setting is maximized. (The window is set to the maximized display state, but is not resized.)

### Displaying full-screen windows

Setting the displayState property of the Stage to StageDisplayState.FULL_SCREEN_INTERACTIVE places the window in full-screen mode, and keyboard input is permitted in this mode. (In SWF content running in a browser, keyboard input is not permitted). To exit full-screen mode, the user presses the Escape key.

**Note:** Some Linux window managers will not change the window dimensions to fill the screen if a maximum size is set for the window (but do remove the window system chrome).

The following HTML page simulates a full screen text terminal:
<html>
<head>
<title>Fullscreen Mode</title>
<script language="JavaScript" type="text/javascript">
function setDisplayState() {
    window.nativeWindow.stage.displayState =
    runtime.flash.display.StageDisplayState.FULL_SCREEN_INTERACTIVE;
}
</script>
<style type="text/css">
body, .mono {
    font-family: Courier New, Courier, monospace;
    font-size: x-large;
    color:#CCFF00;
    background-color:#003030;
}
</style>
</head>
<body onload="setDisplayState();">
    <p class="mono">Welcome to the dumb terminal app. Press the ESC key to exit...</p>
    <textarea name="dumb" class="mono" cols="100" rows="40"></textarea>
</body>
</html>
Chapter 18: Screens

Use the Adobe® AIR® Screen class to access information about the desktop display screens attached to a computer.

Additional online information about screens

You can find more information about the Screen class and working with screens from these sources:

Language Reference
- Screen

Adobe Developer Connection Articles and Samples
- Adobe AIR Developer Connection for HTML and Ajax (search for 'AIR screens')

Screen basics

The screen API contains a single class, Screen, which provides static members for getting system screen information, and instance members for describing a particular screen.

A computer system can have several monitors or displays attached, which can correspond to several desktop screens arranged in a virtual space. The AIR Screen class provides information about the screens, their relative arrangement, and their usable space. If more than one monitor maps to the same screen, only one screen exists. If the size of a screen is larger than the display area of the monitor, there is no way to determine which portion of the screen is currently visible.

A screen represents an independent desktop display area. Screens are described as rectangles within the virtual desktop. The top-left corner of screen designated as the primary display is the origin of the virtual desktop coordinate system. All values used to describe a screen are provided in pixels.
In this screen arrangement, two screens exist on the virtual desktop. The coordinates of the top-left corner of the main screen (1) are always (0,0). If the screen arrangement is changed to designate screen 2 as the main screen, then the coordinates of screen 1 become negative. Menubars, taskbars, and docks are excluded when reporting the usable bounds for a screen.

For detailed information about the screen API class, methods, properties, and events, see the Adobe AIR Language Reference for HTML Developers (http://www.adobe.com/go/learn_air_html_jsln).

### Enumerating the screens

You can enumerate the screens of the virtual desktop with the following screen methods and properties:

<table>
<thead>
<tr>
<th>Method or Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Screen.screens</td>
<td>Provides an array of Screen objects describing the available screens. Note that the order of the array is not significant.</td>
</tr>
<tr>
<td>Screen.mainScreen</td>
<td>Provides a Screen object for the main screen. On Mac OS X, the main screen is the screen displaying the menu bar. On Windows, the main screen is the system-designated primary screen.</td>
</tr>
<tr>
<td>Screen getScreensForRectangle()</td>
<td>Provides an array of Screen objects describing the screens intersected by the given rectangle. The rectangle passed to this method is in pixel coordinates on the virtual desktop. If no screens intersect the rectangle, then the array is empty. You can use this method to find out on which screens a window is displayed.</td>
</tr>
</tbody>
</table>

You should not save the values returned by the Screen class methods and properties. The user or operating system can change the available screens and their arrangement at any time.

The following example uses the screen API to move a window between multiple screens in response to pressing the arrow keys. To move the window to the next screen, the example gets the screens array and sorts it either vertically or horizontally (depending on the arrow key pressed). The code then walks through the sorted array, comparing each screen to the coordinates of the current screen. To identify the current screen of the window, the example calls Screen.getScreensForRectangle(), passing in the window bounds.
<html>
<head>
    <script src="AIRAliases.js" type="text/javascript"></script>
    <script type="text/javascript">
    function onKey(event){
        if(air.Screen.screens.length > 1){
            switch(event.keyCode){
                case air.Keyboard.LEFT :
                    moveLeft();
                    break;
                case air.Keyboard.RIGHT :
                    moveRight();
                    break;
                case air.Keyboard.UP :
                    moveUp();
                    break;
                case air.Keyboard.DOWN :
                    moveDown();
                    break;
            }
        }
    }

    function moveLeft(){
        var currentScreen = getCurrentScreen();
        var left = air.Screen.screens;
        left.sort(sortHorizontal);
        for(var i = 0; i < left.length - 1; i++){
            if(left[i].bounds.left < window.nativeWindow.bounds.left){
                window.nativeWindow.x += left[i].bounds.left - currentScreen.bounds.left;
                window.nativeWindow.y += left[i].bounds.top - currentScreen.bounds.top;
            }
        }
    }

    function moveRight(){
        var currentScreen = getCurrentScreen();
        var left = air.Screen.screens;
        left.sort(sortHorizontal);
        for(var i = left.length - 1; i > 0; i--){
            if(left[i].bounds.left > window.nativeWindow.bounds.left){
                window.nativeWindow.x += left[i].bounds.left - currentScreen.bounds.left;
                window.nativeWindow.y += left[i].bounds.top - currentScreen.bounds.top;
            }
        }
    }

    function moveUp(){
        var currentScreen = getCurrentScreen();
        var top = air.Screen.screens;
        top.sort(sortVertical);
        for(var i = 0; i < top.length - 1; i++){
            if(top[i].bounds.top < window.nativeWindow.bounds.top){
                window.nativeWindow.x += top[i].bounds.left - currentScreen.bounds.left;
                window.nativeWindow.y += top[i].bounds.top - currentScreen.bounds.top;
                break;
            }
        }
    }
    </script>
</head>
<body>
</body>
function moveDown()
{
    var currentScreen = getCurrentScreen();

    var top = air.Screen.screens;
    top.sort(sortVertical);
    for(var i = top.length - 1; i > 0; i--){
        if(top[i].bounds.top > window.nativeWindow.bounds.top){
            window.nativeWindow.x += top[i].bounds.left - currentScreen.bounds.left;
            window.nativeWindow.y += top[i].bounds.top - currentScreen.bounds.top;
            break;
        }
    }
}

function sortHorizontal(a,b){
    if (a.bounds.left > b.bounds.left){
        return 1;
    } else if (a.bounds.left < b.bounds.left){
        return -1;
    } else {return 0;}
}

function sortVertical(a,b){
    if (a.bounds.top > b.bounds.top){
        return 1;
    } else if (a.bounds.top < b.bounds.top){
        return -1;
    } else {return 0;}
}

function getCurrentScreen(){
    var current;
    var screens = air.Screen.getScreensForRectangle(window.nativeWindow.bounds);
    (screens.length > 0) ? current = screens[0] : current = air.Screen.mainScreen;
    return current;
}

function init(){
    window.nativeWindow.stage.addEventListener("keyDown",onKey);
}

<title>Screen Hopper</title>
</head>
<body onload="init()">
    <p>Use the arrow keys to move the window between monitors.</p>
</body>
</html>
Chapter 19: Working with native menus

Use the classes in the native menu API to define application, window, context, and pop-up menus.

Additional online information about native menus

You can find more information about the native menu API and working with native menus from these sources:

Quick Starts (Adobe AIR Developer Connection)
• Adding native menus to an AIR application

Language Reference
• NativeMenu
• NativeMenuItem

Adobe Developer Connection Articles and Samples
• Adobe AIR Developer Connection for HTML and Ajax (search for 'AIR menus')

AIR menu basics

The native menu classes allow you to access the native menu features of the operating system on which your application is running. NativeMenu objects can be used for application menus (available on Mac OS X), window menus (available on Windows and Linux), context menus, and pop-up menus.

AIR menu classes

The Adobe® AIR® Menu classes include:

<table>
<thead>
<tr>
<th>Package</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>flash.display</td>
<td>• NativeMenu</td>
</tr>
<tr>
<td></td>
<td>• NativeMenuItem</td>
</tr>
<tr>
<td>flash.ui</td>
<td>• ContextMenu</td>
</tr>
<tr>
<td></td>
<td>• ContextMenuItem</td>
</tr>
<tr>
<td>flash.events</td>
<td>• Event</td>
</tr>
</tbody>
</table>

Menu varieties

AIR supports the following types of menus:

Application menus An application menu is a global menu that applies to the entire application. Application menus are supported on Mac OS X, but not on Windows or Linux. On Mac OS X, the operating system automatically creates an
application menu. You can use the AIR menu API to add items and submenus to the standard menus. You can add listeners for handling the existing menu commands. Or you can remove existing items.

**Window menus** A window menu is associated with a single window and is displayed below the title bar. Menus can be added to a window by creating a NativeMenu object and assigning it to the `menu` property of the NativeWindow object. Window menus are supported on the Windows and Linux operating systems, but not on Mac OS X. Native window menus can only be used with windows that have system chrome.

**Context menus** Context menus open in response to a right-click or command-click on an interactive object in SWF content or a document element in HTML content. You can create a context menu using either the NativeMenu or the ContextMenu class. In HTML content, you can use the Webkit HTML and JavaScript APIs to add context menus to HTML elements.

**Dock and system tray icon menus** These icon menus are similar to context menus and are assigned to an application icon in the Mac OS X dock or the Windows and Linux notification areas on the taskbar. Dock and system tray icon menus use the NativeMenu class. On Mac OS X, the items in the menu are added above the standard operating system items. On Windows or Linux, there is no standard menu.

**Pop-up menus** An AIR pop-up menu is like a context menu, but is not necessarily associated with a particular application object or component. Pop-up menus can be displayed anywhere in a window by calling the `display()` method of any NativeMenu object.

**Custom menus** Native menus are drawn entirely by the operating system and, as such, exist outside the Flash and HTML rendering models. You are free to create your own non-native menus using MXML, ActionScript, or JavaScript. The AIR menu classes do not provide any facility for controlling the drawing of native menus.

**Default menus**
The following default menus are provided by the operating system or a built-in AIR class:

- Application menu on Mac OS X
- Dock icon menu on Mac OS X
- Context menu for selected text and images in HTML content
- Context menu for selected text in a TextField object (or an object that extends TextField)

**Menu structure**
Menus are hierarchical in nature. NativeMenu objects contain child NativeMenuItem objects. NativeMenuItem objects that represent submenus, in turn, can contain NativeMenu objects. The top- or root-level menu object in the structure represents the menu bar for application and window menus. (Context, icon, and pop-up menus don’t have a menu bar).
The following diagram illustrates the structure of a typical menu. The root menu represents the menu bar and contains two menu items referencing a *File* submenu and an *Edit* submenu. The *File* submenu in this structure contains two command items and an item that references an *Open Recent Menu* submenu, which, itself, contains three items. The *Edit* submenu contains three commands and a separator.

Defining a submenu requires both a `NativeMenu` and a `NativeMenuItem` object. The `NativeMenuItem` object defines the label displayed in the parent menu and allows the user to open the submenu. The `NativeMenu` object serves as a container for items in the submenu. The `NativeMenuItem` object references the `NativeMenu` object through the `submenu` property.

To view a code example that creates this menu see “Example: Window and application menu” on page 166.

**Menu events**

`NativeMenu` and `NativeMenuItem` objects both dispatch `displaying` and `select` events:

**Displaying:** Immediately before a menu is displayed, the menu and its menu items dispatch a `displaying` event to any registered listeners. The `displaying` event gives you an opportunity to update the menu contents or item appearance before it is shown to the user. For example, in the listener for the `displaying` event of an “Open Recent” menu, you could change the menu items to reflect the current list of recently viewed documents.

The `target` property of the event object is always the menu that is about to be displayed. The `currentTarget` is the object on which the listener is registered: either the menu itself, or one of its items.

**Note:** The `displaying` event is also dispatched whenever the state of the menu or one of its items is accessed.
Select: When a command item is chosen by the user, the item dispatches a `select` event to any registered listeners. Submenu and separator items cannot be selected and so never dispatch a `select` event.

A `select` event bubbles up from a menu item to its containing menu, on up to the root menu. You can listen for `select` events directly on an item and you can listen higher up in the menu structure. When you listen for the `select` event on a menu, you can identify the selected item using the event `target` property. As the event bubbles up through the menu hierarchy, the `currentTarget` property of the event object identifies the current menu object.

**Note:** `ContextMenu` and `ContextMenuItem` objects dispatch `menuItemSelect` and `menuSelect` events as well as `select` and `displaying` events.

### Key equivalents for menu commands

You can assign a key equivalent (sometimes called an accelerator) to a menu command. The menu item dispatches a `select` event to any registered listeners when the key, or key combination is pressed. The menu containing the item must be part of the menu of the application or the active window for the command to be invoked.

Key equivalents have two parts, a string representing the primary key and an array of modifier keys that must also be pressed. To assign the primary key, set the menu item `keyEquivalent` property to the single character string for that key. If you use an uppercase letter, the shift key is added to the modifier array automatically.

On Mac OS X, the default modifier is the command key (`Keyboard.COMMAND`). On Windows and Linux, it is the control key (`Keyboard.CONTROL`). These default keys are automatically added to the modifier array. To assign different modifier keys, assign a new array containing the desired key codes to the `keyEquivalentModifiers` property. The default array is overwritten. Whether you use the default modifiers or assign your own modifier array, the shift key is added if the string you assign to the `keyEquivalent` property is an uppercase letter. Constants for the key codes to use for the modifier keys are defined in the `Keyboard` class.

The assigned key equivalent string is automatically displayed beside the menu item name. The format depends on the user’s operating system and system preferences.

**Note:** If you assign the `Keyboard.COMMAND` value to a key modifier array on the Windows operating system, no key equivalent is displayed in the menu. However, the control key must be used to activate the menu command.

The following example assigns `Ctrl+Shift+G` as the key equivalent for a menu item:

```javascript
var item = new air.NativeMenuItem("Ungroup");
item.keyEquivalent = "G";
```

This example assigns `Ctrl+Shift+G` as the key equivalent by setting the modifier array directly:

```javascript
var item = new air.NativeMenuItem("Ungroup");
item.keyEquivalent = "G";
item.keyEquivalentModifiers = [air.Keyboard.CONTROL];
```

**Note:** Key equivalents are only triggered for application and window menus. If you add a key equivalent to a context or pop-up menu, the key equivalent is displayed in the menu label, but the associated menu command is never invoked.

### Mnemonics

Mnemonics are part of the operating system keyboard interface to menus. Linux, Mac OS X, and Windows allow users to open menus and select commands with the keyboard, but there are subtle differences.

On Mac OS X, the user types the first letter or two of the menu or command and then presses the return key. The `mnemonicIndex` property is ignored.
On Windows, only a single letter is significant. By default, the significant letter is the first character in the label, but if you assign a mnemonic to the menu item, then the significant character becomes the designated letter. If two items in a menu have the same significant character (whether or not a mnemonic has been assigned), then the user’s keyboard interaction with the menu changes slightly. Instead of pressing a single letter to select the menu or command, the user must press the letter as many times as necessary to highlight the desired item and then press the enter key to complete the selection. To maintain a consistent behavior, you should assign a unique mnemonic to each item in a menu for window menus.

On Linux, no default mnemonic is provided. You must specify a value for the `mnemonicIndex` property of a menu item to provide a mnemonic.

Specify the mnemonic character as an index into the label string. The index of the first character in a label is 0. Thus, to use “r” as the mnemonic for a menu item labeled, “Format,” you would set the `mnemonicIndex` property equal to 2.

```javascript
var item = new air.NativeMenuItem("Format");
item.mnemonicIndex = 2;
```

### Menu item state

Menu items have the two state properties, `checked` and `enabled`:

- **checked** Set to `true` to display a check mark next to the item label.

```javascript
var item = new air.NativeMenuItem("Format");
item.checked = true;
```

- **enabled** Toggle the value between `true` and `false` to control whether the command is enabled. Disabled items are visually “grayed-out” and do not dispatch `select` events.

```javascript
var item = new air.NativeMenuItem("Format");
item.enabled = false;
```

### Attaching an object to a menu item

The `data` property of the `NativeMenuItem` class allows you to reference an arbitrary object in each item. For example, in an “Open Recent” menu, you could assign the File object for each document to each menu item.

```javascript
var file = air.File.applicationStorageDirectory.resolvePath("GreatGatsby.pdf");
var menuItem = docMenu.addItem(new air.NativeMenuItem(file.name));
menuItem.data = file;
```

### Creating native menus

This topic describes how to create the various types of native menu supported by AIR.

#### Creating a root menu object

To create a `NativeMenu` object to serve as the root of the menu, use the `NativeMenu` constructor:

```javascript
var root = new air.NativeMenu();
```

For application and window menus, the root menu represents the menu bar and should only contain items that open submenus. Context menu and pop-up menus do not have a menu bar, so the root menu can contain commands and separator lines as well as submenus.
After the menu is created, you can add menu items. Items appear in the menu in the order in which they are added, unless you add the items at a specific index using the `addItemAt()` method of a menu object.

Assign the menu as an application, window, or icon menu, or display it as a pop-up menu, as shown in the following sections:

**Setting the application menu or window menu**

It's important that your code accommodate both application menus (supported on Mac OS) and window menus (supported on other operating systems)

```javascript
var root = new air.NativeMenu();
if (air.NativeApplication.supportsMenu)
{
    air.NativeApplication.nativeApplication.menu = root;
}
else if (NativeWindow.supportsMenu)
{
    nativeWindow.menu = root;
}
```

**Note:** Mac OS defines a menu containing standard items for every application. Assigning a new NativeMenu object to the `menu` property of the NativeApplication object replaces the standard menu. You can also use the standard menu instead of replacing it.

The Adobe Flex provides a FlexNativeMenu class for easily creating menus that work across platforms. If you are using the Flex Framework, use the FlexNativeMenu classes instead of the NativeMenu class. See About the FlexNativeMenu control.

For more information on good coding practices, see Developing cross-platform Adobe AIR applications.

**Setting a dock icon menu or system tray icon menu**

```javascript
air.NativeApplication.nativeApplication.icon.menu = root;
```

**Note:** Mac OS X defines a standard menu for the application dock icon. When you assign a new NativeMenu to the `menu` property of the DockIcon object, the items in that menu are displayed above the standard items. You cannot remove, access, or modify the standard menu items.

For more information on good coding practices, see Developing cross-platform Adobe AIR applications.

**Displaying a menu as a pop-up**

```javascript
root.display(window.nativeWindow.stage, x, y);
```

**Creating a submenu**

To create a submenu, you add a NativeMenuItem object to the parent menu and then assign the NativeMenu object defining the submenu to the item’s `submenu` property. AIR provides two ways to create submenu items and their associated menu object:

You can create a menu item and its related menu object in one step with the `addSubmenu()` method:

```javascript
var editMenuItem = root.addSubmenu(new air.NativeMenu(), "Edit");
```

You can also create the menu item and assign the menu object to its `submenu` property separately:

```javascript
var editMenuItem = root.addItem("Edit", false);
editMenuItem.submenu = new air.NativeMenu();
```
Creating a menu command

To create a menu command, add a NativeMenuItem object to a menu and add an event listener referencing the function implementing the menu command:

```javascript
var copy = new air.NativeMenuItem("Copy", false);
copy.addEventListener(air.Event.SELECT, onCopyCommand);
editMenu.addItem(copy);
```

You can listen for the `select` event on the command item itself (as shown in the example), or you can listen for the `select` event on a parent menu object.

*Note: Menu items that represent submenus and separator lines do not dispatch select events and so cannot be used as commands.*

Creating a menu separator line

To create a separator line, create a NativeMenuItem, setting the `isSeparator` parameter to `true` in the constructor. Then add the separator item to the menu in the correct location:

```javascript
var separatorA = new air.NativeMenuItem("A", true);
editMenu.addItem(separatorA);
```

The label specified for the separator, if any, is not displayed.

About context menus in HTML

In HTML content, the `contextmenu` event can be used to display a context menu. By default, a context menu is displayed automatically when the user invokes the context menu event on selected text (by right-clicking or command-clicking the text). To prevent the default menu from opening, listen for the `contextmenu` event and call the event object’s `preventDefault()` method:

```javascript
function showContextMenu(event){
    event.preventDefault();
}
```

You can then display a custom context menu using DHTML techniques or by displaying an AIR native context menu. The following example displays a native context menu by calling the menu `display()` method in response to the HTML `contextmenu` event:
<html>
<head>
<script src="AIRAliases.js" language="JavaScript" type="text/javascript"></script>
<script language="javascript" type="text/javascript">
function showContextMenu(event){
    event.preventDefault();
    contextMenu.display(window.nativeWindow.stage, event.clientX, event.clientY);
}

function createContextMenu(){
    var menu = new air.NativeMenu();
    var command = menu.addItem(new air.NativeMenuItem("Custom command"));
    command.addEventListener(air.Event.SELECT, onCommand);
    return menu;
}

function onCommand(){
    air.trace("Context command invoked.");
}

var contextMenu = createContextMenu();
</script>
</head>
<body>
<p oncontextmenu="showContextMenu(event)" style="-khtml-user-select:auto;">Custom context menu.</p>
</body>
</html>

### Displaying pop-up menus

You can display any NativeMenu object at an arbitrary time and location above a window, by calling the menu display() method. The method requires a reference to the stage; thus, only content in the application sandbox can display a menu as a pop-up.

The following method displays the menu defined by a NativeMenu object named popupMenu in response to a mouse click:

```javascript
function onMouseClick(event) {
    popupMenu.display(window.nativeWindow.stage, event.clientX, event.clientY);
}
```

*Note: The menu does not need to be displayed in direct response to an event. Any method can call the display() function.*

### Handling menu events

A menu dispatches events when the user selects the menu or when the user selects a menu item.
Events summary for menu classes

Add event listeners to menus or individual items to handle menu events.

<table>
<thead>
<tr>
<th>Object</th>
<th>Events dispatched</th>
</tr>
</thead>
<tbody>
<tr>
<td>NativeMenu</td>
<td>NativeMenuEvent.DISPLAYING</td>
</tr>
<tr>
<td></td>
<td>NativeMenuEvent.SELECT (propagated from child items and submenus)</td>
</tr>
<tr>
<td>NativeMenuItem</td>
<td>NativeMenuEvent.SELECT</td>
</tr>
<tr>
<td></td>
<td>NativeMenuEvent.DISPLAYING (propagated from parent menu)</td>
</tr>
</tbody>
</table>

Select menu events

To handle a click on a menu item, add an event listener for the `select` event to the `NativeMenuItem` object:

```javascript
var menuCommandX = new NativeMenuItem("Command X");
menuCommand.addEventListener(air.Event.SELECT, doCommandX);
```

Because `select` events bubble up to the containing menus, you can also listen for `select` events on a parent menu. When listening at the menu level, you can use the event object `target` property to determine which menu command was selected. The following example traces the label of the selected command:

```javascript
var colorMenuItem = new air.NativeMenuItem("Choose a color");
var colorMenu = new air.NativeMenu();
colorMenuItem.submenu = colorMenu;

var red = new air.NativeMenuItem("Red");
var green = new air.NativeMenuItem("Green");
var blue = new air.NativeMenuItem("Blue");
colorMenu.addItem(red);
colorMenu.addItem(green);
colorMenu.addItem(blue);

if(air.NativeApplication.supportsMenu){
    air.NativeApplication.nativeApplication.menu.addItem(colorMenuItem);
    air.NativeApplication.nativeApplication.menu.addEventListener(air.Event.SELECT, colorChoice);
} else if (air.NativeWindow.supportsMenu){
    var windowMenu = new air.NativeMenu();
    window.nativeWindow.menu = windowMenu;
    windowMenu.addItem(colorMenuItem);
    windowMenu.addEventListener(air.Event.SELECT, colorChoice);
}

function colorChoice(event) {
    var menuItem = event.target;
    air.trace(menuItem.label + " has been selected");
}
```

If you are using the `ContextMenuItem` class, you can listen for either the `select` event or the `menuItemSelect` event. The `menuItemSelect` event gives you additional information about the object owning the context menu, but does not bubble up to the containing menus.
Displaying menu events

To handle the opening of a menu, you can add a listener for the `displaying` event, which is dispatched before a menu is displayed. You can use the `displaying` event to update the menu, for example by adding or removing items, or by updating the enabled or checked states of individual items.

Example: Window and application menu

The following example creates the menu shown in “Menu structure” on page 158.

The menu is designed to work both on Windows, for which only window menus are supported, and on Mac OS X, for which only application menus are supported. To make the distinction, the `MenuExample` class constructor checks the static `supportsMenu` properties of the `NativeWindow` and `NativeApplication` classes. If `NativeWindow.supportsMenu` is true, then the constructor creates a `NativeMenu` object for the window and then creates and adds the File and Edit submenus. If `NativeApplication.supportsMenu` is true, then the constructor creates and adds the File and Edit menus to the existing menu provided by the Mac OS X operating system.

The example also illustrates menu event handling. The `select` event is handled at the item level and also at the menu level. Each menu in the chain from the menu containing the selected item to the root menu responds to the `select` event. The `displaying` event is used with the “Open Recent” menu. Just before the menu is opened, the items in the menu are refreshed from the recent Documents array (which doesn’t actually change in this example). Although not shown in this example, you can also listen for `displaying` events on individual items.

```html
<html>
<head>
<script src="AIRAliases.js" type="text/javascript"></script>
</head>
<body>

var application = air.NativeApplication.nativeApplication;
var recentDocuments =
    new Array(new air.File("app-storage:/GreatGatsby.pdf"),
              new air.File("app-storage:/WarAndPeace.pdf"),
              new air.File("app-storage:/Iliad.pdf"));

function MenuExample(){
    var fileMenu;
    var editMenu;

    if (air.NativeWindow.supportsMenu &&
        nativeWindow.systemChrome != air.NativeWindowSystemChrome.NONE) {
        nativeWindow.menu = new air.NativeMenu();
        nativeWindow.menu.addEventListener(air.Event.SELECT, selectCommandMenu);
        fileMenu = nativeWindow.menu.addItem(new air.NativeMenuItem("File"));
        fileMenu.submenu = createFileMenu();
        editMenu = nativeWindow.menu.addItem(new air.NativeMenuItem("Edit"));
        editMenu.submenu = createEditMenu();
    }

    if (air.NativeApplication.supportsMenu) {
        application.menu.addEventListener(air.Event.SELECT, selectCommandMenu);
        fileMenu = application.menu.addItem(new air.NativeMenuItem("File"));
        fileMenu.submenu = createFileMenu();
        editMenu = application.menu.addItem(new air.NativeMenuItem("Edit"));
        editMenu.submenu = createEditMenu();
    }

```
function createFileMenu() {
    var fileMenu = new air.NativeMenu();
    fileMenu.addEventListener(air.Event.SELECT, selectCommandMenu);

    var newCommand = fileMenu.addItem(new air.NativeMenuItem("New"));
    newCommand.addEventListener(air.Event.SELECT, selectCommand);

    var saveCommand = fileMenu.addItem(new air.NativeMenuItem("Save"));
    saveCommand.addEventListener(air.Event.SELECT, selectCommand);

    var openFile = fileMenu.addItem(new air.NativeMenuItem("Open Recent"));
    openFile.submenu = new air.NativeMenu();
    openFile.submenu.addEventListener(air.Event.DISPLAYING, updateRecentDocumentMenu);
    openFile.submenu.addEventListener(air.Event.SELECT, selectCommandMenu);

    return fileMenu;
}

function createEditMenu() {
    var editMenu = new air.NativeMenu();
    editMenu.addEventListener(air.Event.SELECT, selectCommandMenu);

    var copyCommand = editMenu.addItem(new air.NativeMenuItem("Copy"));
    copyCommand.addEventListener(air.Event.SELECT, selectCommand);
    copyCommand.keyEquivalent = "c";

    var pasteCommand = editMenu.addItem(new air.NativeMenuItem("Paste"));
    pasteCommand.addEventListener(air.Event.SELECT, selectCommand);
    pasteCommand.keyEquivalent = "v";

    editMenu.addItem(new air.NativeMenuItem("", true));

    var preferencesCommand = editMenu.addItem(new air.NativeMenuItem("Preferences"));
    preferencesCommand.addEventListener(air.Event.SELECT, selectCommand);

    return editMenu;
}

function updateRecentDocumentMenu(event) {
    air.trace("Updating recent document menu.");
    var docMenu = air.NativeMenu(event.target);

    for (var i = docMenu.numItems - 1; i >= 0; i--) {
        docMenu.removeItemAt(i);
    }

    for (var file in recentDocuments) {
        var menuItem =
            docMenu.addItem(new air.NativeMenuItem(recentDocuments[file].name));
        menuItem.data = recentDocuments[file];
        menuItem.addEventListener(air.Event.SELECT, selectRecentDocument);
    }
}

function selectRecentDocument(event) {
    air.trace("Selected recent document: " + event.target.data.name);
}

function selectCommand(event) {
Using the MenuBuilder framework

In addition to the standard menu classes, Adobe AIR includes a menu builder JavaScript framework to make it easier for developers to create menus. The MenuBuilder framework allows you to define the structure of your menus declaratively in XML or JSON format. It also provides helper methods for creating any of the menu types available to an AIR application. For a complete list of the ways a native menu can be used in AIR, see “AIR menu basics” on page 157.

Creating a menu with the MenuBuilder framework

The MenuBuilder framework allows you to define the structure of a menu using XML or JSON. The framework includes methods for loading and parsing the file containing the menu structure. Once a menu structure is loaded, additional methods allow you to designate how the menu is used in the application. The methods allow you to set the menu as the Mac OS X application menu, as a window menu, or as a context menu.

The MenuBuilder framework is not built in to the runtime. To use the framework, include the AIRMenuBuilder.js file (included with the Adobe AIR SDK) in your application code, as shown here:
The MenuBuilder framework is designed to run in the application sandbox. The framework methods can’t be called from the classic sandbox.

All the framework methods that are for developer use are defined as class methods on the air.ui.Menu class.

**MenuBuilder basic workflow**

In general, regardless of the type of menu you want to create, you follow three steps to create a menu with the MenuBuilder framework:

1. **Define the menu structure**: Create a file containing XML or JSON that defines the menu structure. For some menu types, the top-level menu items are menus (for example in a window or application menu). For other menu types, the top-level items are individual menu commands (such as in a context menu). For details on the format for defining menu structure, see “Defining MenuBuilder menu structure” on page 171.

2. **Load the menu structure**: Call the appropriate Menu class method, either `Menu.createFromXML()` or `Menu.createFromJSON()`, to load the menu structure file and parse it into an actual menu object. Either method returns a NativeMenu object that can be passed to one of the framework’s menu-setting methods.

3. **Assign the menu**: Call the appropriate Menu class method according to how the menu is used. The options are:
   - `Menu.setAsMenu()` for a window or application menu
   - `Menu.setAsContextMenu()` to display the menu as a context menu for a DOM element
   - `Menu.setAsIconMenu()` to set the menu as the context menu for a system tray or dock icon

The timing of when the code executes can be important. In particular, a window menu must be assigned before the actual operating system window is created. Any `setAsMenu()` call that sets a menu as a window menu must execute directly in the HTML page rather than in the `onload` or other event handler. The code to create the menu must run before the operating system opens the window. At the same time, any `setAsContextMenu()` call that refers to a DOM elements must occur after the DOM element is created. The safest approach is to place the `<script>` block containing the menu assignment code just inside the closing `</body>` tag at the end of the HTML page.

**Loading menu structure**

Regardless of the intended use of your menu, you define the structure of the menu as a separate file containing an XML or JSON structure. Before you can assign a menu in your application, first use the framework to load and parse the menu structure file. To load and parse a menu structure file, use one of these two framework methods:

- `Menu.createFromXML()` to load and parse an XML-formatted menu structure file
- `Menu.createFromJSON()` to load and parse a JSON-formatted menu structure file

Both methods accept one argument: the file path of the menu structure file. Both methods load the file from that location. They parse the file contents and return a NativeMenu object with the menu structure defined in the file. For example, the following code loads a menu structure file named “windowMenu.xml” that’s in the same directory as the HTML file that’s loading it:

```javascript
var windowMenu = air.ui.Menu.createFromXML("windowMenu.xml");
```

In the next example, the code loads a menu structure file named “contextMenu.js” from a directory named “menus”:

```javascript
var contextMenu = air.ui.Menu.createFromJSON("menus/contextMenu.js");
```

**Note**: The generated NativeMenu object can only be used once as an application or window menu. However, a generated NativeMenu object can be used multiple times in an application as a context or icon menu. Using the MenuBuilder framework on Mac OS X, if the same NativeMenu is assigned as the application menu and also as another type of menu, it is only used as the application menu.
For details of the specific menu structure that the MenuBuilder framework accepts, see "Defining MenuBuilder menu structure" on page 171.

Creating an application or window menu
When you create an application or window menu using the MenuBuilder framework, the top-level objects or nodes in the menu data structure correspond to the items that show up in the menu bar. Items nested inside one of those top-level items define the individual menu commands. Likewise, those menu items can contain other items. In that case the menu item is a submenu rather than a command. When the user selects the menu item it expands its own menu of items.

You use the Menu.setAsMenu() method to set a menu as the application menu or window menu for the window in which the call executes. The setAsMenu() method takes one parameter: the NativeMenu object to use. The following example loads an XML file and sets the generated menu as the application or window menu:

```javascript
var windowMenu = air.ui.Menu.createFromXML("windowMenu.xml");
air.ui.Menu.setAsMenu(windowMenu);
```

On an operating system that supports window menus, the setAsMenu() call sets the menu as the window menu for the current window (the window that’s represented as window.nativeWindow). On an operating system that supports an application menu, the menu is used as the application menu.

Mac OS X defines a set of standard menus as the default application menu, with the same set of menu items for every application. These menus include an application menu whose name matches the application name, an Edit menu, and a Window menu. When you assign a NativeMenu object as the application menu by calling the Menu.setAsMenu() method, the items in the NativeMenu are inserted into the standard menu structure between the Edit and Window menus. The standard menus are not modified or replaced.

You can replace the standard menus rather than supplement them if you prefer. To replace the existing menu, pass a second argument with the value true to the setAsMenu() call, as in this example:

```javascript
air.ui.Menu.setAsMenu(windowMenu, true);
```

Creating a DOM element context menu
Creating a context menu for a DOM element using the MenuBuilder framework involves two steps. First you create the NativeMenu instance that defines the menu structure using the Menu.createFromXML() or Menu.createFromJSON() method. You then assign that menu as the context menu for a DOM element by calling the Menu.setAsContextMenu() method. Because a context menu consists of a single menu, the top-level menu items in the menu data structure serve as the items in the single menu. Any menu item that contains child menu items defines a submenu. To assign a NativeMenu as the context menu for a DOM element, call the Menu.setAsContextMenu() method. This method requires two parameters: the NativeMenu to set as the context menu, and the id (a string) of the DOM element to which it is assigned:

```javascript
var treeContextMenu = air.ui.Menu.createFromXML("treeContextMenu.xml");
air.ui.Menu.setAsContextMenu(treeContextMenu, "navTree");
```

If you omit the DOM element parameter, the method uses the HTML document from which the method is called as the default value. In other words, the menu is set as the context menu for the HTML document’s entire window. This technique is convenient for removing the default context menu from an entire HTML window by passing null for the first parameter, as in this example:

```javascript
air.ui.Menu.setAsContextMenu(null);
```

You can also remove an assigned context menu from any DOM element. Call the setAsContextMenu() method and pass null and the element id as the two arguments.
Creating an icon context menu

In addition to context menus for DOM elements within an application window, an Adobe AIR application supports two other special context menus: dock icon menus for operating systems that support a dock, and system tray icon menus for operating systems that use a system tray. To set either of these menus, you first create a NativeMenu using the `Menu.createFromXML()` or `Menu.createFromJSON()` method. Then you assign the NativeMenu as the dock or system tray icon menu by calling the `Menu.setAsIconMenu()` method.

This method accepts two arguments. The first argument, which is required, is the NativeMenu to use as the icon menu. The second argument is an Array containing strings that are file paths to images to use as the icon, or BitmapData objects containing image data for the icon. This argument is required unless default icons are specified in the application.xml file. If default icons are specified in the application.xml file, those icons are used by default for the system tray icon.

The following example demonstrates loading menu data and assigning the menu as the dock or system tray icon context menu:

```javascript
// Assumes that icons are specified in the application.xml file. // Otherwise the icons would need to be specified using a second // parameter to the setAsIconMenu() function.
var iconMenu = air.ui.Menu.createFromXML("iconMenu.xml");
air.ui.Menu.setAsIconMenu(iconMenu);

Note: Mac OS X defines a standard context menu for the application dock icon. When you assign a menu as the dock icon context menu, the items in the menu are displayed above the standard OS menu items. You cannot remove, access, or modify the standard menu items.

Defining MenuBuilder menu structure

When you create a NativeMenu object using the `Menu.createFromXML()` or `Menu.createFromJSON()` method, the structure of XML elements or objects defines the structure of the resulting menu. Once the menu is created, you can change its structure or properties at run time. To change a menu item at run time you access the NativeMenuItem object by navigating through the NativeMenu object’s hierarchy.

The MenuBuilder framework looks for certain XML attributes or object properties as it parses through the menu data source. The presence and value of those attributes or properties determines the structure of the menu that’s created.

When you use XML for the menu structure, the XML file must contain a root node. The child nodes of the root node are used as the top-level menu item nodes. The XML nodes can have any name. The names of the XML nodes don’t affect the menu structure. Only the hierarchical structure of the nodes and their attribute values are used to define the menu.

Menu item types

Each entry in the menu data source (each XML element or JSON object) can specify an item type and type-specific information about the menu item it represents. Adobe AIR supports the following menu item types, which can be set as the values of the `type` attribute or property in the data source:
A normal menu item is treated as a submenu if it has children. With an XML data source, this means that the menu item element contains other XML elements. For a JSON data source, give the object representing the menu item a property named `items` containing an array of other objects.

**Menu data source attributes or properties**

Items in the menu data source can specify several XML attributes or object properties that determine how the item is displayed and behaves. The following table lists the attributes you can specify, their data types, their purposes, and how the data source must represent them:

<table>
<thead>
<tr>
<th>Attribute or property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>altKey</td>
<td>Boolean</td>
<td>Specifies whether the Alt key is required as part of the key equivalent for the item.</td>
</tr>
<tr>
<td>cmdKey</td>
<td>Boolean</td>
<td>Specifies whether the Command key is required as part of the key equivalent for the item. The <code>defaultKeyEquivalentModifiers</code> field also affects this value.</td>
</tr>
<tr>
<td>ctrlKey</td>
<td>Boolean</td>
<td>Specifies whether the Control key is required as part of the key equivalent for the item. The <code>defaultKeyEquivalentModifiers</code> field also affects this value.</td>
</tr>
<tr>
<td>defaultKeyEquivalentModifiers</td>
<td>Boolean</td>
<td>Specifies whether the operating system default modifier key (Command for Mac OS X and Control for Windows) is required as part of the key equivalent for the item. If not specified, the MenuBuilder framework treats the item as if the value was <code>true</code>.</td>
</tr>
<tr>
<td>enabled</td>
<td>Boolean</td>
<td>Specifies whether the user can select the menu item (true), or not (false). If not specified, the MenuBuilder framework treats the item as if the value was <code>true</code>.</td>
</tr>
<tr>
<td>items</td>
<td>Array</td>
<td>(JSON only) specifies that the menu item is itself a menu. The objects in the array are the child menu items contained in the menu.</td>
</tr>
</tbody>
</table>
The MenuBuilder framework ignores all other object properties or XML attributes.

## Example: An XML MenuBuilder data source

The following example uses the MenuBuilder framework to define a context menu for a region of text. It shows how to define the menu structure using XML as the data source. For an application that specifies an identical menu structure using a JSON array, see “Example: A JSON MenuBuilder data source” on page 174.

The application consists of two files.

The first file is the menu data source, in a file named “textContextMenu.xml.” While this example uses menu item nodes named “menuitem,” the actual name of the XML nodes doesn’t matter. As described previously, only the structure of the XML and the attribute values affect the structure of the generated menu.
<?xml version="1.0" encoding="utf-8" ?>
<root>
    <menuitem label="MenuItem A"/>
    <menuitem label="MenuItem B" type="check" toggled="true"/>
    <menuitem label="MenuItem C" enabled="false"/>
    <menuitem type="separator"/>
    <menuitem label="MenuItem D">
        <menuitem label="SubMenuItem D-1"/>
        <menuitem label="SubMenuItem D-2"/>
        <menuitem label="SubMenuItem D-3"/>
    </menuitem>
</root>

The second file is the source code for the application user interface (the HTML file specified as the initial window in the application.xml file:

<html>
    <head>
        <title>XML-based menu data source example</title>
        <script type="text/javascript" src="AIRAliases.js"></script>
        <script type="text/javascript" src="AIRMenuBuilder.js"></script>
        <style type="text/css">
            #contextEnabledText {
                margin-left: auto;
                margin-right: auto;
                margin-top: 100px;
                width: 50%
            }
        </style>
    </head>
    <body>
        <div id="contextEnabledText">This block of text is context menu enabled. Right click or Command-click on the text to view the context menu.</div>
        <script type="text/javascript">
            // Create a NativeMenu from "textContextMenu.xml" and set it as context menu for the "contextEnabledText" DOM element:
            var textMenu = air.ui.Menu.createFromXML("textContextMenu.xml");
            air.ui.Menu.setAsContextMenu(textMenu, "contextEnabledText");

            // Remove the default context menu from the page:
            air.ui.Menu.setAsContextMenu(null);
        </script>
    </body>
</html>

Example: A JSON MenuBuilder data source
The following example uses the MenuBuilder framework to define a context menu for a region of text using a JSON array as the data source. For an application that specifies an identical menu structure in XML, see “Example: An XML MenuBuilder data source” on page 173.

The application consists of two files.
The first file is the menu data source, in a file named “textContextMenu.js.”
The second file is the source code for the application user interface (the HTML file specified as the initial window in the application.xml file:

```html
<html>
<head>
<title>JSON-based menu data source example</title>
<script type="text/javascript" src="AIRAliases.js"></script>
<script type="text/javascript" src="AIRMenuBuilder.js"></script>
<style type="text/css">
#contextEnabledText
{
  margin-left: auto;
  margin-right: auto;
  margin-top: 100px;
  width: 50%
}
</style>
</head>
<body>
<div id="contextEnabledText">This block of text is context menu enabled. Right click or Command-click on the text to view the context menu.</div>
<script type="text/javascript">
  // Create a NativeMenu from "textContextMenu.js" and set it as context menu for the "contextEnabledText" DOM element:
  var textMenu = air.ui.Menu.createFromJSON("textContextMenu.js");
  air.ui.Menu.setAsContextMenu(textMenu, "contextEnabledText");

  // Remove the default context menu from the page:
  air.ui.Menu.setAsContextMenu(null);
</script>
</body>
</html>

Adding menu keyboard features with MenuBuilder

Operating system native menus support the use of keyboard shortcuts, and these shortcuts are also available in Adobe AIR. Two of the types of keyboard shortcuts that can be specified in a menu data source are keyboard equivalents for menu commands and mnemonics.
Specifying menu keyboard equivalents

You can specify a key equivalent (sometimes called an accelerator) for a window or application menu command. When the key or key combination is pressed the NativeMenuItem dispatches a select event and any onSelect event handler specified in the data source is called. The behavior is the same as though the user had selected the menu item.

For complete details about menu keyboard equivalents, see “Key equivalents for menu commands” on page 160.

Using the MenuBuilder framework, you can specify a keyboard equivalent for a menu item in its corresponding node in the data source. If the data source has a keyEquivalent field, the MenuBuilder framework uses that value as the key equivalent character.

You can also specify modifier keys that are part of the key equivalent combination. To add a modifier, specify true for the altKey, ctrlKey, cmdKey, or shiftKey field. The specified key or keys become part of the key equivalent combination. By default the Control key is specified for Windows and the Command key is specified for Mac OS X. To override this default behavior, include a defaultKeyEquivalentModifiers field set to false.

The following example shows the data structure for an XML-based menu data source that includes keyboard equivalents, in a file named “keyEquivalentMenu.xml”:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<root>
  <menuitem label="File">
    <menuitem label="New" keyEquivalent="n"/>
    <menuitem label="Open" keyEquivalent="o"/>
    <menuitem label="Save" keyEquivalent="s"/>
    <menuitem label="Save As..." keyEquivalent="s" shiftKey="true"/>
    <menuitem label="Close" keyEquivalent="w"/>
  </menuitem>
  <menuitem label="Edit">
    <menuitem label="Cut" keyEquivalent="x"/>
    <menuitem label="Copy" keyEquivalent="c"/>
    <menuitem label="Paste" keyEquivalent="v"/>
  </menuitem>
</root>
```

The following example application loads the menu structure from "keyEquivalentMenu.xml" and uses it as the structure for the window or application menu for the application:

```html
<html>
<head>
  <title>XML-based menu with key equivalents example</title>
  <script type="text/javascript" src="AIRAliases.js"></script>
  <script type="text/javascript" src="AIRMenuBuilder.js"></script>
</head>
<body>
  <script type="text/javascript">
    // Create a NativeMenu from "keyEquivalentMenu.xml" and set it
    // as the application/window menu
    var keyEquivMenu = air.ui.Menu.createFromXML("keyEquivalentMenu.xml");
    air.ui.Menu.setAsMenu(keyEquivMenu);
  </script>
</body>
</html>
```
Specifying menu item mnemonics

A menu item mnemonic is a key associated with a menu item. When the key is pressed while the menu is displayed, the menu item command is triggered. The behavior is the same as if the user had selected the menu item with the mouse. Typically the operating system indicates a menu item mnemonic by underlining that character in the name of the menu item.

For more information about mnemonics, see “Mnemonics” on page 160.

With the MenuBuilder framework, the simplest way to specify a mnemonic for a menu item is to include an underscore character (“_”) in the menu item’s label field. Place the underscore immediately to the left of the letter that serves as the mnemonic for that menu item. For example, if the following XML node is used in a data source that’s loaded using the MenuBuilder framework, the mnemonic for the command is the first character of the second word (the letter “A”):

```xml
<menuitem label="Save _As"/>
```

When the NativeMenu object is created, the underscore is not included in the label. Instead, the character following the underscore becomes the mnemonic for the menu item. To include a literal underscore character in a menu item’s name, use two underscore characters (“__”). This sequence is converted to an underscore in the menu item label.

As an alternative to using an underscore character in the label field, you can provide an integer index position for the mnemonic character. Specify the index in the mnemonicIndex field in the menu item data source object or XML element.

Handling MenuBuilder menu events

User interaction with a NativeMenu is event-driven. When the user selects a menu item or opens a menu or submenu, the NativeMenuItem object dispatches an event. With a NativeMenu object created using the MenuBuilder framework, you can register event listeners with individual NativeMenuItem objects or with the NativeMenu. You subscribe and respond to these events the same way as if you had created the NativeMenu and NativeMenuItem objects manually rather than using the MenuBuilder framework. For more information see “Menu events” on page 159.

The MenuBuilder framework supplements the standard event handling, providing a way to specify a select event handler function for a menu item within the menu data source. If you specify an onSelect field in the menu item data source, the specified function is called when the user selects the menu item. For example, suppose the following XML node is included in a data source that’s loaded using the MenuBuilder framework. When the menu item is selected the function named doSave() is called:

```xml
<menuitem label="Save" onSelect="doSave"/>
```

The onSelect field is a String when it’s used with an XML data source. With a JSON array, the field can be a String with the name of the function. In addition, for a JSON array only, the field can also be a variable reference to the function as an object. However, if the JSON array uses a Function variable reference the menu must be created before or during the onload event handler or a JavaScript security violation occurs. In all cases, the specified function must be defined in the global scope.

When the specified function is called, the runtime passes two arguments to it. The first argument is the event object dispatched by the select event. It is an instance of the Event class. The second argument that’s passed to the function is an anonymous object containing the data that was used to create the menu item. This object has the following properties. Each property’s value matches the value in the original data structure or null if the property is not set in the original data structure:

- altKey
- cmdKey
The following example lets you experiment with NativeMenu events. The example includes two menus. The window and application menu is created using an XML data source. The context menu for the list of items represented by the `<ul>` and `<li>` elements is created using a JSON array data source. A text area on the screen displays information about each event as the user selects menu items.

The following listing is the source code of the application:

```html
<html>
<head>
<title>Menu event handling example</title>
<script type="text/javascript" src="AIRAliases.js"></script>
<script type="text/javascript" src="AIRMenuBuilder.js"></script>
<script type="text/javascript" src="printObject.js"></script>
<script type="text/javascript">
    function fileMenuCommand(event, data) {
        print("fileMenuCommand", event, data);
    }

    function editMenuCommand(event, data) {
        print("editMenuCommand", event, data);
    }

    function moveItemUp(event, data) {
        print("moveItemUp", event, data);
    }

    function moveItemDown(event, data) {
        print("moveItemDown", event, data);
    }

    function print(command, event, data) {
        var result = "";
        result += "<h1>Command: " + command + '</h1>';
        result += "<p>" + printObject(event) + "</p>";
        result += "<p>Data: </p>";
        result += "<ul>";
        for (var s in data) {
            result += "<li>" + s + ": " + printObject(data[s]) + "</li>";
        }
        result += "</ul>";
    }
</script>
</head>
<body>
</body>
</html>
```
var o = document.getElementById("output");
o.innerHTML = result;
}
</script>
<style type="text/css">
#contextList {
    position: absolute; left: 0; top: 25px; bottom: 0; width: 100px;
    background: #eeeeee;
}
#output {
    position: absolute; left: 125px; top: 25px; right: 0; bottom: 0;
}
</style>
</head>
<body>
<div id="contextList">
<ul>
<li>List item 1</li>
<li>List item 2</li>
<li>List item 3</li>
</ul>
</div>
<div id="output">
Choose menu commands. Information about the events displays here.
</div>
<script type="text/javascript">
var mainMenu = air.ui.Menu.createFromXML("mainMenu.xml");
air.ui.Menu.setAsMenu(mainMenu);
var listContextMenu = air.ui.Menu.createFromJSON("listContextMenu.js");
air.ui.Menu.setAsContextMenu(listContextMenu, "contextList")
// clear the default context menu
air.ui.Menu.setAsContextMenu(null);
</script>
</body>
</html>

The following listing is the data source for the main menu ("mainMenu.xml"):

<?xml version="1.0" encoding="utf-8" ?>
<root>
    <menuitem label="File">
        <menuitem label="New" keyEquivalent="n" onSelect="fileMenuCommand"/>
        <menuitem label="Open" keyEquivalent="o" onSelect="fileMenuCommand"/>
        <menuitem label="Save" keyEquivalent="s" onSelect="fileMenuCommand"/>
        <menuitem label="Save As..." keyEquivalent="S" onSelect="fileMenuCommand"/>
        <menuitem label="Close" keyEquivalent="w" onSelect="fileMenuCommand"/>
    </menuitem>
    <menuitem label="Edit">
        <menuitem label="Cut" keyEquivalent="x" onSelect="editMenuCommand"/>
        <menuitem label="Copy" keyEquivalent="c" onSelect="editMenuCommand"/>
        <menuitem label="Paste" keyEquivalent="v" onSelect="editMenuCommand"/>
    </menuitem>
</root>

The following listing is the data source for the context menu ("listContextMenu.js");
The following listing contains the code from the printObject.js file. The file includes the printObject() function, which the application uses but which doesn’t affect the operation of the menus in the example.

```javascript
function printObject(obj) {
    if (obj) {
        if (typeof obj == "undefined") { return "[undefined]"; }
        if (typeof obj == "object") { return "[null]"; }
        return "[false]";
    } else {
        if (typeof obj == "boolean") { return "[true]"; }
        if (typeof obj == "object") {
            if (typeof obj.length == "number") {
                var ret = [];
                for (var i=0; i<obj.length; i++) {
                    ret.push(printObject(obj[i]));
                }
                return ["[", ret.join(", "), "]"].join( " ");
            } else {
                var ret = [];
                var hadChildren = false;
                for (var k in obj) {
                    hadChildren = true;
                    ret.push ([k, " => ", printObject(obj[k])]);
                }
                if (hadChildren) {
                    return ["{
                    
                
                
                    
                
            }
        return String(obj);
    }
```
Chapter 20: Taskbar icons

Many operating systems provide a taskbar, such as the Mac OS X dock, that can contain an icon to represent an application. Adobe® AIR® provides an interface for interacting with the application task bar icon through the `NativeApplication.nativeApplication.icon` property.

Additional online information about taskbar icons

You can find more information about working with taskbars from these sources:

Quick Starts (Adobe AIR Developer Connection)

Language Reference
- DockIcon
- SystemTrayIcon

Adobe Developer Connection Articles and Samples
- Adobe AIR Developer Connection for HTML and Ajax (search for 'AIR taskbar icons')

About taskbar icons

AIR creates the `NativeApplication.nativeApplication.icon` object automatically. The object type is either DockIcon or SystemTrayIcon, depending on the operating system. You can determine which of these InteractiveIcon subclasses that AIR supports on the current operating system using the `NativeApplication.supportsDockIcon` and `NativeApplication.supportsSystemTrayIcon` properties. The InteractiveIcon base class provides the properties `width`, `height`, and `bitmaps`, which you can use to change the image used for the icon. However, accessing properties specific to DockIcon or SystemTrayIcon on the wrong operating system generates a runtime error.

To set or change the image used for an icon, create an array containing one or more images and assign it to the `NativeApplication.nativeApplication.icon.bitmaps` property. The size of taskbar icons can be different on different operating systems. To avoid image degradation due to scaling, you can add multiple sizes of images to the `bitmaps` array. If you provide more than one image, AIR selects the size closest to the current display size of the taskbar icon, scaling it only if necessary. The following example sets the image for a taskbar icon using two images:

```
air.NativeApplication.nativeApplication.icon.bitmaps =
[bmp16x16.bitmapData, bmp128x128.bitmapData];
```

To change the icon image, assign an array containing the new image or images to the `bitmaps` property. You can animate the icon by changing the image in response to an `enterFrame` or `timer` event.

To remove the icon from the notification area on Windows and Linux, or to restore the default icon appearance on Mac OS X, set `bitmaps` to an empty array:

```javascript
air.NativeApplication.nativeApplication.icon.bitmaps = [];
```
Dock icons

AIR supports dock icons when `NativeApplication.supportsDockIcon` is true. The `NativeApplication.nativeApplication.icon` property represents the application icon on the dock (not a window dock icon).

*Note: AIR does not support changing window icons on the dock under Mac OS X. Also, changes to the application dock icon only apply while an application is running — the icon reverts to its normal appearance when the application terminates.*

Dock icon menus

You can add commands to the standard dock menu by creating a `NativeMenu` object containing the commands and assigning it to the `NativeApplication.nativeApplication.icon.menu` property. The items in the menu are displayed above the standard dock icon menu items.

Bouncing the dock

You can bounce the dock icon by calling the `NativeApplication.nativeApplication.icon.bounce()` method. If you set the `bounce()` priority parameter to informational, then the icon bounces once. If you set it to critical, then the icon bounces until the user activates the application. Constants for the `priority` parameter are defined in the `NotificationType` class.

*Note: The icon does not bounce if the application is already active.*

Dock icon events

When the dock icon is clicked, the NativeApplication object dispatches an `invoke` event. If the application is not running, the system launches it. Otherwise, the `invoke` event is delivered to the running application instance.

System Tray icons

AIR supports system tray icons when `NativeApplication.supportsSystemTrayIcon` is true, which is currently the case only on Windows and most Linux distributions. On Windows and Linux, system tray icons are displayed in the notification area of the taskbar. No icon is displayed by default. To show an icon, assign an array containing `BitmapData` objects to the `icon.bitmaps` property. To change the icon image, assign an array containing the new images to `bitmaps`. To remove the icon, set `bitmaps` to `null`.

System tray icon menus

You can add a menu to the system tray icon by creating a `NativeMenu` object and assigning it to the `NativeApplication.nativeApplication.icon.menu` property (no default menu is provided by the operating system). Access the system tray icon menu by right-clicking the icon.

System tray icon tooltips

Add a tooltip to an icon by setting the `tooltip` property:

```
air.NativeApplication.nativeApplication.icon.tooltip = "Application name";
```
System tray icon events
The SystemTrayIcon object referenced by the NativeApplication.nativeApplication.icon property dispatches a ScreenMouseEvent for click, mouseDown, mouseUp, rightClick, rightMouseDown, and rightMouseUp events. You can use these events, along with an icon menu, to allow users to interact with your application when it has no visible windows.

Example: Creating an application with no windows
The following example creates an AIR application which has a system tray icon, but no visible windows. (The visible property of the application must not be set to true in the application descriptor, or the window will be visible when the application starts up.)

```html
<html>
<head>
<script src="AIRAliases.js" language="JavaScript" type="text/javascript"></script>
</head>
<body>
<script language="JavaScript" type="text/javascript">
var iconLoadComplete = function(event)
{
    air.NativeApplication.nativeApplication.icon.bitmaps = [event.target.content.bitmapData];
}

air.NativeApplication.nativeApplication.autoExit = false;
var iconLoad = new air.Loader();
var iconMenu = new air.NativeMenu();
var exitCommand = iconMenu.addItem(new air.NativeMenuItem("Exit"));
exitCommand.addEventListener(air.Event.SELECT,function(event){
    air.NativeApplication.nativeApplication.icon.bitmaps = [];
    air.NativeApplication.nativeApplication.exit();
});

if (air.NativeApplication.supportsSystemTrayIcon) {
    air.NativeApplication.nativeApplication.autoExit = false;
    iconLoad.contentLoaderInfo.addEventListener(air.Event.COMPLETE,iconLoadComplete);
    iconLoad.load(new air.URLRequest("icons/AIRApp_16.png"));
    air.NativeApplication.nativeApplication.icon.tooltip = "AIR application";
    air.NativeApplication.nativeApplication.icon.menu = iconMenu;
}

if (air.NativeApplication.supportsDockIcon) {
    iconLoad.contentLoaderInfo.addEventListener(air.Event.COMPLETE,iconLoadComplete);
    iconLoad.load(new air.URLRequest("icons/AIRApp_128.png"));
    air.NativeApplication.nativeApplication.icon.menu = iconMenu;
}
</script>
</head>
<body>
</body>
</html>

Note: The example assumes that there are image files named AIRApp_16.png and AIRApp_128.png in an icons subdirectory of the application. (Sample icon files, which you can copy to your project folder, are included in the AIR SDK.)
Window taskbar icons and buttons

Iconified representations of windows are typically displayed in the window area of a taskbar or dock to allow users to easily access background or minimized windows. The Mac OS X dock displays an icon for your application as well as an icon for each minimized window. The Microsoft Windows and Linux taskbars display a button containing the program icon and title for each normal-type window in your application.

Highlighting the taskbar window button

When a window is in the background, you can notify the user that an event of interest related to the window has occurred. On Mac OS X, you can notify the user by bouncing the application dock icon (as described in “Bouncing the dock” on page 182). On Windows and Linux, you can highlight the window taskbar button by calling the notifyUser() method of the NativeWindow instance. The type parameter passed to the method determines the urgency of the notification:

- NotificationType.CRITICAL: the window icon flashes until the user brings the window to the foreground.
- NotificationType.INFORMATIONAL: the window icon highlights by changing color.

*Note:* On Linux, only the informational type of notification is supported. Passing either type value to the notifyUser() function will create the same effect.

The following statement highlights the taskbar button of a window:

```javascript
window.nativeWindow.notifyUser(air.NotificationType.INFORMATIONAL);
```

Calling the NativeWindow.notifyUser() method on an operating system that does not support window-level notification has no effect. Use the NativeWindow.supportsNotification property to determine if window notification is supported.

Creating windows without taskbar buttons or icons

On the Windows operating system, windows created with the types utility or lightweight do not appear on the taskbar. Invisible windows do not appear on the taskbar, either.

Because the initial window is necessarily of type, normal, in order to create an application without any windows appearing in the taskbar, you must either close the initial window or leave it invisible. To close all windows in your application without terminating the application, set the autoExit property of the NativeApplication object to false before closing the last window. To simply prevent the initial window from ever becoming visible, add `<visible>false</visible>` to the `<initialWindow>` element of the application descriptor file (and do not set the visible property to true or call the activate() method of the window).

In new windows opened by the application, set the type property of the NativeWindowInitOption object passed to the window constructor to NativeWindowType.UTILITY or NativeWindowType.LIGHTWEIGHT.

On Mac OS X, windows that are minimized are displayed on the dock taskbar. You can prevent the minimized icon from being displayed by hiding the window instead of minimizing it. The following example listens for a nativeWindowDisplayState change event and cancels it if the window is being minimized. Instead the handler sets the window visible property to false:
function preventMinimize(event) {
    if (event.afterDisplayState == air.NativeWindowDisplayState.MINIMIZED) {
        event.preventDefault();
        event.target.visible = false;
    }
}

If a window is minimized on the Mac OS X dock when you set the `visible` property to `false`, the dock icon is not removed. A user can still click the icon to make the window reappear.
Chapter 21: Working with the file system

You use the classes provided by the Adobe® AIR® file system API to access the file system of the host computer. Using these classes, you can access and manage directories and files, create directories and files, write data to files, and so on.

Additional online information about the AIR File API

You can find more information about using the File API classes from these sources:

Quick Starts (Adobe AIR Developer Connection)
- Building a text-file editor
- Building a directory search application
- Reading and writing from an XML preferences file

Language Reference
- File
- FileMode
- FileStream

Adobe Developer Connection Articles and Samples
- Adobe AIR Developer Connection for HTML and Ajax (search for 'AIR filesystem')

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:

Adobe AIR provides classes that you can use to access, create, and manage both files and folders. These classes, contained in the runtime.flash.filesystem package, are used as follows:

<table>
<thead>
<tr>
<th>File classes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>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.</td>
</tr>
<tr>
<td>FileMode</td>
<td>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.</td>
</tr>
<tr>
<td>FileStream</td>
<td>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.</td>
</tr>
</tbody>
</table>

Some methods in the File class have both synchronous and asynchronous versions:

- `File.copyTo()` and `File.copyToAsync()`
Working with the file system

- File.deleteDirectory() and File.deleteDirectoryAsync()
- File.deleteFile() and File.deleteFileAsync()
- File.getDirectoryListing() and File.getDirectoryListingAsync()
- File.moveTo() and File.moveToAsync()
- File.moveToTrash() and File.moveToTrashAsync()

Also, FileStream operations work synchronously or asynchronously depending on how the FileStream object opens the file: by calling the open() method or by calling the openAsync() method.

The asynchronous versions let you initiate processes that run in the background and dispatch events when complete (or when error events occur). Other code can execute while these asynchronous background processes are taking place. With asynchronous versions of the operations, you must set up event listener functions, using the addEventListener() method of the File or FileStream object that calls the function.

The synchronous versions let you write simpler code that does not rely on setting up event listeners. However, since other code cannot execute while a synchronous method is executing, important processes such as display object rendering and animation may be paused.

**Working with File objects**

A File object is a pointer to a file or directory in the file system.

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

**About the File class**

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
- Coping files and directories
- Moving files and directories
- Deleting files and directories (or moving them to the trash)
- Listing files and directories contained in a directory
- Creating temporary files and folders

Once a File object points to a file path, you can use it to read and write file data, using the FileStream class.

A File object can point to the path of a file or directory that does not yet exist. You can use such a File object in creating a file or directory.

**Paths of File objects**

Each File object has two properties that each define its path:
The File class includes static properties for pointing to standard directories on Mac OS, Windows, and Linux. These properties include:

- `File.applicationStorageDirectory`—a storage directory unique to each installed AIR application
- `File.applicationDirectory`—the directory where the application is installed (along with any installed assets)
- `File.desktopDirectory`—the user’s desktop directory
- `File.documentsDirectory`—the user’s documents directory
- `File.userDirectory`—the user directory

These properties have meaningful values on different operating systems. For example, Mac OS, Linux, and Windows each have different native paths to the user’s desktop directory. However, the `File.desktopDirectory` property points to the correct desktop directory path on each of these platforms. To write applications that work well across platforms, use these properties as the basis for referencing other directories and files used by the application. Then use the `resolvePath()` method to refine the path. For example, this code points to the preferences.xml file in the application storage directory:

```javascript
var prefsFile:File = air.File.applicationStorageDirectory;
prefsFile = prefsFile.resolvePath("preferences.xml");
```

Although the File class lets you point to a specific file path, doing so can lead to applications that will not work across platforms. For example, the path `C:\Documents and Settings\joe\` only works on Windows. For these reasons, it is best to use the static properties of the File class, such as `File.documentsDirectory`.

More information on these File properties is provided in the next section.

### Pointing a File object to a directory

There are different ways to set a File object to point to a directory.

#### Pointing to the user’s home directory

You can point a File object to the user’s home directory. The path varies by operating system. On Windows XP, for example, 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. On Linux, it is the /home/.userName directory. The following code sets a File object to point to an AIR Test subdirectory of the home directory:

```javascript
var file = air.File.userDirectory.resolvePath("AIR Test");
```
Pointing to the user’s documents directory

You can point a File object to the user’s documents directory. On Windows, the default location is the “My Documents” directory (for example, “C:\Documents and Settings\userName\My Documents”). On Mac OS, the default location is the Users/userName/Documents directory. On Linux, the default location is the /home/userName/Documents directory. The following code sets a File object to point to an AIR Test subdirectory of the documents directory:

```javascript
var file = air.File.documentsDirectory.resolvePath("AIR Test");
```

Pointing to the desktop directory

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:

```javascript
var file = air.File.desktopDirectory.resolvePath("AIR Test");
```

Pointing to the application storage directory

You can point a File object to the application storage directory. For every AIR application, there is a unique associated path that defines the application storage directory. This directory is unique to each application and user. You may want to use this directory to store user-specific, application-specific data (such as user data or preferences files). For example, the following code points a File object to a preferences file, prefs.xml, contained in the application storage directory:

```javascript
var file = air.File.applicationStorageDirectory;
file = file.resolvePath("prefs.xml");
```

The application storage directory location is based on the user name, the application ID, and, if defined, the publisher ID:

- **On Mac OS**—In:
  
  `/Users/user name/Library/Preferences/applicationID.publisherID/Local Store/`

  For example:

  `/Users/babbage/Library/Preferences/com.example.TestApp.02D88EEED35F84C264A183921344EEA353A629FD.1/Local Store`

- **On Windows**—In the documents and Settings directory, in:

  `user name/Application Data/applicationID.publisherID/Local Store/`

  For example:

  `C:\Documents and Settings\babbage\Application Data\com.example.TestApp.02D88EEED35F84C264A183921344EEA353A629FD.1\Local Store`

- **On Linux**—In:

  `/home/user name/.appdata/applicationID.publisherID/Local Store/`

  For example:

  `/home/babbage/.appdata/com.example.TestApp.02D88EEED35F84C264A183921344EEA353A629FD.1/Local Store`

The URL (and `url` property) for a File object created with `File.applicationStorageDirectory` uses the `app-storage` URL scheme (see “Supported URL schemes” on page 193), as in the following:

```javascript
var dir = air.File.applicationStorageDirectory;
dir = dir.resolvePath("prefs.xml");
air.trace(dir.url); // app-storage:/preferences
**Note:** If the publisher ID changes during an update signed with a migration signature, AIR creates a new application storage directory for the updated application. The files in the original directory are not removed, but the `applicationStorageDirectory` property and `app-storage` URL schemes resolve to the new directory instead of the old one. A publisher ID change can only occur with updates that specify AIR version 1.5.2, or earlier, in the application descriptor. After AIR 1.5.3, the publisher ID cannot change as a result of an update.

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

```javascript
var dir = air.File.applicationDirectory;
dir = dir.resolvePath("images");
```

The URL (and `url` property) for a File object created with `File.applicationDirectory` uses the `app` URL scheme (see “Supported URL schemes” on page 193), as in the following:

```javascript
var dir = air.File.applicationDirectory;
dir = dir.resolvePath("images");
air.trace(dir.url); // app:/images
```

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

### Pointing to an explicit 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):

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

**Important:** Pointing to an explicit path this way can lead to code that does not work across platforms. For example, the previous example only works on Windows. You should use the static properties of the File object, such as `File.applicationStorageDirectory`, to locate a directory that works cross-platform. Then use the `resolvePath()` method (see the next section) to navigate to a relative path.

### Navigating to relative paths
You can use the `resolvePath()` method to obtain a path relative to another given path. For example, the following code sets a File object to point to an "AIR Test" subdirectory of the user's home directory:

```javascript
var file = air.File.userDirectory;
file = file.resolvePath("AIR Test");
```

You can also use the `url` property of a File object to point it to a directory based on a URL string, as in the following:

```javascript
var urlStr = "file:///C:/AIR Test/";
var file = new air.File()
file.url = urlStr;
```

For more information, see “Modifying File paths” on page 192.
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:

```javascript
var file = new air.File();
file.addEventListener(air.Event.SELECT, dirSelected);
file.browseForDirectory("Select a directory");
function dirSelected(event) {
    alert(file.nativePath);
}
```

Pointing to the directory from which the application was invoked
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 329.

Pointing a File object to a file
There are different ways to set the file to which a File object points.

Pointing to an explicit file path
Important: Pointing to an explicit path can lead to code that does not work across platforms. For example, the path `C:/foo.txt` only works on Windows. You should use the static properties of the File object, such as `File.applicationStorageDirectory`, to locate a directory that works cross-platform. Then use the `resolvePath()` method (see “Modifying File paths” on page 192) to navigate to a relative path.

You can use the `url` property of a File object to point it to a file or directory based on a URL string, as in the following:

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

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

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

The `url` property always returns the URI-encoded version of the URL (for example, blank spaces are replaced with `%20`):

```javascript
file.url = "file:///c:/AIR Test";
alert(file.url); // file:///c:/AIR%20Test
```

You can also use the `nativePath` property of a File object to set an explicit path. For example, the following code, when run on a Windows computer, sets a File object to the `test.txt` file in the AIR Test subdirectory of the C: drive:

```javascript
var file = new air.File();
file.nativePath = "C:/AIR Test/test.txt";
```

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

```javascript
var file = new air.File("C:/AIR Test/test.txt");
```
Use the forward slash (/) character as the path delimiter for the `nativePath` property. On Windows, you can also use the backslash (\) character, but doing so leads to applications that do not work across platforms.

For more information, see “Modifying File paths” on page 192.

**Enumerating files in a directory**

You can use the `getDirectoryListing()` method of a `File` object to get an array of `File` objects pointing to files and subdirectories at the root level of a directory. For more information, see “Enumerating directories” on page 197.

**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()`
- `browseForOpenMultiple()`

These methods are each asynchronous. The `browseForOpen()` and `browseForSave()` methods dispatch the `select` event when the user selects a file (or a target path, in the case of `browseForSave()`). With the `browseForOpen()` and `browseForSave()` methods, upon selection the target `File` object points to the selected files. The `browseForOpenMultiple()` method dispatches a `selectMultiple` event when the user selects files. The `selectMultiple` event is of type `FileListEvent`, which has a `files` property that is an array of `File` objects (pointing to the selected files).

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

```javascript
var fileToOpen = air.File.documentsDirectory;
selectTextFile(fileToOpen);

function selectTextFile(root)
{
    var txtFilter = new air.FileFilter("Text", "*.as;*.css;*.html;*.txt;*.xml");
    root.browseForOpen("Open", new window.runtime.Array(txtFilter));
    root.addEventListener(air.Event.SELECT, fileSelected);
}

function fileSelected(event)
{
    trace(fileToOpen.nativePath);
}
```

If the application has another browser dialog box open when you call a browse method, the runtime throws an Error exception.

**Modifying File paths**

You can also modify the path of an existing `File` object by calling the `resolvePath()` method or by modifying the `nativePath` or `url` property of the object, as in the following examples (on Windows):
file1 = air.File.documentsDirectory;
file1 = file1.resolvePath("AIR Test");
alert(file1.nativePath); // C:\Documents and Settings\userName\My Documents\AIR Test
var file2 = air.File.documentsDirectory;
file2 = file2.resolvePath("..");
alert(file2.nativePath); // C:\Documents and Settings\userName
var file3 = air.File.documentsDirectory;
file3.nativePath += "/subdirectory";
alert(file3.nativePath); // C:\Documents and Settings\userName\My Documents\subdirectory
var file4 = new air.File();
file4.url = "file:///c:/AIR Test/test.txt";
alert(file4.nativePath); // C:\AIR Test\test.txt

When using the nativePath property, use the forward slash (/) character as the directory separator character. On Windows, you can use the backslash (\) character as well, but you should not do so, as it leads to code that does not work cross-platform.

**Supported URL schemes**

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

<table>
<thead>
<tr>
<th>URL scheme</th>
<th>Description</th>
</tr>
</thead>
</table>
| file | Use to specify a path relative to the root of the file system. For example:  
  file:///c:/AIR Test/test.txt  
  The URL standard specifies that a file URL takes the form file://<host>/<path>. As a special case, <host> can be the empty string, which is interpreted as "the machine from which the URL is being interpreted." For this reason, file URLs often have three slashes (///). |
| app | Use 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 an images subdirectory of the directory of the installed application:  
  app:/images |
| app-storage | Use to specify a path relative to the application store directory. For each installed application, AIR defines a unique application store directory, which is a useful place to store data specific to that application. For example, the following path points to a 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 getRelativePath() method to find the relative path between two files:

```javascript
var file1 = air.File.documentsDirectory
file1 = file1.resolvePath("AIR Test");
var file2 = air.File.documentsDirectory
file2 = file2.resolvePath("AIR Test/bob/test.txt");

alert(file1.getRelativePath(file2)); // bob/test.txt
```

The second parameter of the getRelativePath() method, the useDotDot parameter, allows for .. syntax to be returned in results, to indicate parent directories:
var file1 = air.File.documentsDirectory;
  file1 = file1.resolvePath("AIR Test");
var file2 = air.File.documentsDirectory;
  file2 = file2.resolvePath("AIR Test/bob/test.txt");
var file3 = air.File.documentsDirectory;
  file3 = file3.resolvePath("AIR Test/susan/test.txt");

alert(file2.getRelativePath(file1, true)); // ../..
alert(file3.getRelativePath(file2, true)); // ../../../bob/test.txt

**Obtaining canonical versions of file names**

File and path names are not case sensitive on Windows and Mac OS. In the following, two File objects point to the same file:

File.documentsDirectory.resolvePath("test.txt");
File.documentsDirectory.resolvePath("TeSt.TxT");

However, documents and directory names do include capitalization. For example, the following assumes that there is a folder named AIR Test in the documents directory, as in the following examples:

var file = air.File.documentsDirectory;
  file = file.resolvePath("AIR test");
trace(file.nativePath); // ... AIR test
file.canonicalize();
alert(file.nativePath); // ... AIR Test

The `canonicalize()` method converts the `nativePath` object to use the correct capitalization for the file or directory name. On case sensitive file systems (such as Linux), when multiple files exists with names differing only in case, the `canonicalize()` method adjusts the path to match the first file found (in an order determined by the file system).

You can also use the `canonicalize()` method to convert short file names ("8.3" names) to long file names on Windows, as in the following examples:

var path = new air.File();
  path.nativePath = "C:\AIR~1"
  path.canonicalize();
alert(path.nativePath); // C:\AIR Test

**Working with packages and symbolic links**

Various operating systems support package files and symbolic link files:

**Packages**—On Mac OS, directories can be designated as packages and show up in the Mac OS Finder as a single file rather than as a directory.

**Symbolic links**—Mac OS, Linux, and Windows Vista support symbolic links. Symbolic links allow a file to point to another file or directory on disk. Although similar, symbolic links are not the same as aliases. An alias is always reported as a file (rather than a directory), and reading or writing to an alias or shortcut never affects the original file or directory that it points to. On the other hand, a symbolic link behaves exactly like the file or directory it points to. It can be reported as a file or a directory, and reading or writing to a symbolic link affects the file or directory that it points to, not the symbolic link itself. Additionally, on Windows the `isSymbolicLink` property for a File object referencing a junction point (used in the NTFS file system) is set to `true`.

The File class includes the `isPackage` and `isSymbolicLink` properties for checking if a File object references a package or symbolic link.

The following code iterates through the user's desktop directory, listing subdirectories that are not packages:
The following code iterates through the user’s desktop directory, listing files and directories that are not symbolic links:

```javascript
var desktopNodes = air.File.desktopDirectory.getDirectoryListing();
for (i = 0; i < desktopNodes.length; i++)
{
    if (!desktopNodes[i].isSymbolicLink)
    {
        air.trace(desktopNodes[i].name);
    }
}
```

The `canonicalize()` method changes the path of a symbolic link to point to the file or directory to which the link refers. The following code iterates through the user’s desktop directory, and reports the paths referenced by files that are symbolic links:

```javascript
var desktopNodes = air.File.desktopDirectory.getDirectoryListing();
for (i = 0; i < desktopNodes.length; i++)
{
    if (desktopNodes[i].isSymbolicLink)
    {
        var linkNode = desktopNodes[i];
        linkNode.canonicalize();
        air.trace(desktopNodes[i].name);
    }
}
```

**Determining space available on a volume**

The `spaceAvailable` property of a File object is the space available for use at the File location, in bytes. For example, the following code checks the space available in the application storage directory:

```javascript
air.trace(air.File.applicationStorageDirectory.spaceAvailable);
```

If the File object references a directory, the `spaceAvailable` property indicates the space in the directory that files can use. If the File object references a file, the `spaceAvailable` property indicates the space into which the file could grow. If the file location does not exist, the `spaceAvailable` property is set to 0. If the File object references a symbolic link, the `spaceAvailable` property is set to space available at the location the symbolic link points to.

Typically the space available for a directory or file is the same as the space available on the volume containing the directory or file. However, space available can take into account quotas and per-directory limits.

Adding a file or directory to a volume generally requires more space than the actual size of the file or the size of the contents of the directory. For example, the operating system may require more space to store index information. Or the disk sectors required may use additional space. Also, available space changes dynamically. So, you cannot expect to allocate all of the reported space for file storage. For information on writing to the file system, see "Reading and writing files" on page 201.
Getting file system information

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File.lineEnding</td>
<td>The line-ending character sequence used by the host operating system. On Mac OS and Linux, this is the line-feed character. On Windows, this is the carriage return character followed by the line-feed character.</td>
</tr>
<tr>
<td>File.separator</td>
<td>The host operating system's path component separator character. On Mac OS and Linux, this is the forward slash (/) character. On Windows, it is the backslash () character.</td>
</tr>
<tr>
<td>File.systemCharset</td>
<td>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.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capabilities.hasIME</td>
<td>Specifies whether the player is running on a system that does (true) or does not (false) have an input method editor (IME) installed.</td>
</tr>
<tr>
<td>Capabilities.language</td>
<td>Specifies the language code of the system on which the player is running.</td>
</tr>
<tr>
<td>Capabilities.os</td>
<td>Specifies the current operating system.</td>
</tr>
</tbody>
</table>

Note: Be careful when using Capabilities.os to determine system characteristics. If a more specific property exists to determine a system characteristic, use it. Otherwise, you run the risk of writing code that will not work correctly on all platforms. For example, consider the following code:

```javascript
var separator:String;
if (Capabilities.os.indexOf("Mac") > -1)
{
  separator = "/";
}
else
{
  separator = "\";
}
```

This code will lead to problems on Linux. It is better to simply use the File.separator property.

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 188.

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:

```javascript
var dir = air.File.userDirectory.resolvePath("AIR Test");
dir.createDirectory();
```
If the directory exists, the `createDirectory()` method does nothing.

Also, in some modes, a `FileStream` object creates directories when opening files. Missing directories are created when you instantiate a `FileStream` instance with the `fileMode` parameter of the `FileStream()` constructor set to `FileMode.APPEND` or `FileMode.WRITE`. For more information, see “Workflow for reading and writing files” on page 201.

### Creating a temporary directory

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

```javascript
var temp = air.File.createTempDirectory();
```

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

You 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 `getDirectoryListing()` method or the `getDirectoryListingAsync()` method of a `File` object to get an array of `File` objects pointing to files and subfolders in a directory.

For example, the following code lists the contents of the user’s documents directory (without examining subdirectories):

```javascript
var directory = air.File.documentsDirectory;
var contents = directory.getDirectoryListing();
for (i = 0; i < contents.length; i++)
{
    alert(contents[i].name, contents[i].size);
}
```

When using the asynchronous version of the method, the `directoryListing` event object has a `files` property that is the array of `File` objects pertaining to the directories:

```javascript
var directory = air.File.documentsDirectory;
directory.getDirectoryListingAsync();
directory.addEventListener(air.FileListEvent.DIRECTORY_LISTING, dirListHandler);

function dirListHandler(event)
{
    var contents = event.files;
    for (i = 0; i < contents.length; i++)
    {
        alert(contents[i].name, contents[i].size);
    }
}
```

### Copying and moving directories

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:

```javascript
var directory = air.File.documentsDirectory;
directory.copyTo(new File(‘myNewDirectory’), true);
```
var sourceDir = air.File.documentsDirectory.resolvePath("AIR Test");
var resultDir = air.File.documentsDirectory.resolvePath("AIR Test Copy");
sourceDir.copyTo(resultDir);

When you specify true for the overwrite parameter of the copyTo() method, all files and folders in an existing target
directory are deleted and replaced with the files and folders in the source directory (even if the target file does not exist
in the source directory).

The directory that you specify as the newLocation parameter of the copyTo() method specifies the path to the
resulting directory; it does not specify the parent directory that will contain the resulting directory.

For details, see “Copying and moving files” on page 199.

Deleting directory contents

The File class includes a deleteDirectory() method and a deleteDirectoryAsync() method. These methods
delete directories, the first working synchronously, the second working asynchronously (see “AIR file basics” on
page 186). Both methods include a deleteDirectoryContents parameter (which takes a Boolean value); when this
parameter is set to true (the default value is false) the call to the method deletes non-empty directories; otherwise,
only empty directories are deleted.

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

var directory = air.File.documentsDirectory.resolvePath("AIR Test");
directory.deleteDirectory(true);

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

var directory = air.File.documentsDirectory.resolvePath("AIR Test");
directory.addEventListener(air.Event.COMPLETE, completeHandler)
directory.deleteDirectoryAsync(true);

function completeHandler(event) {
    alert("Deleted.")
}

Also included are the moveToTrash() and moveToTrashAsync() methods, which you can use to move a directory to
the System trash. For details, see “Moving a file to the trash” on page 200.

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, refer to “AIR
security” on page 100, if you haven’t already done so.

Note: You can associate a file type with an AIR application (so that double-clicking it opens the application). For details,
see "Managing file associations" on page 337.

Getting file information

The File class includes the following properties that provide information about a file or directory to which a File object
points:
For details on these properties, see the File class entry in the Adobe AIR Language Reference for HTML Developers (http://www.adobe.com/go/learn_air_html_jslr).

**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 186).

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:

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

In this example, the value of `overwrite` parameter of the `copyTo()` method (the second parameter) is set to `true`. By setting this to `true`, an existing target file is overwritten. This parameter is optional. If you set it to `false` (the default value), the operation dispatches an IOErrorEvent event if the target file exists (and the file is not copied).

The “Async” versions of the copy and move methods work asynchronously. Use the `addEventListener()` method to monitor completion of the task or error conditions, as in the following code:
var original = air.File.documentsDirectory;
original = original.resolvePath("AIR Test/test.txt");

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

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

function fileMoveCompleteHandler(event) {
    alert(event.target); // [object File]
}
function fileMoveIOErrorEventHandler(event) {
    alert("I/O Error.");
}

The File class also includes the File.moveToTrash() and File.moveToTrashAsync() methods, which move a file or directory to the system trash.

### Deleting a file

The File class includes a deleteFile() method and a deleteFileAsync() method. These methods delete files, the first working synchronously, the second working asynchronously (see “AIR file basics” on page 186).

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

```javascript
var file = air.File.documentsDirectory.resolvePath("test.txt");
file.deleteFile();
```

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

```javascript
var file = air.File.documentsDirectory.resolvePath("test.txt");
file.addEventListener(air.Event.COMPLETE, completeHandler)
file.deleteFileAsync();

function completeHandler(event) {
    alert("Deleted.")
}
```

Also included are the moveToTrash() and moveToTrashAsync methods, which you can use to move a file or directory to the System trash. For details, see “Moving a file to the trash” on page 200.

### Moving a file to the trash

The File class includes a moveToTrash() method and a moveToTrashAsync() method. These methods send a file or directory to the System trash, the first working synchronously, the second working asynchronously (see “AIR file basics” on page 186).

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

```javascript
var file = air.File.documentsDirectory.resolvePath("test.txt");
file.moveToTrash();
```
Creating a temporary file
The File class includes a `createTempFile()` method, which creates a file in the temporary directory folder for the System, as in the following example:

```javascript
var temp = air.File.createTempFile();
```

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

You 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 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.

Reading and writing files
The FileStream class lets AIR applications read and write to the file system.

Workflow for reading and writing files
The workflow for reading and writing files is as follows.

**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).

```javascript
var file = air.File.documentsDirectory;
file = file.resolvePath("AIR Test/testFile.txt");
```

This example uses the `File.documentsDirectory` property and the `resolvePath()` method of a File object to initialize the File object. However, there are many other ways to point a File object to a file. For more information, see “Pointing a File object to a file” on page 191.

**Initialize a FileStream object.**

**Call the open() method or the openAsync() method of the FileStream object.**
The method you call depends on whether you want to open the file for synchronous or asynchronous operations. Use the File object as the `file` parameter of the open method. For the `fileMode` parameter, specify a constant from the FileMode class that specifies the way in which you will use the file.

For example, the following code initializes a FileStream object that is used to create a file and overwrite any existing data:

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

For more information, see “Initializing a FileStream object, and opening and closing files” on page 203 and “FileStream open modes” on page 202.
If you opened the file asynchronously (using the openAsync() method), add and set up event listeners for the FileStream object.

These event listener methods respond to events dispatched by the FileStream object in 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 207.

Include code for reading and writing data, as needed.

There are many methods of the FileStream class related to reading and writing. (They each begin with "read" or "write"). The method you choose to use to read or write data depends on the format of the data in the target file.

For example, if the data in the target file is UTF-encoded text, you may use the readUTFBytes() and writeUTFBytes() methods. If you want to deal with the data as byte arrays, you may use the readByte(), readBytes(), writeByte(), and writeBytes() methods. For details, see “Data formats, and choosing the read and write methods to use” on page 207.

If you opened the file asynchronously, then be sure that enough data is available before calling a read method. For details, see “The read buffer and the bytesAvailable property of a FileStream object” on page 205.

Before writing to a file, if you want to check the amount of disk space available, you can check the spaceAvailable property of the File object. For more information, see “Determining space available on a volume” on page 195.

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 203.

To see a sample application that uses the FileStream class to read and write files, see the following articles at the Adobe AIR Developer Center:

- Building a text-file editor
- Reading and writing from an XML preferences file

Working with FileStream objects

The FileStream class defines methods for opening, reading, and writing files.

FileStream open modes

The open() and openAsync() methods of a FileStream object each include a fileMode parameter, which defines some properties for a file stream, including the following:

- The ability to read from the file
- The ability to write to the file
- Whether data will always be appended past the end of the file (when writing)
- What to do when the file does not exist (and when its parent directories do not exist)

The following are the various file modes (which you can specify as the fileMode parameter of the open() and openAsync() methods):
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:

```javascript
var myFile = air.File.documentsDirectory;
myFile = myFile.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.open(myFile, air.FileMode.READ);
```

The `fileMode` parameter (the second parameter of the `open()` and `openAsync()` methods), specifies the mode in which to open the file: for read, write, append, or update. For details, see the previous section, “FileStream open modes” on page 202.

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 205.

You can call the `close()` method of a FileStream object to close the associated file, making it available for use by other applications.

### File mode Description

<table>
<thead>
<tr>
<th>File mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileMode.READ</td>
<td>Specifies that the file is open for reading only.</td>
</tr>
<tr>
<td>FileMode.WRITE</td>
<td>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.</td>
</tr>
<tr>
<td>FileMode.APPEND</td>
<td>Specifies that the file is open for appending. The file is created if it does not exist. If the file exists, existing data is not overwritten, and all writing begins at the end of the file.</td>
</tr>
<tr>
<td>FileMode.UPDATE</td>
<td>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).</td>
</tr>
</tbody>
</table>
The position property of a FileStream object

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

Before a read or write operation, set the `position` property to any valid position in the file.

For example, the following code writes the string "hello" (in UTF encoding) at position 8 in the file:

```javascript
var myFile = air.File.documentsDirectory;
myFile = myFile.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.open(myFile, air.FileMode.UPDATE);
myFileStream.position = 8;
myFileStream.writeUTFBytes("hello");
```

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

Before a read operation, the value of `position` must be at least 0 and less than the number of bytes in the file (which are existing positions in the file).

The value of the `position` property is modified only in the following conditions:

- When you explicitly set the `position` property.
- When you call a read method.
- When you call a write method.

When you call a read or write method of a FileStream object, the `position` property is immediately incremented by the number of bytes that you read or write. Depending on the read method you use, the `position` property is either incremented by the number of bytes you specify to read or by the number of bytes available. When you call a read or write method subsequently, it reads or writes starting at the new position.

```javascript
var myFile = air.File.documentsDirectory;
myFile = myFile.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.open(myFile, air.FileMode.UPDATE);
myFileStream.position = 4000;
alert(myFileStream.position);  // 4000
myFileStream.writeBytes(myByteArray, 0, 200);
alert(myFileStream.position);  // 4200
```

There is, however, one exception: for a FileStream opened in append mode, the `position` property is not changed after a call to a write method. (In append mode, data is always written to the end of the file, independent of the value of the `position` property.)

For a file opened for asynchronous operations, the write operation does not complete before the next line of code is executed. However, you can call multiple asynchronous methods sequentially, and the runtime executes them in order:
var myFile = air.File.documentsDirectory;
myFile = myFile.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.openAsync(myFile, air.FileMode.WRITE);
myFileStream.writeUTFBytes("hello");
myFileStream.writeUTFBytes("world");
myFileStream.addEventListener(air.Event.CLOSE, closeHandler);
myFileStream.close();
air.trace("started.");

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

The read buffer and the bytesAvailable property of a FileStream object
When a FileStream object with read capabilities (one in which the fileMode parameter of the open() or openAsync() method was set to READ or UPDATE) is opened, the runtime stores the data in an internal buffer. The FileStream object begins reading data into the buffer as soon as you open the file (by calling the open() or openAsync() method of the FileStream object).

For a file opened for synchronous operations (using the open() method), you can always set the position pointer to any valid position (within the bounds of the file) and begin reading any amount of data (within the bounds of the file), as shown in the following code (which assumes that the file contains at least 100 bytes):

var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.open(myFile, air.FileMode.READ);
myFileStream.position = 10;
myFileStream.readBytes(myByteArray, 0, 20);
myFileStream.position = 89;
myFileStream.readBytes(myByteArray, 0, 10);

Whether a file is opened for synchronous or asynchronous operations, the read methods always read from the "available" bytes, represented by the bytesAvailable property. When reading synchronously, all of the bytes of the file are available all of the time. When reading asynchronously, the bytes become available starting at the position specified by the position property, in a series of asynchronous buffer fills signaled by progress events.
For files opened for *synchronous* operations, the `bytesAvailable` property is always set to represent the number of bytes from the `position` property to the end of the file (all bytes in the file are always available for reading).

For files opened for *asynchronous* operations, you need to ensure that the read buffer has consumed enough data before calling a read method. For a file opened asynchronously, as the read operation progresses, the data from the file, starting at the `position` specified when the read operation started, is added to the buffer, and the `bytesAvailable` property increments with each byte read. The `bytesAvailable` property indicates the number of bytes available starting with the byte at the position specified by the `position` property to the end of the buffer. Periodically, the `FileStream` object sends a `progress` event.

For a file opened asynchronously, as data becomes available in the read buffer, the `FileStream` object periodically dispatches the `progress` event. For example, the following code reads data into a `ByteArray` object, `bytes`, as it is read into the buffer:

```javascript
var bytes = new air.ByteArray();
var myFile = new air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.addEventListener(air.ProgressEvent.PROGRESS, progressHandler);
myFileStream.openAsync(myFile, air.FileMode.READ);
function progressHandler(event)
{
    myFileStream.readBytes(bytes, myFileStream.position, myFileStream.bytesAvailable);
}
```

For a file opened asynchronously, only the data in the read buffer can be read. Furthermore, as you read the data, it is removed from the read buffer. For read operations, you need to ensure that the data exists in the read buffer before calling the read operation. For example, the following code reads 8000 bytes of data starting from position 4000 in the file:

```javascript
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.addEventListener(air.ProgressEvent.PROGRESS, progressHandler);
myFileStream.addEventListener(air.Event.COMPLETE, completed);
myFileStream.openAsync(myFile, air.FileMode.READ);
myFileStream.position = 4000;
var str = "";
function progressHandler(event)
{
    if (myFileStream.bytesAvailable > 8000 )
    {
        str += myFileStream.readMultiByte(8000, "iso-8859-1");
    }
}
```

During a write operation, the `FileStream` object does not read data into the read buffer. When a write operation completes (all data in the write buffer is written to the file), the `FileStream` object starts a new read buffer (assuming that the associated `FileStream` object was opened with read capabilities), and starts reading data into the read buffer, starting from the position specified by the `position` property. The `position` property may be the position of the last byte written, or it may be a different position, if the user specifies a different value for the `position` object after the write operation.
Asynchronous programming and the events generated by a FileStream object opened asynchronously

When a file is opened asynchronously (using the openAsync() method), reading and writing files are done asynchronously. As data is read into the read buffer and as output data is being written, other ActionScript code can execute.

This means that you need to register for events generated by the FileStream object opened asynchronously.

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

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

function progressHandler(event)
{
    str += myFileStream.readMultiByte(myFileStream.bytesAvailable, "iso-8859-1");
}
```

You can read the entire data by registering for the complete event, as in the following code:

```actionscript
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.addEventListener(air.Event.COMPLETE, completed);
myFileStream.openAsync(myFile, air.FileMode.READ);
var str = ""

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

In much the same way that input data is buffered to enable asynchronous reading, data that you write on an asynchronous stream is buffered and written to the file asynchronously. As data is written to a file, the FileStream object periodically dispatches an OutputProgressEvent object. An OutputProgressEvent object includes a bytesPending property that is set to the number of bytes remaining to be written. You can register for the outputProgress event to be notified as this buffer is actually written to the file, perhaps in order to display a progress dialog. However, in general, it is not necessary to do so. In particular, you may call the close() method without concern for the unwritten bytes. The FileStream object will continue writing data and the close event will be delivered after the final byte is written to the file and the underlying file is closed.

Data formats, and choosing the read and write methods to use

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:

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

function progressHandler(event)
{
    str += myFileStream.readMultiByte(myFileStream.bytesAvailable, "iso-8859-1");
}
```

You can read the entire data by registering for the complete event, as in the following code:

```actionscript
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.addEventListener(air.Event.COMPLETE, completed);
myFileStream.openAsync(myFile, air.FileMode.READ);
var str = ""

function completeHandler(event)
{
    str = myFileStream.readMultiByte(myFileStream.bytesAvailable, "iso-8859-1");
}
```
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.addEventListener(air.Event.COMPLETE, completeHandler);
myFileStream.openAsync(myFile, air.FileMode.READ);
var bytes = new air.ByteArray();

function completeHandler(event)
{
    myFileStream.readBytes(bytes, 0, myFileStream.bytesAvailable);
}

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

var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.open(myFile, air.FileMode.WRITE);
myFileStream.writeBytes(bytes, 0, bytes.length);

However, often you do not want to store the data in an ActionScript ByteArray object. And often the data file is in a specified file format. For example, the data in the file may be in a text file format, and you may want to represent such data in a String object. For this reason, the FileStream class includes read and write methods for reading and writing data to and from types other than ByteArray objects. For example, the readMultiByte() method lets you read data from a file and store it to a string, as in the following code:

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

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

The second parameter of the readMultiByte() method specifies the text format that ActionScript uses to interpret the data ("iso-8859-1" in the example). ActionScript supports common character set encodings, and these are listed in the ActionScript 3.0 Language Reference (see Supported character sets at 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 variable length, do not use readUTFBytes() in a method that responds to the progress event, since the data at the end of the read buffer may represent an incomplete character. (This is also true when using the readMultiByte() method with a variable-length character encoding.) For this reason, read the entire set of data when the FileStream object dispatches the complete event.

There are also similar write methods, writeMultiByte() and writeUTFBytes(), for working with String objects and text files.

The readUTF() and the writeUTF() methods (not to be confused with readUTFBytes() and writeUTFBytes()) also read and write the text data to a file, but they assume that the text data is preceded by data specifying the length of the text data, which is not a common practice in standard text files.
Some UTF-encoded text files begin with a "UTF-BOM" (byte order mark) character that defines the endianness as well as the encoding format (such as UTF-16 or UTF-32).

For an example of reading and writing to a text file, see “Example: Reading an XML file into an XML object” on page 210.

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 some other read and write methods (such as readDouble() and writeDouble()). However, if you use these, make sure that the file format matches the formats of the data defined by these methods.

File formats are often more complex than simple text formats. For example, an MP3 file includes compressed data that can only be interpreted with the decompression and decoding algorithms specific to MP3 files. MP3 files also may include ID3 tags that contain meta tag information about the file (such as the title and artist for a song). There are multiple versions of the ID3 format, but the simplest (ID3 version 1) is discussed in the “Example: Reading and writing data with random access” on page 211 section.

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

### Using the load() and save() methods

Flash Player 10 added the load() and save() methods to the FileReference class. These methods are also in AIR 1.5, and the File class inherits the methods from the FileReference class. These methods were designed to provide a secure means for users to load and save file data in Flash Player. However, AIR applications can also use these methods as an easy way to load and save files asynchronously.

For example, the following code saves a string to a text file:

```javascript
var file = air.File.applicationStorageDirectory.resolvePath("test.txt");
var str = "Hello."
file.addEventListener(air.Event.COMPLETE, fileSaved);
file.save(str);
function fileSaved(event)
{
    air.trace("Done.");
}
```

The data parameter of the save() method can take a String or ByteArray value. When the argument is a String value, the method saves the file as a UTF-8–encoded text file.

When this code sample executes, the application displays a dialog box in which the user selects the saved file destination.

The following code loads a string from a UTF-8–encoded text file:
var file = air.File.applicationStorageDirectory.resolvePath("test.txt");
file.addEventListener(air.Event.COMPLETE, loaded);
file.load();
var str;
function loaded(event)
{
    var bytes = file.data;
    str = bytes.readUTFBytes(bytes.length);
    air.trace(str);
}

The FileStream class provides more functionality than that provided by the load() and save() methods:

- Using the FileStream class, you can read and write data both synchronously and asynchronously.
- Using the FileStream class lets you write incrementally to a file.
- Using the FileStream class lets you open a file for random access (both reading from and writing to any section of the file).
- The FileStream class lets you specify the type of file access you have to the file, by setting the fileMode parameter of the open() or openAsync() method.
- The FileStream class lets you save data to files without presenting the user with an Open or Save dialog box.
- You can directly use types other than byte arrays when reading data with the FileStream class.

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

The following examples demonstrate how to read and write to a text file that contains XML data.

To read from the file, initialize the File and FileStream objects, call the readUTFBytes() method of the FileStream and convert the string to an XML object:

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

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

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

var outputString = '<xml version="1.0" encoding="utf-8">\n<prefs><autoSave>true</autoSave></prefs>'

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

These examples use the readUTFBytes() and writeUTFBytes() methods, because they assume that the files are in UTF-8 format. If not, you may need to use a different method (see “Data formats, and choosing the read and write methods to use” on page 207).
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:

```javascript
var file = air.File.documentsDirectory.resolvePath("AIR Test/preferences.xml");
var fileStream = new air.FileStream();
fileStream.addEventListener(air.Event.COMPLETE, processXMLData);
fileStream.openAsync(file, air.FileMode.READ);

var prefsXML;
function processXMLData(event)
{
    var xmlString = fileStream.readUTFBytes(fileStream.bytesAvailable);
    prefsXML = domParser.parseFromString(xmlString, "text/xml");
    fileStream.close();
}
```

The `processXMLData()` method is invoked when the entire file is read into the read buffer (when the FileStream object dispatches the `complete` event). It calls the `readUTFBytes()` method to get a string version of the read data, and it creates an XML object, `prefsXML`, based on that string.

To see a sample application that shows these capabilities, see [Reading and writing from an XML Preferences File](#).

### 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 meta data identifying the recording. The ID3 tag format itself has different revisions. This example describes how to read and write from an MP3 file that contains the simplest ID3 format (ID3 version 1.0) using "random access to file data", which means that it reads from and writes to arbitrary locations in the file.

An MP3 file that contains an ID3 version 1 tag includes the ID3 data at the end of the file, in the final 128 bytes. When accessing a file for random read/write access, it is important to specify `FileManager.UPDATE` as the `fileMode` parameter for the `open()` or `openAsync()` method:

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

This lets you both read and write to the file.

Upon opening the file, you can set the `position` pointer to the position 128 bytes before the end of the file:

```javascript
fileStr.position = file.size - 128;
```

This code sets the `position` property to this location in the file because the ID3 v1.0 format specifies that the ID3 tag data is stored in the last 128 bytes of the file. The specification also says the following:

- The first 3 bytes of the tag contain the string "TAG".
- The next 30 characters contain the title for the MP3 track, as a string.
- The next 30 characters contain the name of the artist, as a string.
- The next 30 characters contain the name of the album, as a string.
- The next 4 characters contain the year, as a string.
- The next 30 characters contain the comment, as a string.
The next byte contains a code indicating the track's genre.

All text data is in ISO 8859-1 format.

The `id3TagRead()` method checks the data after it is read in (upon the complete event):

```javascript
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:

```javascript
fileStr.position = file.length - 125;    // 128 - 3
fileStr.writeMultiByte(newTitle, "iso-8859-1");
```

To conform with the ID3 version 1 standard, the length of the `newTitle` string should be 30 characters, padded at the end with the character code 0 (`String.fromCharCode(0)`).
Chapter 22: Drag and drop

Use the classes in the drag-and-drop API to support user-interface drag-and-drop gestures. A gesture in this sense is an action by the user, mediated by both the operating system and your application, expressing an intent to copy, move, or link information. A drag-out gesture occurs when the user drags an object out of a component or application. A drag-in gesture occurs when the user drags an object in from outside a component or application.

With the drag-and-drop API, you can allow a user to drag data between applications and between components within an application. Supported transfer formats include:

- Bitmaps
- Files
- HTML-formatted text
- Text
- URLs

Additional online information about dragging and dropping

You can find more information about working with the drag and drop API from these sources:

Adobe Developer Connection Articles and Samples
- Adobe AIR Developer Connection for HTML and Ajax (search for 'AIR drag and drop')

Drag and drop basics

To drag data into and out of an HTML-based application (or into and out of the HTML displayed in an HTMLLoader), you can use HTML drag and drop events. The HTML drag-and-drop API allows you to drag to and from DOM elements in the HTML content.

**Note:** You can also use the AIR NativeDragEvent and NativeDragManager APIs by listening for events on the HTMLLoader object containing the HTML content. However, the HTML API is better integrated with the HTML DOM and gives you control of the default behavior. The NativeDragEvent and NativeDragManager APIs are not commonly used in HTML-based applications and so are not covered in the Adobe AIR Language Reference for HTML Developers (http://www.adobe.com/go/learn_air_html_jslr). For more information about using these classes, refer to Developing AIR Applications with Adobe Flex 3 (http://www.adobe.com/go/learn_air_flex3) and the Flex 3 Language Reference (http://www.adobe.com/go/learn_flex3_aslr).
Default drag-and-drop behavior

The HTML environment provides default behavior for drag-and-drop gestures involving text, images, and URLs. Using the default behavior, you can always drag these types of data out of an element. However, you can only drag text into an element and only to elements in an editable region of a page. When you drag text between or within editable regions of a page, the default behavior performs a move action. When you drag text to an editable region from a non-editable region or from outside the application, then the default behavior performs a copy action.

You can override the default behavior by handling the drag-and-drop events yourself. To cancel the default behavior, you must call the `preventDefault()` methods of the objects dispatched for the drag-and-drop events. You can then insert data into the drop target and remove data from the drag source as necessary to perform the chosen action.

By default, the user can select and drag any text, and drag images and links. You can use the WebKit CSS property, `webkit-user-select` to control how any HTML element can be selected. For example, if you set `webkit-user-select` to `none`, then the element contents are not selectable and so cannot be dragged. You can also use the `webkit-user-drag` CSS property to control whether an element as a whole can be dragged. However, the contents of the element are treated separately. The user could still drag a selected portion of the text. For more information, see “Extensions to CSS” on page 80.

Drag-and-drop events in HTML

The events dispatched by the initiator element from which a drag originates, are:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dragstart</td>
<td>Dispatched when the user starts the drag gesture. The handler for this event can prevent the drag, if necessary, by calling the <code>preventDefault()</code> method of the event object. To control whether the dragged data can be copied, linked, or moved, set the <code>effectAllowed</code> property. Selected text, images, and links are put onto the clipboard by the default behavior, but you can set different data for the drag gesture using the <code>dataTransfer</code> property of the event object.</td>
</tr>
<tr>
<td>drag</td>
<td>Dispatched continuously during the drag gesture.</td>
</tr>
<tr>
<td>dragend</td>
<td>Dispatched when the user releases the mouse button to end the drag gesture.</td>
</tr>
</tbody>
</table>

The events dispatched by a drag target are:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dragover</td>
<td>Dispatched continuously while the drag gesture remains within the element boundaries. The handler for this event should set the <code>dataTransfer.dropEffect</code> property to indicate whether the drop will result in a copy, move, or link action if the user releases the mouse.</td>
</tr>
<tr>
<td>dragenter</td>
<td>Dispatched when the drag gesture enters the boundaries of the element. If you change any properties of a <code>dataTransfer</code> object in a <code>dragenter</code> event handler, those changes are quickly overridden by the next <code>dragover</code> event. On the other hand, there is a short delay between a <code>dragenter</code> and the first <code>dragover</code> event that can cause the cursor to flash if different properties are set. In many cases, you can use the same event handler for both events.</td>
</tr>
<tr>
<td>dragleave</td>
<td>Dispatched when the drag gesture leaves the element boundaries.</td>
</tr>
<tr>
<td>drop</td>
<td>Dispatched when the user drops the data onto the element. The data being dragged can only be accessed within the handler for this event.</td>
</tr>
</tbody>
</table>
The event object dispatched in response to these events is similar to a mouse event. You can use mouse event properties such as (clientX, clientY) and (screenX, screenY), to determine the mouse position.

The most important property of a drag event object is dataTransfer, which contains the data being dragged. The dataTransfer object itself has the following properties and methods:

<table>
<thead>
<tr>
<th>Property or Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>effectAllowed</td>
<td>The effect allowed by the source of the drag. Typically, the handler for the dragstart event sets this value. See Drag effects in HTML.</td>
</tr>
<tr>
<td>dropEffect</td>
<td>The effect chosen by the target or the user. If you set the dropEffect in a dragover or dragenter event handler, then AIR updates the mouse cursor to indicate the effect that occurs if the user releases the mouse. If the dropEffect set does not match one of the allowed effects, no drop is allowed and the unavailable cursor is displayed. If you have not set a dropEffect in response to the latest dragover or dragenter event, then the user can choose from the allowed effects with the standard operating system modifier keys. The final effect is reported by the dropEffect property of the object dispatched for dragend. If the user abandons the drop by releasing the mouse outside an eligible target, then dropEffect is set to none.</td>
</tr>
<tr>
<td>types</td>
<td>An array containing the MIME type strings for each data format present in the dataTransfer object.</td>
</tr>
<tr>
<td>getData(mimeType)</td>
<td>Gets the data in the format specified by the mimeType parameter. The getData() method can only be called in response to the drop event.</td>
</tr>
<tr>
<td>setData(mimeType)</td>
<td>Adds data to the dataTransfer in the format specified by the mimeType parameter. You can add data in multiple formats by calling setData() for each MIME type. Any data placed in the dataTransfer object by the default drag behavior is cleared. The setData() method can only be called in response to the dragstart event.</td>
</tr>
<tr>
<td>clearData(mimeType)</td>
<td>Clears any data in the format specified by the mimeType parameter.</td>
</tr>
<tr>
<td>setDragImage(image, offsetX, offsetY)</td>
<td>Sets a custom drag image. The setDragImage() method can only be called in response to the dragstart event and only when an entire HTML element is dragged by setting its -webkit-user-drag CSS style to element. The image parameter can be a JavaScript Element or Image object.</td>
</tr>
</tbody>
</table>

**MIME types for the HTML drag-and-drop**

The MIME types to use with the dataTransfer object of an HTML drag-and-drop event include:

<table>
<thead>
<tr>
<th>Data format</th>
<th>MIME type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>&quot;text/plain&quot;</td>
</tr>
<tr>
<td>HTML</td>
<td>&quot;text/html&quot;</td>
</tr>
<tr>
<td>URL</td>
<td>&quot;text/uri-list&quot;</td>
</tr>
<tr>
<td>Bitmap</td>
<td>&quot;image/x-vnd.adobe.air.bitmap&quot;</td>
</tr>
<tr>
<td>File list</td>
<td>&quot;application/x-vnd.adobe.air.file-list&quot;</td>
</tr>
</tbody>
</table>

You can also use other MIME strings, including application-defined strings. However, other applications may not be able to recognize or use the transferred data. It is your responsibility to add data to the dataTransfer object in the expected format.

**Important:** Only code running in the application sandbox can access dropped files. Attempting to read or set any property of a File object within a non-application sandbox generates a security error. See Handling file drops in non-application HTML sandboxes for more information.
Drag effects in HTML

The initiator of the drag gesture can limit the allowed drag effects by setting the `dataTransfer.effectAllowed` property in the handler for the `dragstart` event. The following string values can be used:

<table>
<thead>
<tr>
<th>String value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;none&quot;</td>
<td>No drag operations are allowed.</td>
</tr>
<tr>
<td>&quot;copy&quot;</td>
<td>The data will be copied to the destination, leaving the original in place.</td>
</tr>
<tr>
<td>&quot;link&quot;</td>
<td>The data will be shared with the drop destination using a link back to the original.</td>
</tr>
<tr>
<td>&quot;move&quot;</td>
<td>The data will be copied to the destination and removed from the original location.</td>
</tr>
<tr>
<td>&quot;copyLink&quot;</td>
<td>The data can be copied or linked.</td>
</tr>
<tr>
<td>&quot;copyMove&quot;</td>
<td>The data can be copied or moved.</td>
</tr>
<tr>
<td>&quot;linkMove&quot;</td>
<td>The data can be linked or moved.</td>
</tr>
<tr>
<td>&quot;all&quot;</td>
<td>The data can be copied, moved, or linked. All is the default effect when you prevent the default behavior.</td>
</tr>
</tbody>
</table>

The target of the drag gesture can set the `dataTransfer.dropEffect` property to indicate the action that is taken if the user completes the drop. If the drop effect is one of the allowed actions, then the system displays the appropriate copy, move, or link cursor. If not, then the system displays the `unavailable` cursor. If no drop effect is set by the target, the user can choose from the allowed actions with the modifier keys.

Set the `dropEffect` value in the handlers for both the `dragover` and `dragenter` events:

```javascript
function doDragStart(event) {
    event.dataTransfer.setData("text/plain","Text to drag");
    event.dataTransfer.effectAllowed = "copyMove";
}

function doDragOver(event) {
    event.dataTransfer.dropEffect = "copy";
}

function doDragEnter(event) {
    event.dataTransfer.dropEffect = "copy";
}
```

*Note: Although you should always set the `dropEffect` property in the handler for `dragenter`, be aware that the next `dragover` event resets the property to its default value. Set `dropEffect` in response to both events.*

Dragging data out of an HTML element

The default behavior allows most content in an HTML page to be copied by dragging. You can control the content allowed to be dragged using CSS properties `-webkit-user-select` and `-webkit-user-drag`.

Override the default drag-out behavior in the handler for the `dragstart` event. Call the `setData()` method of the `dataTransfer` property of the event object to put your own data into the drag gesture.
To indicate which drag effects a source object supports when you are not relying on the default behavior, set the `dataTransfer.effectAllowed` property of the event object dispatched for the `dragstart` event. You can choose any combination of effects. For example, if a source element supports both `copy` and `link` effects, set the property to "copyLink".

**Setting the dragged data**

Add the data for the drag gesture in the handler for the `dragstart` event with the `dataTransfer` property. Use the `dataTransfer.setData()` method to put data onto the clipboard, passing in the MIME type and the data to transfer.

For example, if you had an image element in your application, with the id `imageOfGeorge`, you could use the following dragstart event handler. This example adds representations of a picture of George in several data formats, which increases the likelihood that other applications can use the dragged data.

```javascript
function dragStartHandler(event){
    event.dataTransfer.effectAllowed = "copy";
    var dragImage = document.getElementById("imageOfGeorge");
    var dragFile = new air.File(dragImage.src);
    event.dataTransfer.setData("text/plain","A picture of George");
    event.dataTransfer.setData("image/x-vnd.adobe.air.bitmap", dragImage);
    event.dataTransfer.setData("application/x-vnd.adobe.air.file-list",
                             new Array(dragFile));
}
```

*Note:* When you call the `setData()` method of `dataTransfer` object, no data is added by the default drag-and-drop behavior.

**Dragging data into an HTML element**

The default behavior only allows text to be dragged into editable regions of the page. You can specify that an element and its children can be made editable by including the `contentEditable` attribute in the opening tag of the element. You can also make an entire document editable by setting the document object `designMode` property to "on".

You can support alternate drag-in behavior on a page by handling the `dragenter`, `dragover`, and `drop` events for any elements that can accept dragged data.

**Enabling drag-in**

To handle the drag-in gesture, you must first cancel the default behavior. Listen for the `dragenter` and `dragover` events on any HTML elements you want to use as drop targets. In the handlers for these events, call the `preventDefault()` method of the dispatched event object. Canceling the default behavior allows non-editable regions to receive a drop.

**Getting the dropped data**

You can access the dropped data in the handler for the `ondrop` event:

```javascript
function doDrop(event){
    droppedText = event.dataTransfer.getData("text/plain");
}
```
Use the `dataTransfer.getData()` method to read the data onto the clipboard, passing in the MIME type of the data format to read. You can find out which data formats are available using the `types` property of the `dataTransfer` object. The `types` array contains the MIME type string of each available format.

When you cancel the default behavior in the dragenter or dragover events, you are responsible for inserting any dropped data into its proper place in the document. No API exists to convert a mouse position into an insertion point within an element. This limitation can make it difficult to implement insertion-type drag gestures.

**Example: Overriding the default HTML drag-in behavior**

This example implements a drop target that displays a table showing each data format available in the dropped item. The default behavior is used to allow text, links, and images to be dragged within the application. The example overrides the default drag-in behavior for the `div` element that serves as the drop target. The key step to enabling non-editable content to accept a drag-in gesture is to call the `preventDefault()` method of the event object dispatched for both the `dragenter` and `dragover` events. In response to a `drop` event, the handler converts the transferred data into an HTML row element and inserts the row into a table for display.

```html
<html>
<head>
<title>Drag-and-drop</title>
<script language="javascript" type="text/javascript" src="AIRAliases.js"></script>
<script language="javascript" type="text/javascript">

function init(){
    var target = document.getElementById('target');
    target.addEventListener("dragenter", dragEnterOverHandler);
    target.addEventListener("dragover", dragEnterOverHandler);
    target.addEventListener("drop", dropHandler);

    var source = document.getElementById('source');
    source.addEventListener("dragstart", dragStartHandler);
    source.addEventListener("dragend", dragEndHandler);

    emptyRow = document.getElementById("emptyTargetRow");
}

function dragStartHandler(event){
    event.dataTransfer.effectAllowed = "copy";
}

function dragEndHandler(event){
    air.trace(event.type + ": " + event.dataTransfer.dropEffect);
}

function dragEnterOverHandler(event){
    event.preventDefault();
}

var emptyRow;
function dropHandler(event){
    for(var prop in event){
        air.trace(prop + " = " + event[prop]);
    }
    var row = document.createElement('tr');
```

```html
```
row.innerHTML = "<td>" + event.dataTransfer.getData("text/plain") + "</td>" +  
"<td>" + event.dataTransfer.getData("text/html") + "</td>" +  
"<td>" + event.dataTransfer.getData("text/uri-list") + "</td>" +  
"<td>" + event.dataTransfer.getData("application/x-vnd.adobe.air.file-list") + "</td>";

var imageCell = document.createElement('td');
if((event.dataTransfer.types.toString()).search("image/x-vnd.adobe.air.bitmap") > -1){
    imageCell.appendChild(event.dataTransfer.getData("image/x-vnd.adobe.air.bitmap"));}
row.appendChild(imageCell);
var parent = emptyRow.parentNode;
parent.insertBefore(row, emptyRow);
Handling file drops in non-application HTML sandboxes

Non-application content cannot access the File objects that result when files are dragged into an AIR application. Nor is it possible to pass one of these File objects to application content through a sandbox bridge. (The object properties must be accessed during serialization.) However, you can still drop files in your application by listening for the AIR nativeDragDrop events on the HTMLLoader object.

Normally, if a user drops a file into a frame that hosts non-application content, the drop event does not propagate from the child to the parent. However, since the events dispatched by the HTMLLoader (which is the container for all HTML content in an AIR application) are not part of the HTML event flow, you can still receive the drop event in application content.

To receive the event for a file drop, the parent document adds an event listener to the HTMLLoader object using the reference provided by window.htmlLoader:

```javascript
window.htmlLoader.addEventListener("nativeDragDrop",function(event){
    var filelist = event.clipboard.getData(air.ClipboardFormats.FILE_LIST_FORMAT);
    air.trace(filelist[0].url);
});
```

The NativeDragEvent objects behave like their HTML event counterparts, but the names of some of the properties and methods are different. For example, the HTML `dataTransfer` property is the AIR `clipboard` property. The NativeDragEvent and NativeDragManager APIs are not covered in the Adobe AIR Language Reference for HTML Developers. For more information about using these classes, refer to Developing AIR Applications with Adobe Flex 3 and the Flex 3 Language Reference.

The following example uses a parent document that loads a child page into a remote sandbox (http://localhost/). The parent listens for the `nativeDragDrop` event on the HTMLLoader object and traces out the file url.

```html
<html>
<head>
<title>Drag-and-drop in a remote sandbox</title>
<script language="javascript" type="text/javascript" src="AIRAliases.js"></script>
<script language="javascript">
    window.htmlLoader.addEventListener("nativeDragDrop",function(event){
        var filelist = event.clipboard.getData(air.ClipboardFormats.FILE_LIST_FORMAT);
        air.trace(filelist[0].url);
    });
</script>
</head>
<body>
    <iframe src="child.html"
        sandboxRoot="http://localhost/
        documentRoot="/n        frameborder="0" width="100%" height="100%">
    </iframe>
</body>
</html>
```

The child document must present a valid drop target by preventing the Event object `preventDefault()` method in the HTML `dragenter` and `dragover` event handlers or the drop event can never occur.
<html>
<head>
  <title>Drag and drop target</title>
  <script language="javascript" type="text/javascript">
    function preventDefault(event)
    {
        event.preventDefault();
    }
  </script>
</head>
<body ondragenter="preventDefault(event)" ondragover="preventDefault(event)">
  <div>
    <h1>Drop Files Here</h1>
  </div>
</body>
</html>
Chapter 23: Copy and paste

Use the classes in the clipboard API to copy information to and from the system clipboard. The data formats that can be transferred into or out of an Adobe® AIR® application include:

- Bitmaps
- Files
- Text
- HTML-formatted text
- Rich Text Format data
- URL strings
- Serialized objects
- Object references (only valid within the originating application)

Additional online information about copying and pasting

You can find more information about copying and pasting from these sources:

Quick Starts (Adobe AIR Developer Connection)

Language Reference
- Clipboard
- ClipboardFormats
- ClipboardTransferMode

More Information
- Adobe AIR Developer Connection for HTML and Ajax (search for 'AIR copy and paste')

Copy-and-paste basics

The copy-and-paste API contains the following classes.
The static `Clipboard.generalClipboard` property represents the operating system clipboard. The `Clipboard` class provides methods for reading and writing data to clipboard objects. New `Clipboard` objects can also be created to transfer data through the drag-and-drop API.

The HTML environment provides an alternate API for copy and paste. Either API can be used by code running within the application sandbox, but only the HTML API can be used in non-application content. (See “HTML copy and paste” on page 224.)

The `HTMLLoader` and `TextField` classes implement default behavior for the normal copy and paste keyboard shortcuts. To implement copy and paste shortcut behavior for custom components, you can listen for these keystrokes directly. You can also use native menu commands along with key equivalents to respond to the keystrokes indirectly.

Different representations of the same information can be made available in a single `Clipboard` object to increase the ability of other applications to understand and use the data. For example, an image might be included as image data, a serialized `Bitmap` object, and as a file. Rendering of the data in a format can be deferred so that the format is not actually created until the data in that format is read.

**Note:** On Linux, clipboard data does not persist when the AIR application closes.

### Reading from and writing to the system clipboard

To read the operating system clipboard, call the `getData()` method of the `Clipboard.generalClipboard` object, passing in the name of the format to read:

```javascript
if(air.Clipboard.generalClipboard.hasFormat("text/plain")){
    var text = air.Clipboard.generalClipboard.getData("text/plain");
}
```

To write to the clipboard, add the data to the `Clipboard.generalClipboard` object in one or more formats. Any existing data in the same format is overwritten automatically. However, it is a good practice to also clear the system clipboard before writing new data to it to make sure that unrelated data in any other formats is also deleted.

```javascript
var textToCopy = "Copy to clipboard.*;
air.Clipboard.generalClipboard.clear();
air.Clipboard.generalClipboard.setData("text/plain", textToCopy, false);
```

**Note:** Only code running in the application sandbox can access the system clipboard directly. In non-application HTML content, you can only access the clipboard through the `clipboardData` property of an event object dispatched by one of the HTML copy or paste events.
HTML copy and paste

The HTML environment provides its own set of events and default behavior for copy and paste. Only code running in the application sandbox can access the system clipboard directly through the AIR Clipboard.generalClipboard object. JavaScript code in a non-application sandbox can access the clipboard through the event object dispatched in response to one of the copy or paste events dispatched by an element in an HTML document.

Copy and paste events include: copy, cut, and paste. The object dispatched for these events provides access to the clipboard through the clipboardData property.

Default behavior

By default, AIR copies selected items in response to the copy command, which can be generated either by a keyboard shortcut or a context menu. Within editable regions, AIR cuts text in response to the cut command or pastes text to the cursor or selection in response to the paste command.

To prevent the default behavior, your event handler can call the preventDefault() method of the dispatched event object.

Using the clipboardData property of the event object

The clipboardData property of the event object dispatched as a result of one of the copy or paste events allows you to read and write clipboard data.

To write to the clipboard when handling a copy or cut event, use the setData() method of the clipboardData object, passing in the data to copy and the MIME type:

```javascript
function customCopy(event){
    event.clipboardData.setData("text/plain", "A copied string.");
}
```

To access the data that is being pasted, you can use the getData() method of the clipboardData object, passing in the MIME type of the data format. The available formats are reported by the types property.

```javascript
function customPaste(event){
    var pastedData = event.clipboardData("text/plain");
}
```

The getData() method and the types property can only be accessed in the event object dispatched by the paste event.

The following example illustrates how to override the default copy and paste behavior in an HTML page. The copy event handler italicizes the copied text and copies it to the clipboard as HTML text. The cut event handler copies the selected data to the clipboard and removes it from the document. The paste handler inserts the clipboard contents as HTML and styles the insertion as bold text.
Menu commands and keystrokes for copy and paste

Copy and paste functionality is commonly triggered through menu commands and keyboard shortcuts. On OS X, an edit menu with the copy and paste commands is automatically created by the operating system, but you must add listeners to these menu commands to hook up your own copy and paste functions. On Windows, you can add a native edit menu to any window that uses system chrome. (You can also create non-native menus with DHTML, but that is beyond the scope of this discussion.)

To trigger copy and paste commands in response to keyboard shortcuts, you can either assign key equivalents to the appropriate command items in a native application or window menu, or you can listen for the keystrokes directly.
Starting a copy or paste operation with a menu command

To trigger a copy or paste operation with a menu command, you must add listeners for the `select` event on the menu items that call your handler functions.

In HTML content, the default copy and paste behavior can be triggered using the NativeApplication edit commands. For example, the NativeApplication copy() method sends a copy command to the page, just as if the CMD-C or CTRL-C keys were pressed on the keyboard. Similar commands are available for cut, paste, and select all. The following example creates an edit menu for an editable HTML document:

```html
<html>
<head>
<title>Edit Menu</title>
<script src="AIRAliases.js" type="text/javascript"></script>
<script language="javascript" type="text/javascript">

function init(){
    document.designMode = "On";
    addEditMenu();
}

function addEditMenu(){
    var menu = new air.NativeMenu;
    var edit = menu.addSubmenu(new air.NativeMenu(), "Edit");

    var copy = edit.submenu.addItem(new air.NativeMenuItem("Copy"));
    var cut = edit.submenu.addItem(new air.NativeMenuItem("Cut"));
    var paste = edit.submenu.addItem(new air.NativeMenuItem("Paste"));
    var selectAll = edit.submenu.addItem(new air.NativeMenuItem("Select All"));

    copy.addEventListener(air.Event.SELECT, function(){
        air.NativeApplication.nativeApplication.copy();
    });
    cut.addEventListener(air.Event.SELECT, function(){
        air.NativeApplication.nativeApplication.cut();
    });
    paste.addEventListener(air.Event.SELECT, function(){
        air.NativeApplication.nativeApplication.paste();
    });
    selectAll.addEventListener(air.Event.SELECT, function(){
        air.NativeApplication.nativeApplication.selectAll();
    });

</script>
</head>
</html>
```
The previous example replaces the application menu on Mac OS X, but you can also make use of the default Edit menu by finding the existing items and adding event listeners to them.

### Finding default menu items on Mac OS X

To find the default edit menu and the specific copy, cut, and paste command items in the application menu on Mac OS X, you can search through the menu hierarchy using the `label` property of the `NativeMenuItem` objects. For example, the following function takes a name and finds the item with the matching label in the menu:

```javascript
function findItemByName(menu, name, recurse) {
  var searchItem = null;
  for (var i = 0; i < menu.items.length; i++) {
    if (menu.items[i].label == name) {
      searchItem = menu.items[i];
      break;
    }
    if ((menu.items[i].submenu != null) && recurse) {
      searchItem = findItemByName(menu.items[i].submenu, name, recurse);
    }
    if (searchItem != null) { break; }
  }
  return searchItem;
}
```

You can set the `recurse` parameter to `true` to include submenus in the search, or `false` to include only the passed-in menu.

### Starting a copy or paste command with a keystroke

If your application uses native window or application menus for copy and paste, you can add key equivalents to the menu items to implement keyboard shortcuts.

In HTML content, the keyboard shortcuts for copy and paste commands are implemented by default. It is not possible to trap all of the keystrokes commonly used for copy and paste using a key event listener. If you need to override the default behavior, a better strategy is to listen for the `copy` and `paste` events themselves.
Clipboard data formats

Clipboard formats describe the data placed in a Clipboard object. AIR automatically translates the standard data formats between AIR objects and system clipboard formats. In addition, application objects can be transferred within and between AIR applications using application-defined formats.

A Clipboard object can contain representations of the same information in different formats. For example, a Clipboard object representing an AIR object could include a reference format for use within the same application, a serialized format for use by another AIR application, a bitmap format for use by an image editor, and a file list format, perhaps with deferred rendering to encode a PNG file, for copying or dragging a representation of the object to the file system.

Standard data formats

The constants defining the standard format names are provided in the ClipboardFormats class:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXT_FORMAT</td>
<td>Text-format data is translated to and from the ActionScript String class.</td>
</tr>
<tr>
<td>HTML_FORMAT</td>
<td>Text with HTML markup.</td>
</tr>
<tr>
<td>RICH_TEXT_FORMAT</td>
<td>Rich-text-format data is translated to and from the ActionScript ByteArray class. The RTF markup is not interpreted or translated in any way.</td>
</tr>
<tr>
<td>BITMAP_FORMAT</td>
<td>Bitmap-format data is translated to and from the ActionScript BitmapData class.</td>
</tr>
<tr>
<td>FILE_LIST_FORMAT</td>
<td>File-list-format data is translated to and from an array of ActionScript File objects.</td>
</tr>
<tr>
<td>URL_FORMAT</td>
<td>URL-format data is translated to and from the ActionScript String class.</td>
</tr>
</tbody>
</table>

When copying and pasting data in response to a `copy`, `cut`, or `paste` event in HTML content, MIME types must be used instead of the ClipboardFormat strings. The valid data MIME types are:

<table>
<thead>
<tr>
<th>MIME type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>&quot;text/plain&quot;</td>
</tr>
<tr>
<td>URL</td>
<td>&quot;text/uri-list&quot;</td>
</tr>
<tr>
<td>Bitmap</td>
<td>&quot;image/x-vnd.adobe.air.bitmap&quot;</td>
</tr>
<tr>
<td>File list</td>
<td>&quot;application/x-vnd.adobe.air.file-list&quot;</td>
</tr>
</tbody>
</table>

**Note:** Rich text format data is not available from the `clipboardData` property of the event object dispatched as a result of a `paste` event within HTML content.

Custom data formats

You can use application-defined custom formats to transfer objects as references or as serialized copies. References are only valid within the same AIR application. Serialized objects can be transferred between Adobe AIR applications, but can only be used with objects that remain valid when serialized and deserialized. Objects can usually be serialized if their properties are either simple types or serializable objects.

To add a serialized object to a Clipboard object, set the serializable parameter to `true` when calling the `Clipboard.setData()` method. The format name can be one of the standard formats or an arbitrary string defined by your application.
Transfer modes
When an object is written to the clipboard using a custom data format, the object data can be read from the clipboard either as reference or as a serialized copy of the original object. AIR defines four transfer modes that determine whether objects are transferred as references or as serialized copies:

<table>
<thead>
<tr>
<th>Transfer mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClipboardTransferModes.ORIGINAL_ONLY</td>
<td>Only a reference is returned. If no reference is available, a null value is returned.</td>
</tr>
<tr>
<td>ClipboardTransferModes.ORIGINAL_PREFERRED</td>
<td>A reference is returned, if available. Otherwise a serialized copy is returned.</td>
</tr>
<tr>
<td>ClipboardTransferModes.CLONE_ONLY</td>
<td>Only a serialized copy is returned. If no serialized copy is available, then a null value is returned.</td>
</tr>
<tr>
<td>ClipboardTransferModes.CLONE_PREFERRED</td>
<td>A serialized copy is returned, if available. Otherwise a reference is returned.</td>
</tr>
</tbody>
</table>

Reading and writing custom data formats
You can use any string that does not begin with the reserved prefix `air:` for the format parameter when writing an object to the clipboard. Use the same string as the format to read the object. The following examples illustrate how to read and write objects to the clipboard:

```javascript
function createClipboardObject(object) {
    var transfer = new air.Clipboard();
    transfer.setData("object", object, true);
}
```

To extract a serialized object from the clipboard object (after a drop or paste operation), use the same format name and the `cloneOnly` or `clonePreferred` transfer modes.

```javascript
var transfer = clipboard.getData("object", air.ClipboardTransferMode.CLONE_ONLY);
```

A reference is always added to the Clipboard object. To extract the reference from the clipboard object (after a drop or paste operation), instead of the serialized copy, use the `originalOnly` or `originalPreferred` transfer modes:

```javascript
var transferredObject = clipboard.getData("object", air.ClipboardTransferMode.ORIGINAL_ONLY);
```

References are only valid if the Clipboard 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 Clipboard object using the `setDataHandler()` method. The function must return the data in the appropriate format. For example, if you called `setDataHandler(ClipboardFormat.TEXT_FORMAT, writeText)`, then the `writeText()` function must return a string.

If a data format of the same type is added to a Clipboard object with the `setData()` method, that data will take precedence over the deferred version (the rendering function is never called). The rendering function may or may not be called again if the same clipboard data is accessed a second time.
Note: On Mac OS X, deferred rendering does not occur when using the standard AIR clipboard formats. The rendering function is called immediately.

Pasting text using a deferred rendering function

The following example illustrates how to implement a deferred rendering function.

When the Copy button in the example is pressed, the application clears the system clipboard to ensure that no data is left over from previous clipboard operations, then puts the renderData() function onto the clipboard with the clipboard setDataHandler() method.

When the Paste button is pressed, the application accesses the clipboard and sets the destination text. Since the text data format on the clipboard has been set with a function rather than a string, the clipboard will call the renderData() function. The renderData() function returns the text in the source text, which is then assigned to the destination text.

Notice that if you edit the source text before pressing the Paste button, the edit will be reflected in the pasted text, even when the edit occurs after the copy button was pressed. This is because the rendering function doesn’t copy the source text until the paste button is pressed. (When using deferred rendering in a real application, you might want to store or protect the source data in some way to prevent this problem.)

```html
<html>
<head>
  <title>Deferred rendering</title>
  <script src="AIRAliases.js" type="text/javascript"></script>
  <script language="javascript" type="text/javascript">
    function doCopy(){
      air.Clipboard.generalClipboard.clear();
      air.Clipboard.generalClipboard.setDataHandler(air.ClipboardFormats.TEXT_FORMAT, renderData);
    }
    function doPaste(){
      document.getElementById("destination").innerHTML = air.Clipboard.generalClipboard.getData(air.ClipboardFormats.TEXT_FORMAT);
    }
    function renderData(){
      air.trace("Rendering data");
      return document.getElementById("source").innerHTML;
    }
  </script>
</head>
<body>
  <button onClick="doCopy()">Copy</button>
  <button onClick="doPaste()">Paste</button>
  <p>Source:</p>
  <p id="source" contentEditable="true">Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit.</p>
  <hr>
  <p>Destination:</p>
</body>
</html>
```
Chapter 24: Working with byte arrays

The ByteArray class allows you to read from and write to a binary stream of data, which is essentially an array of bytes. This class provides a way to access data at the most elemental level. Because computer data consists of bytes, or groups of 8 bits, the ability to read data in bytes means that you can access data for which classes and access methods do not exist. The ByteArray class allows you to parse any stream of data, from a bitmap to a stream of data traveling over the network, at the byte level.

The writeObject() method allows you to write an object in serialized Action Message Format (AMF) to a ByteArray, while the readObject() method allows you to read a serialized object from a ByteArray to a variable of the original data type. You can serialize any object except for display objects, which are those objects that can be placed on the display list. You can also assign serialized objects back to custom class instances if the custom class is available to the runtime. After converting an object to AMF, you can efficiently transfer it over a network connection or save it to a file.

The sample Adobe® AIR® application described here reads a .zip file as an example of processing a byte stream; extracting a list of the files that the .zip file contains and writing them to the desktop.

Reading and writing a ByteArray

The ByteArray class is part of the flash.utils package; you can also use the alias air.ByteArray to refer to the ByteArray class if your code includes the AIRAliases.js file. To create a ByteArray, invoke the ByteArray constructor as shown in the following example:

```
var stream = new air.ByteArray();
```

ByteArray methods

Any meaningful data stream is organized into a format that you can analyze to find the information that you want. A record in a simple employee file, for example, would probably include an ID number, a name, an address, a phone number, and so on. An MP3 audio file contains an ID3 tag that identifies the title, author, album, publishing date, and genre of the file that’s being downloaded. The format allows you to know the order in which to expect the data on the data stream. It allows you to read the byte stream intelligently.

The ByteArray class includes several methods that make it easier to read from and write to a data stream. Some of these methods include readBytes() and writeBytes(), readInt() and writeInt(), readFloat() and writeFloat(), readObject() and writeObject(), and readUTFBytes() and writeUTFBytes(). These methods enable you to read data from the data stream into variables of specific data types and write from specific data types directly to the binary data stream.

For example, the following code reads a simple array of strings and floating-point numbers and writes each element to a ByteArray. The organization of the array allows the code to call the appropriate ByteArray methods (writeUTFBytes() and writeFloat()) to write the data. The repeating data pattern makes it possible to read the array with a loop.
// The following example reads a simple Array (groceries), made up of strings
// and floating-point numbers, and writes it to a ByteArray.

// define the grocery list Array
var groceries = ["milk", 4.50, "soup", 1.79, "eggs", 3.19, "bread", 2.35]
// define the ByteArray
var bytes = new air.ByteArray();
// for each item in the array
for (i = 0; i < groceries.length; i++) {
    bytes.writeUTFBytes(groceries[i++]); // write the string and position to the next item
    bytes.writeFloat(groceries[i]);// write the float
    air.trace("bytes.position is: " + bytes.position);// display the position in ByteArray
}
air.trace("bytes length is: " + bytes.length); // display the length

The position property
The position property stores the current position of the pointer that indexes the ByteArray during reading or writing. The initial value of the position property is 0 (zero) as shown in the following code:

var bytes = new air.ByteArray();
air.trace("bytes.position is initially: " + bytes.position); // 0

When you read from or write to a ByteArray, the method that you use updates the position property to point to the location immediately following the last byte that was read or written. For example, the following code writes a string to a ByteArray and afterward the position property points to the byte immediately following the string in the ByteArray:

var bytes = new air.ByteArray();
air.trace("bytes.position is initially: " + bytes.position); // 0
bytes.writeUTFBytes("Hello World!");
air.trace("bytes.position is now: " + bytes.position); // 12

Likewise, a read operation increments the position property by the number of bytes read.

var bytes = new air.ByteArray();

air.trace("bytes.position is initially: " + bytes.position); // 0
bytes.writeUTFBytes("Hello World!");
air.trace("bytes.position is now: " + bytes.position); // 12
bytes.position = 0;
air.trace("The first 6 bytes are: " + (bytes.readUTFBytes(6))); // Hello
air.trace("And the next 6 bytes are: " + (bytes.readUTFBytes(6))); // World!

Notice that you can set the position property to a specific location in the ByteArray to read or write at that offset.

The bytesAvailable and length properties
The length and bytesAvailable properties tell you how long a ByteArray is and how many bytes remain in it from the current position to the end. The following example illustrates how you can use these properties. The example writes a String of text to the ByteArray and then reads the ByteArray one byte at a time until it encounters either the character “a” or the end (bytesAvailable <= 0).
var bytes = new air.ByteArray();
var text = "Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Vivamus etc.";

bytes.writeUTFBytes(text); // write the text to the ByteArray
air.trace("The length of the ByteArray is: " + bytes.length); // 70
bytes.position = 0; // reset position
while (bytes.bytesAvailable > 0 && (bytes.readUTFBytes(1) != 'a')) {
    // read to letter a or end of bytes
}
if (bytes.position < bytes.bytesAvailable) {
    air.trace("Found the letter a; position is: " + bytes.position); // 23
    air.trace("and the number of bytes available is: " + bytes.bytesAvailable); // 47
}

The endian property
Computers can differ in how they store multibyte numbers, that is, numbers that require more than 1 byte of memory to store them. An integer, for example, can take 4 bytes, or 32 bits, of memory. Some computers store the most significant byte of the number first, in the lowest memory address, and others store the least significant byte first. This attribute of a computer, or of byte ordering, is referred to as being either big endian (most significant byte first) or little endian (least significant byte first). For example, the number 0x31323334 would be stored as follows for big endian and little endian byte ordering, where a0 represents the lowest memory address of the 4 bytes and a3 represents the highest:

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>Big Endian</th>
<th>Big Endian</th>
<th>Big Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>a1</td>
<td>a2</td>
<td>a3</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little Endian</th>
<th>Little Endian</th>
<th>Little Endian</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>a1</td>
<td>a2</td>
<td>a3</td>
</tr>
<tr>
<td>34</td>
<td>33</td>
<td>32</td>
<td>31</td>
</tr>
</tbody>
</table>

The endian property of the ByteArray class allows you to denote this byte order for multibyte numbers that you are processing. The acceptable values for this property are either "bigEndian" or "littleEndian" and the Endian class defines the constants BIG_ENDIAN and LITTLE_ENDIAN for setting the endian property with these strings.

The compress() and uncompress() methods
The compress() method allows you to compress a ByteArray in accordance with a compression algorithm that you specify as a parameter. The uncompress() method allows you to uncompress a compressed ByteArray in accordance with a compression algorithm. After calling compress() and uncompress(), the length of the byte array is set to the new length and the position property is set to the end.

The CompressionAlgorithm class defines constants that you can use to specify the compression algorithm. AIR supports both the deflate and zlib algorithms. The deflate compression algorithm is used in several compression formats, such as zlib, gzip, and some zip implementations. The zlib compressed data format is described at http://www.ietf.org/rfc/rfc1950.txt and the deflate compression algorithm is described at http://www.ietf.org/rfc/rfc1951.txt.

The following example compresses a ByteArray called bytes using the deflate algorithm:
The following example uncompresses a compressed ByteArray using the deflate algorithm:

```javascript
bytes.uncompress(CompressionAlgorithm.DEFLATE);
```

### Reading and writing objects

The `readObject()` and `writeObject()` methods read an object from and write an object to a ByteArray, encoded in serialized Action Message Format (AMF). AMF is a proprietary message protocol created by Adobe and used by various ActionScript 3.0 classes, including Netstream, NetConnection, NetStream, LocalConnection, and Shared Objects.

A one-byte type marker describes the type of the encoded data that follows. AMF uses the following 13 data types:

- `undefined-marker`
- `null-marker`
- `false-marker`
- `true-marker`
- `integer-type`
- `double-type`
- `string-type`
- `xml-doc-type`
- `date-type`
- `array-type`
- `object-type`
- `xml-type`
- `byte-array-type`

The encoded data follows the type marker unless the marker represents a single possible value, such as null or true or false, in which case nothing else is encoded.

There are two versions of AMF: AMF0 and AMF3. AMF 0 supports sending complex objects by reference and allows endpoints to restore object relationships. AMF 3 improves AMF 0 by sending object traits and strings by reference, in addition to object references, and by supporting new data types that were introduced in ActionScript 3.0. The `ByteArray.objectEncoding` property specifies the version of AMF that is used to encode the object data. The `flash.net.ObjectEncoding` class defines constants for specifying the AMF version: `ObjectEncoding.AMF0` and `ObjectEncoding.AMF3`.

The following example calls `writeObject()` to write an XML object to a ByteArray, which it then writes to the order file on the desktop. The example displays the message “Wrote order file to desktop!” in the AIR window when it is finished.

```javascript
function init() {

    <!-- define ByteArray -->
    var inBytes = new air.ByteArray();

    // add objectEncoding value and file heading to output text
    var output = "Object encoding is: " + inBytes.objectEncoding + "\norder file: \n\n";

    // define style
    #taFiles
    {
        border: 1px solid black;
        font-family: Courier, monospace;
        white-space: pre;
        width: 95%;
        height: 95%;
        overflow-y: scroll;
    }

    // define script
    <script type="text/javascript" src="AIRAliases.js"></script>
    <script type="text/javascript">
        // define ByteArray
        var inBytes = new air.ByteArray();
        // add objectEncoding value and file heading to output text
        var output = "Object encoding is: " + inBytes.objectEncoding + "\norder file: \n\n";

        function init() {
            // code goes here
        }
    </script>

    <!-- define HTML Document -->
    <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"><html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <style type="text/css">
            #taFiles
            {
                border: 1px solid black;
                font-family: Courier, monospace;
                white-space: pre;
                width: 95%;
                height: 95%;
                overflow-y: scroll;
            }
        </style>
        <script type="text/javascript" src="AIRAliases.js"></script>
        <script type="text/javascript">
            // define ByteArray
            var inBytes = new air.ByteArray();
            // add objectEncoding value and file heading to output text
            var output = "Object encoding is: " + inBytes.objectEncoding + "\norder file: \n\n";

            function init() {
                // code goes here
            }
        </script>
    </head>
    <body>
        <!-- code goes here -->
    </body>
</html>
```
readFile("order", inBytes);
inBytes.position = 0; // reset position to beginning
// read XML from ByteArray
var orderXML = inBytes.readObject();
// convert to XML Document object
var myXML = (new DOMParser()).parseFromString(orderXML, "text/xml");
document.write(myXML.getElementsByTagName("menuName")[0].childNodes[0].nodeValue + ": ");
document.write(myXML.getElementsByTagName("price")[0].childNodes[0].nodeValue + ": ");
    // burger: 3.95
    document.write(myXML.getElementsByTagName("menuName")[1].childNodes[0].nodeValue + ": ");
    document.write(myXML.getElementsByTagName("price")[1].childNodes[0].nodeValue + ": ");
    // fries: 1.45
} // end of init()

// read specified file into byte array
function readFile(fileName, data) {
    var inFile = air.File.desktopDirectory; // source folder is desktop
    inFile = inFile.resolvePath(fileName);  // name of file to read
    var inStream = new air.FileStream();
inStream.open(inFile, air.FileMode.READ);
inStream.readBytes(data, 0, data.length);
inStream.close();
}
</script>
</head>

<body onload = "init();">
    <div id="taFiles"></div>
</body>
</html>

The readObject() method reads an object in serialized AMF from a ByteArray and stores it in an object of the specified type. The following example reads the order file from the desktop into a ByteArray (inBytes) and calls readObject() to store it in orderXML, which it then converts to an XML object document, myXML, and displays the values of two item and price elements. The example also displays the value of the objectEncoding property along with a header for the contents of the order file.
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<style type="text/css">
#taFiles
{
    border: 1px solid black;
    font-family: Courier, monospace;
    white-space: pre;
    width: 95%;
    height: 95%;
    overflow-y: scroll;
}
</style>
<script type="text/javascript" src="AIRAliases.js" ></script>
<script type="text/javascript">
//define ByteArray
var inBytes = new air.ByteArray();
//add objectEncoding value and file heading to output text
var output = "Object encoding is: " + inBytes.objectEncoding + "<br/><br/>" + "order file items:" + "<br/>";

function init() {
    readFile("order", inBytes);
    inBytes.position = 0; //reset position to beginning
    // read XML from ByteArray
    var orderXML = inBytes.readObject();
    // convert to XML Document object
    var myXML = (new DOMParser()).parseFromString(orderXML, "text/xml");
    document.write(output);
    document.write(myXML.getElementsByTagName("menuName")[0].childNodes[0].nodeValue + ": ");
    document.write(myXML.getElementsByTagName("price")[0].childNodes[0].nodeValue + "<br/>"); // burger: 3.95
    document.write(myXML.getElementsByTagName("menuName")[1].childNodes[0].nodeValue + ":
    ");
    document.write(myXML.getElementsByTagName("price")[1].childNodes[0].nodeValue + "<br/>"); // dessert: 2.95

    // read XML items
    var items = myXML.getElementsByTagName("items");
    for (var i = 0; i < items.length; i++) {
        document.write(items[i].childNodes[0].nodeValue + "<br/>");
    }

    // read XML menu names
    var menuNames = myXML.getElementsByTagName("menuName");
    for (var i = 0; i < menuNames.length; i++) {
        document.write(menuNames[i].childNodes[0].nodeValue + "<br/>");
    }

    // read XML prices
    var prices = myXML.getElementsByTagName("price");
    for (var i = 0; i < prices.length; i++) {
        document.write(prices[i].childNodes[0].nodeValue + "<br/>");
    }
}
</script>
</head>
<body>
</body>
</html>
ByteArray example: Reading a .zip file

This example demonstrates how to read a simple .zip file containing several files of different types. It does so by extracting relevant data from the metadata for each file, uncompressing each file into a ByteArray and writing the file to the desktop.

The general structure of a .zip file is based on the specification by PKWARE Inc., which is maintained at http://www.pkware.com/documents/casestudies/APPNOTE.TXT. First is a file header and file data for the first file in the .zip archive, followed by a file header and file data pair for each additional file. (The structure of the file header is described later.) Next, the .zip file optionally includes a data descriptor record (usually when the output zip file was created in memory rather than saved to a disk). Next are several additional optional elements: archive decryption header, archive extra data record, central directory structure, Zip64 end of central directory record, Zip64 end of central directory locator, and end of central directory record.

The code in this example is written to only parse zip files that do not contain folders and it does not expect data descriptor records. It ignores all information following the last file data.

The format of the file header for each file is as follows:

<table>
<thead>
<tr>
<th>File Header Field</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>File header signature</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Required version</td>
<td>2 bytes</td>
</tr>
<tr>
<td>General-purpose bit flag</td>
<td>2 bytes</td>
</tr>
<tr>
<td>Compression method</td>
<td>2 bytes (8=DEFLATE; 0=UNCOMPRESSED)</td>
</tr>
<tr>
<td>Last modified file time</td>
<td>2 bytes</td>
</tr>
<tr>
<td>Last modified file date</td>
<td>2 bytes</td>
</tr>
<tr>
<td>Crc-32</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Compressed size</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Uncompressed size</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>
Following the file header is the actual file data, which can be either compressed or uncompressed, depending on the compression method flag. The flag is 0 (zero) if the file data is uncompressed, 8 if the data is compressed using the DEFLATE algorithm, or another value for other compression algorithms.

The user interface for this example consists of a label and a text area (taFiles). The application writes the following information to the text area for each file it encounters in the .zip file: the file name, the compressed size, and the uncompressed size. The following HTML page defines the user interface for the application:

```html
<html>
<head>
  <style type="text/css">
    #taFiles
    {
      border: 1px solid black;
      font-family: Courier, monospace;
      white-space: pre;
      width: 95%;
      height: 95%;
      overflow-y: scroll;
    }
  </style>
  <script type="text/javascript" src="AIRAliases.js"></script>
  <script type="text/javascript">
    // The application code goes here
  </script>
</head>
<body onload="init();">
  <div id="taFiles"></div>
</body>
</html>
```

The beginning of the program performs the following tasks:

- Defines the bytes ByteArray
  ```javascript
  var bytes = new air.ByteArray();
  ```
- Defines variables to store metadata from the file header
  ```javascript
  // variables for reading fixed portion of file header
  var fileName = new String();
  var fNameLength;
  var xfldLength;
  var offset;
  var compSize;
  var uncompSize;
  var compMethod;
  var signature;
  var output;
  ```
• Defines File (zfile) and FileStream (zStream) objects to represent the .zip file, and specifies the location of the .zip file from which the files are extracted—a file named “HelloAIR.zip” in the desktop directory.

```javascript
// File variables for accessing .zip file
var zfile = air.File.desktopDirectory.resolvePath("HelloAIR.zip");
var zStream = new air.FileStream();
```

The program code starts in the init() method, which is called as the onload event handler for the body tag.

```javascript
function init()
{
    The program begins by opening the .zip file in READ mode.

    zStream.open(zfile, air.FileMode.READ);

    It then sets the endian property of bytes to LITTLE_ENDIAN to indicate that the byte order of numeric fields has the least significant byte first.

    bytes.endian = air.Endian.LITTLE_ENDIAN;

    Next, a while() statement begins a loop that continues until the current position in the file stream is greater than or equal to the size of the file.

    while (zStream.position < zfile.size)
    {
        The first statement inside the loop reads the first 30 bytes of the file stream into the ByteArray bytes. The first 30 bytes make up the fixed-size part of the first file header.

        // read fixed metadata portion of local file header
        zStream.readBytes(bytes, 0, 30);

        Next, the code reads an integer (signature) from the first bytes of the 30-byte header. The ZIP format definition specifies that the signature for every file header is the hexadecimal value 0x04034b50; if the signature is different it means that the code has moved beyond the file portion of the .zip file and there are no more files to extract. In that case the code exits the while loop immediately rather than waiting for the end of the byte array.

        bytes.position = 0;
        signature = bytes.readInt();  // store compression method (8 == Deflate)
        // if no longer reading data files, quit
        if (signature != 0x04034b50)
        {
            break;
        }
    }
```

The next part of the code reads the header byte at offset position 8 and stores the value in the variable compMethod. This byte contains a value indicating the compression method that was used to compress this file. Several compression methods are allowed, but in practice nearly all .zip files use the DEFLATE compression algorithm. If the current file is compressed with DEFLATE compression, compMethod is 8; if the file is uncompressed, compMethod is 0.

```javascript
bytes.position = 8;
compMethod = bytes.readByte();  // store compression method (8 == Deflate)
```

Following the first 30 bytes is a variable-length portion of the header that contains the file name and, possibly, an extra field. The variable offset stores the size of this portion. The size is calculated by adding the file name length and extra field length, read from the header at offsets 26 and 28.
offset = 0;  // stores length of variable portion of metadata
bytes.position = 26;  // offset to file name length
flNameLength = bytes.readShort();  // store file name
offset += flNameLength;  // add length of file name
bytes.position = 28;  // offset to extra field length
xfldLength = bytes.readShort();
offset += xfldLength;  // add length of extra field

Next the program reads the variable-length portion of the file header for the number of bytes stored in the offset variable.

// read variable length bytes between fixed-length header and compressed file data
zStream.readBytes(bytes, 30, offset);

The program reads the file name from the variable length portion of the header and displays it in the text area along with the compressed (zipped) and uncompressed (original) sizes of the file.

bytes.position = 30;
fileName = bytes.readUTFBytes(flNameLength);  // read file name
output += fileName + '<br />';
bytes.position = 18;
compSize = bytes.readUnsignedInt();  // store size of compressed portion
output += '
\tCompressed size is: ' + compSize + '<br />';
bytes.position = 22;  // offset to uncompressed size
uncompSize = bytes.readUnsignedInt();  // store uncompressed size
output += '
\tUncompressed size is: ' + uncompSize + '<br />';

The example reads the rest of the file from the file stream into bytes for the length specified by the compressed size, overwriting the file header in the first 30 bytes. The compressed size is accurate even if the file is not compressed because in that case the compressed size is equal to the uncompressed size of the file.

// read compressed file to offset 0 of bytes; for uncompressed files
// the compressed and uncompressed size is the same
zStream.readBytes(bytes, 0, compSize);

Next, the example uncompresses the compressed file and calls the outfile() function to write it to the output file stream. It passes outfile() the file name and the byte array containing the file data.

if (compMethod == 8) // if file is compressed, uncompress
{
    bytes.uncompress(air.CompressionAlgorithm.DEFLATE);
}
outFile(fileName, bytes);  // call outFile() to write out the file

The closing braces indicate the end of the while loop and of the init() method and the application code, except for the outFile() method. Execution loops back to the beginning of the while loop and continues processing the next bytes in the .zip file—either extracting another file or ending processing of the .zip file if the last file has been processed. When all the files have been processed, the example writes the contents of the output variable to the div element taFiles to display the file information on the screen.

} // end of while loop

document.getElementById("taFiles").innerHTML = output;
} // end of init() method

The outFile() function opens an output file in WRITE mode on the desktop, giving it the name supplied by the filename parameter. It then writes the file data from the data parameter to the output file stream (outStream) and closes the file.
function outFile(fileName, data)
{
    var outFile = air.File.desktopDirectory; // dest folder is desktop
    outFile = outFile.resolvePath(fileName); // name of file to write
    var outStream = new air.FileStream();
    // open output file stream in WRITE mode
    outStream.open(outFile, air FileMode.WRITE);
    // write out the file
    outStream.writeBytes(data, 0, data.length);
    // close it
    outStream.close();
}
Chapter 25: Working with local SQL databases

Adobe® AIR® includes the capability of creating and working with local SQL databases. The runtime includes a SQL database engine with support for many standard SQL features, using the open source SQLite database system. A local SQL database can be used for storing local, persistent data. For example, it can be used for application data, application user settings, documents, or any other type of data that you want your application to save locally.

Additional online information about local SQL databases

You can find more information about working with local SQL databases from these sources:

Quick Starts (Adobe AIR Developer Connection)
- Working asynchronously with a local SQL database
- Working synchronously with a local SQL database
- Using an encrypted database

Language Reference
- SQLCollationType
- SQLColumnNameStyle
- SQLColumnSchema
- SQLConnection
- SQLError
- SQLErrorEvent
- SQLErrorOperation
- SQLErrorEvent
- SQLIndexSchema
- SQLMode
- SQLResult
- SQLSchema
- SQLSchemaResult
- SQLStatement
- SQLTableSchema
- SQLTransactionLockType
- SQLTriggerSchema
- SQLUpdateEvent
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, the runtime’s local SQL database engine provides a convenient mechanism for storing persistent, local application data, particularly if you have experience with SQL and relational databases.

Uses for local SQL databases

The AIR local SQL database functionality can be used for any purpose for which you might want to store application data on a user’s local computer. Adobe AIR includes several mechanisms for storing data locally, each of which has different advantages. The following are some possible uses for a local SQL database in your AIR application:

- For a data-oriented application (for example an address book), a database can be used to store the main application data.
- For a document-oriented application, where users create documents to save and possibly share, each document could be saved as a database file, in a user-designated location. (Note, however, that unless the database is encrypted any AIR application would be able to open the database file. Encryption is recommended for potentially sensitive documents.)
- For a network-aware application, a database can be used to store a local cache of application data, or to store data temporarily when a network connection isn’t available. You could create a mechanism for synchronizing the local database with the network data store.
- For any application, a database can be used to store individual users’ application settings, such as user options or application information like window size and position.

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 specify the location in the file system where the database file is stored. A single AIR application can access one or many separate databases (that is, separate database files). Because the runtime stores each database as a single file on the file system, you can locate your database as needed by the design of your application and file access constraints of the operating system. Each user can have a separate database file for their specific data, or a database file can be accessed by all application users on a single computer for shared data. Because the data is local to a single computer, data is not automatically shared among users on different computers. The local SQL database engine doesn’t provide any capability to execute SQL statements against a remote or server-based database.
About relational databases

A relational database is a mechanism for storing (and retrieving) data on a computer. Data is organized into tables: rows represent records or items, and columns (sometimes called “fields”) divide each record into individual values. For example, an address book application could contain a “friends” table. Each row in the table would represent a single friend stored in the database. The table’s columns would represent data such as first name, last name, birth date, and so forth. For each friend row in the table, the database stores a separate value for each column.

Relational databases are designed to store complex data, where one item is associated with or related to items of another type. In a relational database, any data that has a one-to-many relationship—where a single record can be related to multiple records of a different type—should be divided among different tables. For example, suppose you want your address book application to store multiple phone numbers for each friend; this is a one-to-many relationship. The “friends” table would contain all the personal information for each friend. A separate “phone numbers” table would contain all the phone numbers for all the friends.

In addition to storing the data about friends and phone numbers, each table would need a piece of data to keep track of the relationship between the two tables—to match individual friend records with their phone numbers. This data is known as a primary key—a unique identifier that distinguishes each row in a table from other rows in that table. The primary key can be a “natural key,” meaning it’s one of the items of data that naturally distinguishes each record in a table. In the “friends” table, if you knew that none of your friends share a birth date, you could use the birth date column as the primary key (a natural key) of the “friends” table. If there isn’t a natural key, you would create a separate primary key column such as a “friend id” —an artificial value that the application uses to distinguish between rows.

Using a primary key, you can set up relationships between multiple tables. For example, suppose the “friends” table has a column “friend id” that contains a unique number for each row (each friend). The related “phone numbers” table can be structured with two columns: one with the “friend id” of the friend to whom the phone number belongs, and one with the actual phone number. That way, no matter how many phone numbers a single friend has, they can all be stored in the “phone numbers” table and can be linked to the related friend using the “friend id” primary key. When a primary key from one table is used in a related table to specify the connection between the records, the value in the related table is known as a foreign key. Unlike many databases, the AIR local database engine does not allow you to create foreign key constraints, which are constraints that automatically check that an inserted or updated foreign key value has a corresponding row in the primary key table. Nevertheless, foreign key relationships are an important part of the structure of a relational database, and foreign keys should be used when creating relationships between tables in your database.

About SQL

Structured Query Language (SQL) is used with relational databases to manipulate and retrieve data. SQL is a descriptive language rather than a procedural language. Instead of giving the computer instructions on how it should retrieve data, a SQL statement describes the set of data you want. The database engine determines how to retrieve that data.

The SQL language has been standardized by the American National Standards Institute (ANSI). The Adobe AIR local SQL database supports most of the SQL-92 standard. For specific descriptions of the SQL language supported in Adobe AIR, see the appendix “SQL support in local databases” in the Adobe AIR Language Reference for HTML Developers.

About SQL database classes

To work with local SQL databases in JavaScript, you use instances of the following classes. (Note that you need to load the file AIRAliases.js in your HTML document in order to use the air.* aliases for these classes):
To obtain schema information describing the structure of a database, you use these classes:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>air.SQLSchemaResult</td>
<td>Serves as a container for database schema results generated by calling the SQLConnection.loadSchema() method.</td>
</tr>
<tr>
<td>air.SQLTableSchema</td>
<td>Provides information describing a single table in a database.</td>
</tr>
<tr>
<td>air.SQLViewSchema</td>
<td>Provides information describing a single view in a database.</td>
</tr>
<tr>
<td>air.SQLIndexSchema</td>
<td>Provides information describing a single column of a table or view in a database.</td>
</tr>
<tr>
<td>air.SQLTriggerSchema</td>
<td>Provides information describing a single trigger in a database.</td>
</tr>
</tbody>
</table>

The following classes provide constants that are used with the SQLConnection class:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>air.SQLMode</td>
<td>Defines a set of constants representing the possible values for the openMode parameter of the SQLConnection.open() and SQLConnection.openAsync() methods.</td>
</tr>
<tr>
<td>air.SQLColumnNameStyle</td>
<td>Defines a set of constants representing the possible values for the SQLConnection.columnNameStyle property.</td>
</tr>
<tr>
<td>air.SQLTransactionLockType</td>
<td>Defines a set of constants representing the possible values for the option parameter of the SQLConnection.begin() method.</td>
</tr>
<tr>
<td>air.SQLCollationType</td>
<td>Defines a set of constants representing the possible values for the SQLColumnSchema.defaultCollationType property and the defaultCollationType parameter of the SQLColumnSchema() constructor.</td>
</tr>
</tbody>
</table>

In addition, the following classes represent the events (and supporting constants) that you use:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>air.SQLEvent</td>
<td>Defines the events that a SQLConnection or SQLStatement instance dispatches when any of its operations execute successfully. Each operation has an associated event type constant defined in the SQLEvent class.</td>
</tr>
<tr>
<td>air.SQLErrorEvent</td>
<td>Defines the event that a SQLConnection or SQLStatement instance dispatches when any of its operations results in an error.</td>
</tr>
<tr>
<td>air.SQLUpdateEvent</td>
<td>Defines the event that a SQLConnection instances dispatches when table data in one of its connected databases changes as a result of an INSERT, UPDATE, or DELETE SQL statement being executed.</td>
</tr>
</tbody>
</table>

Finally, the following classes provide information about database operation errors:
About synchronous and asynchronous execution modes

When you’re writing code to work with a local SQL database, you specify that database operations execution in one of two execution modes: asynchronous or synchronous execution mode. In general, the code examples show how to perform each operation in both ways, so that you can use the example that’s most appropriate for your needs.

In asynchronous execution mode, you give the runtime an instruction and the runtime dispatches an event when your requested operation completes or fails. First you tell the database engine to perform an operation. The database engine does its work in the background while the application continues running. Finally, when the operation is completed (or when it fails) the database engine dispatches an event. Your code, triggered by the event, carries out subsequent operations. This approach has a significant benefit: the runtime performs the database operations in the background while the main application code continues executing. If the database operation takes a notable amount of time, the application continues to run. Most importantly, the user can continue to interact with it without the screen freezing. Nevertheless, asynchronous operation code can be more complex to write than other code. This complexity is usually in cases where multiple dependent operations must be divided up among various event listener methods.

Conceptually, it is simpler to code operations as a single sequence of steps—a set of synchronous operations—rather than a set of operations split into several event listener methods. In addition to asynchronous database operations, Adobe AIR also allows you to execute database operations synchronously. In synchronous execution mode, operations don’t run in the background. Instead they run in the same execution sequence as all other application code. You tell the database engine to perform an operation. The code then pauses at that point while the database engine does its work. When the operation completes, execution continues with the next line of your code.

Whether operations execute asynchronously or synchronously is set at the SQLConnection level. Using a single database connection, you can’t execute some operations or statements synchronously and others asynchronously. You specify whether a SQLConnection operates in synchronous or asynchronous execution mode by calling a SQLConnection method to open the database. If you call SQLConnection.open() the connection operates in synchronous execution mode, and if you call SQLConnection.openAsync() the connection operates in asynchronous execution mode. Once a SQLConnection instance is connected to a database using open() or openAsync(), it is fixed to synchronous or asynchronous execution mode unless you close and reopen the connection to the database.

Each execution mode has benefits. While most aspects of each mode are similar, there are some differences you’ll want to keep in mind when working in each mode. For more information on these topics, and suggestions for working in each mode, see “Using synchronous and asynchronous database operations” on page 269.

Creating and modifying a database

Before your application can add or retrieve data, there must be a database with tables defined in it that your application can access. Described here are the tasks of creating a database and creating the data structure within a database. While these tasks are less frequently used than data insertion and retrieval, they are necessary for most applications.
Creating a database

To create a database file, you first create a SQLConnection instance. You call its open() method to open it in synchronous execution mode, or its openAsync() method to open it in asynchronous execution mode. The open() and openAsync() methods are used to open a connection to a database. If you pass a File instance that refers to a non-existent file location for the reference parameter (the first parameter), the open() or openAsync() method creates a database file at that file location and open a connection to the newly created database.

Whether you call the open() method or the openAsync() method to create a database, the database file’s name can be any valid filename, with any filename extension. If you call the open() or openAsync() method with null for the reference parameter, a new in-memory database is created rather than a database file on disk.

The following code listing shows the process of creating a database file (a new database) using asynchronous execution mode. In this case, the database file is saved in the application storage directory, with the filename “DBSample.db”:

```javascript
// Include AIRAliases.js to use air.* shortcuts
var conn = new air.SQLConnection();
conn.addEventListener(air.SQLEvent.OPEN, openHandler);
conn.addEventListener(air.SQLErrorEvent.ERROR, errorHandler);

// The database file is in the application storage directory
var folder = air.File.applicationStorageDirectory;
var dbFile = folder.resolvePath("DBSample.db");
conn.openAsync(dbFile);

function openHandler(event)
{
    air.trace("the database was created successfully");
}

function errorHandler(event)
{
    air.trace("Error message:", event.error.message);
    air.trace("Details:", event.error.details);
}
```

**Note:** Although the File class lets you point to a specific native file path, doing so can lead to applications that will not work across platforms. For example, the path C:\Documents and Settings\joe\test.db only works on Windows. For these reasons, it is best to use the static properties of the File class, such as File.applicationDirectory, and the resolvePath() method (as shown in the previous example). For more information, see "Paths of File objects" on page 187.

To execute operations synchronously, when you open a database connection with the SQLConnection instance, call the open() method. The following example shows how to create and open a SQLConnection instance that executes its operations synchronously:
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. For more information about executing SQL statements, see “Working with SQL statements” on page 252.

The following example demonstrates creating a table named "employees" in an existing database file, using asynchronous execution mode. Note that this code assumes there is a SQLConnection instance named conn that is already instantiated and is already connected to a database.
// Include AIRAliases.js to use air.* shortcuts

// ... create and open the SQLConnection instance named conn ...

var createStmt = new air.SQLStatement();
createStmt.sqlConnection = conn;

var sql = "CREATE TABLE IF NOT EXISTS employees (" +
    " empId INTEGER PRIMARY KEY AUTOINCREMENT, " +
    " firstName TEXT, " +
    " lastName TEXT, " +
    " salary NUMERIC CHECK (salary > 0)" +
    ");
createStmt.text = sql;

createStmt.addEventListener(air.SQLEvent.RESULT, createResult);
createStmt.addEventListener(air.SQLErrorEvent.ERROR, createError);

createStmt.execute();

function createResult(event)
{
    air.trace("Table created");
}

function createError(event)
{
    air.trace("Error message:", event.error.message);
    air.trace("Details:", event.error.details);
}

The following example demonstrates how to create a table named "employees" in an existing database file, using synchronous execution mode. Note that this code assumes there is a SQLConnection instance named conn that is already instantiated and is already connected to a database.
Manipulating SQL database data

There are some common tasks that you perform when you’re working with local SQL databases. These tasks include connecting to a database, adding data to tables, and retrieving data from tables in a database. There are also several issues you’ll want to keep in mind while performing these tasks, such as working with data types and handling errors.

Note that there are also several database tasks that are things you’ll deal with less frequently, but will often need to do before you can perform these more common tasks. For example, before you can connect to a database and retrieve data from a table, you’ll need to create the database and create the table structure in the database. Those less-frequent initial setup tasks are discussed in “Creating and modifying a database” on page 246.

You can choose to perform database operations asynchronously, meaning the database engine runs in the background and notifies you when the operation succeeds or fails by dispatching an event. You can also perform these operations synchronously. In that case the database operations are performed one after another and the entire application (including updates to the screen) waits for the operations to complete before executing other code. The examples in this section demonstrate how to perform the operations both asynchronously and synchronously. For more information on working in asynchronous or synchronous execution mode, see “Using synchronous and asynchronous database operations” on page 269.

Connecting to a database

Before you can perform any database operations, first open a connection to the database file. A SQLConnection instance is used to represent a connection to one or more databases. The first database that is connected using a SQLConnection instance is known as the “main” database. This database is connected using the open() method (for synchronous execution mode) or the openAsync() method (for asynchronous execution mode).
If you open a database using the asynchronous openAsync() operation, register for the SQLConnection instance’s open event in order to know when the openAsync() operation completes. Register for the SQLConnection instance’s error event to determine if the operation fails.

The following example shows how to open an existing database file for asynchronous execution. The database file is named “DBSample.db” and is located in the user’s application storage directory.

```javascript
// Include AIRAliases.js to use air.* shortcuts

var conn = new air.SQLConnection();

conn.addEventListener(air.SQLEvent.OPEN, openHandler);
conn.addEventListener(air.SQLErrorEvent.ERROR, errorHandler);

// The database file is in the application storage directory
var folder = air.File.applicationStorageDirectory;
var dbFile = folder.resolvePath("DBSample.db");

conn.openAsync(dbFile, air.SQLMode.UPDATE);

function openHandler(event)
{
    air.trace("the database opened successfully");
}

function errorHandler(event)
{
    air.trace("Error message:", event.error.message);
    air.trace("Details:", event.error.details);
}
```

The following example shows how to open an existing database file for synchronous execution. The database file is named “DBSample.db” and is located in the user’s application storage directory.

```javascript
// Include AIRAliases.js to use air.* shortcuts

var conn = new air.SQLConnection();

// The database file is in the application storage directory
var folder = air.File.applicationStorageDirectory;
var dbFile = folder.resolvePath("DBSample.db");

try
{
    conn.open(dbFile, air.SQLMode.UPDATE);
    air.trace("the database opened successfully");
}
catch (error)
{
    air.trace("Error message:", error.message);
    air.trace("Details:", error.details);
}
Notice that in the `openAsync()` method call in the asynchronous example, and the `open()` method call in the synchronous example, the second argument is the constant `SQLMode.UPDATE`. Specifying `SQLMode.UPDATE` for the second parameter (`openMode`) causes the runtime to dispatch an error if the specified file doesn’t exist. If you pass `SQLMode.CREATE` for the `openMode` parameter (or if you leave the `openMode` parameter off), the runtime attempts to create a database file if the specified file doesn’t exist. However, if the file exists it is opened, which is the same as if you use `SQLMode.Update`. You can also specify `SQLMode.READ` for the `openMode` parameter to open an existing database in a read-only mode. In that case data can be retrieved from the database but no data can be added, deleted, or changed.

### Working with SQL statements

An individual SQL statement (a query or command) is represented in the runtime as a `SQLStatement` object. Follow these steps to create and execute a SQL statement:

**Create a SQLStatement instance.**
The `SQLStatement` object represents the SQL statement in your application.

```
var selectData = new air.SQLStatement();
```

**Specify which database the query runs against.**
To do this, set the `SQLStatement` object’s `sqlConnection` property to the `SQLConnection` instance that’s connected with the desired database.

```
// A SQLConnection named "conn" has been created previously
selectData.sqlConnection = conn;
```

**Specify the actual SQL statement.**
Create the statement text as a String and assign it to the `SQLStatement` instance’s `text` property.

```
selectData.text = "SELECT col1, col2 FROM my_table WHERE col1 = :param1";
```

**Define functions to handle the result of the execute operation (asynchronous execution mode only).**
Use the `addEventListener()` method to register functions as listeners for the `SQLStatement` instance’s `result` and `error` events.

```
// using listener methods and addEventListener()
selectData.addEventListener(air.SQLEvent.RESULT, resultHandler);
selectData.addEventListener(air.SQLErrorEvent.ERROR, errorHandler);
```

```
function resultHandler(event)
{
    // do something after the statement execution succeeds
}
```

```
function errorHandler(event)
{
    // do something after the statement execution fails
}
```

Alternatively, you can specify listener methods using a Responder object. In that case you create the Responder instance and link the listener methods to it.
// using a Responder

var selectResponder = new air.Responder(onResult, onError);

function onResult(result)
{
    // do something after the statement execution succeeds
}

function onError(error)
{
    // do something after the statement execution fails
}

If the statement text includes parameter definitions, assign values for those parameters.
To assign parameter values, use the SQLStatement instance’s parameters associative array property.

selectData.parameters[":param1"] = 25;

Execute the SQL statement.
Call the SQLStatement instance’s execute() method.

// using synchronous execution mode
// or listener methods in asynchronous execution mode
selectData.execute();

Additionally, if you’re using a Responder instead of event listeners in asynchronous execution mode, pass the
Responder instance to the execute() method.

// using a Responder in asynchronous execution mode
selectData.execute(-1, selectResponder);

For specific examples that demonstrate these steps, see the following topics:
“Retrieving data from a database” on page 256
“Inserting data” on page 262
“Changing or deleting data” on page 265

Using parameters in statements
A SQL statement parameter allows you to create a reusable SQL statement. When you use statement parameters, values
within the statement can change (such as values being added in an INSERT statement) but the basic statement text
remains unchanged. Consequently, using parameters provides performance benefits as well as making it easier to code
an application.

Understanding statement parameters
Frequently an application uses a single SQL statement multiple times in an application, with slight variation. For
example, consider an inventory-tracking application where a user can add new inventory items to the database. The
application code that adds an inventory item to the database executes a SQL INSERT statement that actually adds the
data to the database. However, each time the statement is executed there is a slight variation. Specifically, the actual
values that are inserted in the table are different because they are specific to the inventory item being added.
In cases where you have a SQL statement that’s used multiple times with different values in the statement, the best approach is to use a SQL statement that includes parameters rather than literal values in the SQL text. A parameter is a placeholder in the statement text that is replaced with an actual value each time the statement is executed. To use parameters in a SQL statement, you create the SQLStatement instance as usual. For the actual SQL statement assigned to the text property, use parameter placeholders rather than literal values. You then define the value for each parameter by setting the value of an element in the SQLStatement instance’s parameters property. The parameters property is an associative array, so you set a particular value using the following syntax:

statement.parameters[parameter_identifier] = value;

The parameter_identifier is a string if you’re using a named parameter, or an integer index if you’re using an unnamed parameter.

**Using named parameters**

A parameter can be a named parameter. A named parameter has a specific name that the database uses to match the parameter value to its placeholder location in the statement text. A parameter name consists of the “:” or “@” character followed by a name, as in the following examples:

:name

@firstName

The following code listing demonstrates the use of named parameters:

```javascript
var sql = 
  "INSERT INTO inventoryItems (name, productCode)" + 
  "VALUES (:name, :productCode)";

var addItemStmt = new air.SQLStatement();
addItemStmt.sqlConnection = conn;
addItemStmt.text = sql;

// set parameter values
addItemStmt.parameters[":name"] = "Item name";
addItemStmt.parameters[":productCode"] = "12345";

addItemStmt.execute();
```

**Using unnamed parameters**

As an alternative to using named parameters, you can also use unnamed parameters. To use an unnamed parameter you denote a parameter in a SQL statement using a “?” character. Each parameter is assigned a numeric index, according to the order of the parameters in the statement, starting with index 0 for the first parameter. The following example demonstrates a version of the previous example, using unnamed parameters:

```javascript
var sql = 
  "INSERT INTO inventoryItems (name, productCode)" + 
  "VALUES (?, ?)";

var addItemStmt = new air.SQLStatement();
addItemStmt.sqlConnection = conn;
addItemStmt.text = sql;

// set parameter values
addItemStmt.parameters[0] = "Item name";
addItemStmt.parameters[1] = "12345";

addItemStmt.execute();
```
Benefits of using parameters

Using parameters in a SQL statement provides several benefits:

**Better performance** A SQLStatement instance that uses parameters can execute more efficiently compared to one that dynamically creates the SQL text each time it executes. The performance improvement is because the statement is prepared a single time and can then be executed multiple times using different parameter values, without needing to recompile the SQL statement.

**Explicit data typing** Parameters are used to allow for typed substitution of values that are unknown at the time the SQL statement is constructed. The use of parameters is the only way to guarantee the storage class for a value passed in to the database. When parameters are not used, the runtime attempts to convert all values from their text representation to a storage class based on the associated column’s type affinity. For more information on storage classes and column affinity, see the section "Data type support" in the appendix "SQL support in local databases" in the Adobe AIR Language Reference for HTML Developers.

**Greater security** The use of parameters helps prevent a malicious technique known as a SQL injection attack. In a SQL injection attack, a user enters SQL code in a user-accessible location (for example, a data entry field). If application code constructs a SQL statement by directly concatenating user input into the SQL text, the user-entered SQL code is executed against the database. The following listing shows an example of concatenating user input into SQL text. Do not use this technique:

```javascript
// assume the variables "username" and "password"
// contain user-entered data

var sql =
    "SELECT userId " +
    "FROM users " +
    "WHERE username = " + username + "'" +
    " AND password = " + password + ";";

var statement = new air.SQLStatement();
statement.text = sql;

Using statement parameters instead of concatenating user-entered values into a statement’s text prevents a SQL injection attack. SQL injection can’t happen because the parameter values are treated explicitly as substituted values, rather than becoming part of the literal statement text. The following is the recommended alternative to the previous listing:

```javascript
// assume the variables "username" and "password"
// contain user-entered data

var sql =
    "SELECT userId " +
    "FROM users " +
    "WHERE username = :username " +
    " AND password = :password";

var statement = new air.SQLStatement();
statement.text = sql;

// set parameter values
statement.parameters[":username"] = username;
statement.parameters[":password"] = password;
```
Retrieving data from a database

Retrieving data from a database involves two steps. First, you execute a SQL `SELECT` statement, describing the set of data you want from the database. Next, you access the retrieved data and display or manipulate it as needed by your application.

Executing a `SELECT` statement

To retrieve existing data from a database, you use a `SQLStatement` instance. Assign the appropriate SQL `SELECT` statement to the instance’s `text` property, then call its `execute()` method.

For details on the syntax of the `SELECT` statement, see the appendix "SQL support in local databases" in the Adobe AIR Language Reference for HTML Developers.

The following example demonstrates executing a `SELECT` statement to retrieve data from a table named “products,” using asynchronous execution mode:

```javascript
// Include AIRAliases.js to use air.* shortcuts
var selectStmt = new air.SQLStatement();

// A SQLConnection named "conn" has been created previously
selectStmt.sqlConnection = conn;

selectStmt.text = "SELECT itemId, itemName, price FROM products";
selectStmt.addEventListener(air.SQLEvent.RESULT, resultHandler);
selectStmt.addEventListener(air.SQLErrorEvent.ERROR, errorHandler);
selectStmt.execute();

function resultHandler(event){
var result = selectStmt.getResult();
var numResults = result.data.length;
for (i = 0; i < numResults; i++) {
    var row = result.data[i];
    var output = "itemId: " + row.itemId;
    output += "; itemName: " + row.itemName;
    output += "; price: " + row.price;
    air.trace(output);
}
}

function errorHandler(event){
    // Information about the error is available in the
    // event.error property, which is an instance of
    // the SQLError class.
}
```

The following example demonstrates executing a `SELECT` statement to retrieve data from a table named “products,” using synchronous execution mode:
DEVELOPING ADOBE AIR 1.5 APPLICATIONS WITH HTML AND AJAX

Working with local SQL databases

// Include AIRAliases.js to use air.* shortcuts
var selectStmt = new air.SQLStatement();

// A SQLConnection named "conn" has been created previously
selectStmt.sqlConnection = conn;

selectStmt.text = "SELECT itemId, itemName, price FROM products";

try
{
    selectStmt.execute();
    var result = selectStmt.getResult();

    var numResults = result.data.length;
    for (i = 0; i < numResults; i++)
    {
        var row = result.data[i];
        var output = "itemId: " + row.itemId;
        output += "; itemName: " + row.itemName;
        output += "; price: " + row.price;
        air.trace(output);
    }
}

// Information about the error is available in the
// error variable, which is an instance of
// the SQLError class.

In asynchronous execution mode, when the statement finishes executing, the SQLStatement instance dispatches a result event (SQLEvent.RESULT) indicating that the statement was run successfully. Alternatively, if a Responder object is passed as an argument in the execute() call, the Responder object’s result handler function is called. In synchronous execution mode, execution pauses until the execute() operation completes, then continues on the next line of code.

Accessing SELECT statement result data

Once the SELECT statement has finished executing, the next step is to access the data that was retrieved. You retrieve the result data from executing a SELECT statement by calling the SQLStatement object’s getResult() method:

var result = selectStmt.getResult();

The getResult() method returns a SQLResult object. The SQLResult object’s data property is an Array containing the results of the SELECT statement:

var numResults = result.data.length;
for (var i = 0; i < numResults; i++)
{
    // row is an Object representing one row of result data
    var row = result.data[i];
}
Each row of data in the `SELECT` result set becomes an Object instance contained in the `data` Array. That object has properties whose names match the result set's column names. The properties contain the values from the result set's columns. For example, suppose a `SELECT` statement specifies a result set with three columns named “itemId,” “itemName,” and “price.” For each row in the result set, an Object instance is created with properties named `itemId`, `itemName`, and `price`. Those properties contain the values from their respective columns.

The following code listing defines a `SQLStatement` instance whose text is a `SELECT` statement. The statement retrieves rows containing the `firstName` and `lastName` column values of all the rows of a table named `employees`. This example uses asynchronous execution mode. When the execution completes, the `selectResult()` method is called, and the resulting rows of data are accessed using `SQLStatement.getResult()` and displayed using the `trace()` method. Note that this listing assumes there is a `SQLConnection` instance named `conn` that has already been instantiated and is already connected to the database. It also assumes that the “employees” table has already been created and populated with data.

```javascript
// Include AIRAliases.js to use air.* shortcuts

// ... create and open the SQLConnection instance named conn ...

// create the SQL statement
var selectStmt = new air.SQLStatement();
selectStmt.sqlConnection = conn;

// define the SQL text
var sql =
   "SELECT firstName, lastName " +
   "FROM employees";
selectStmt.text = sql;

// register listeners for the result and error events
selectStmt.addEventListener(air.SQLEvent.RESULT, selectResult);
selectStmt.addEventListener(air.SQLErrorEvent.ERROR, selectError);

// execute the statement
selectStmt.execute();

function selectResult(event) {

```
// access the result data
var result = selectStmt.getResult();

var numRows = result.data.length;
for (i = 0; i < numRows; i++)
{
    var output = ""
    for (columnName in result.data[i])
    {
        output += columnName + ": " + result.data[i][columnName] + "; ";
    }
    air.trace("row[" + i.toString() + "]	", output);
}

function selectError(event)
{
    air.trace("Error message:", event.error.message);
    air.trace("Details:", event.error.details);
}

The following code listing demonstrates the same techniques as the preceding one, but uses synchronous execution mode. The example defines a SQLStatement instance whose text is a SELECT statement. The statement retrieves rows containing the firstName and lastName column values of all the rows of a table named employees. The resulting rows of data are accessed using SQLStatement.getResult() and displayed using the trace() method. Note that this listing assumes there is a SQLConnection instance named conn that has already been instantiated and is already connected to the database. It also assumes that the “employees” table has already been created and populated with data.
// Include AIRAliases.js to use air.* shortcuts

// ... create and open the SQLConnection instance named conn ...

// create the SQL statement
var selectStmt = new air.SQLStatement();
selectStmt.sqlConnection = conn;

// define the SQL text
var sql =
    "SELECT firstName, lastName " +
    "FROM employees";
selectStmt.text = sql;

try
{
    // execute the statement
    selectStmt.execute();

    // access the result data
    var result = selectStmt.getResult();
    var numRows = result.data.length;
    for (i = 0; i < numRows; i++)
    {
        var output = "";
        for (columnName in result.data[i])
        {
            output += columnName + ": " + result.data[i][columnName] + "; ";
        }
        air.trace("row[" + i.toString() + "]	", output);
    }
}
catch (error)
{
    air.trace("Error message:", error.message);
    air.trace("Details:", error.details);
}

**Defining the data type of SELECT result data**

By default, each row returned by a SELECT statement is created as an Object instance with properties named for the result set's column names and with the value of each column as the value of its associated property. However, before executing a SQL SELECT statement, you can set the itemClass property of the SQLStatement instance to a class. By setting the itemClass property, each row returned by the SELECT statement is created as an instance of the designated class. The runtime assigns result column values to property values by matching the column names in the SELECT result set to the names of the properties in the itemClass class.

Any class assigned as an itemClass property value must have a constructor that does not require any parameters. In addition, the class must have a single property for each column returned by the SELECT statement. It is considered an error if a column in the SELECT list does not have a matching property name in the itemClass class.
Retrieving SELECT results in parts

By default, a SELECT statement execution retrieves all the rows of the result set at one time. Once the statement completes, you usually process the retrieved data in some way, such as creating objects or displaying the data on the screen. If the statement returns a large number of rows, processing all the data at once can be demanding for the computer, which in turn will cause the user interface to not redraw itself.

You can improve the perceived performance of your application by instructing the runtime to return a specific number of result rows at a time. Doing so causes the initial result data to return more quickly. It also allows you to divide the result rows into sets, so that the user interface is updated after each set of rows is processed. Note that it’s only practical to use this technique in asynchronous execution mode.

To retrieve SELECT results in parts, specify a value for the SQLStatement.execute() method’s first parameter (the prefetch parameter). The prefetch parameter indicates the number of rows to retrieve the first time the statement executes. When you call a SQLStatement instance’s execute() method, specify a prefetch parameter value and only that many rows are retrieved:

```javascript
// Include AIRAliases.js to use air.* shortcuts
var stmt = new air.SQLStatement();
stmt.sqlConnection = conn;
stmt.text = "SELECT ...";

stmt.addEventListener(air.SQLEvent.RESULT, selectResult);

stmt.execute(20); // only the first 20 rows (or fewer) are returned
```

The statement dispatches the result event, indicating that the first set of result rows is available. The resulting SQLResult instance’s data property contains the rows of data, and its complete property indicates whether there are additional result rows to retrieve. To retrieve additional result rows, call the SQLStatement instance’s next() method. Like the execute() method, the next() method’s first parameter is used to indicate how many rows to retrieve the next time the result event is dispatched.

```javascript
function selectResult(event)
{
    var result = stmt.getResult();
    if (result.data != null)
    {
        // ... loop through the rows or perform other processing ...

        if (!result.complete)
        {
            stmt.next(20); // retrieve the next 20 rows
        }
        else
        {
            stmt.removeEventListener(air.SQLEvent.RESULT, selectResult);
        }
    }
}
```

The SQLStatement dispatches a result event each time the next() method returns a subsequent set of result rows. Consequently, the same listener function can be used to continue processing results (from next() calls) until all the rows are retrieved.

For more information, see the language reference descriptions for the SQLStatement.execute() method (the prefetch parameter description) and the SQLStatement.next() method.
Inserting data

Retrieving data from a database involves executing a SQL **INSERT** statement. Once the statement has finished executing, you can access the primary key for the newly inserted row if the key was generated by the database.

**Executing an INSERT statement**

To add data to a table in a database, you create and execute a SQLStatement instance whose text is a SQL **INSERT** statement.

The following example uses a SQLStatement instance to add a row of data to the already-existing employees table. This example demonstrates inserting data using asynchronous execution mode. Note that this listing assumes that there is a SQLConnection instance named `conn` that has already been instantiated and is already connected to a database. It also assumes that the "employees" table has already been created.

```javascript
// Include AIRAliases.js to use air.* shortcuts

// ... create and open the SQLConnection instance named conn ...

// create the SQL statement
var insertStmt = new air.SQLStatement();
insertStmt.sqlConnection = conn;

// define the SQL text
var sql =
    "INSERT INTO employees (firstName, lastName, salary) " +
    "VALUES ('Bob', 'Smith', 8000)";
insertStmt.text = sql;

// register listeners for the result and failure (status) events
insertStmt.addEventListener(airSQLEvent.RESULT, insertResult);
insertStmt.addEventListener(airSQLErrorEvent.ERROR, insertError);

// execute the statement
insertStmt.execute();

function insertResult(event)
{
    air.trace("INSERT statement succeeded");
}

function insertError(event)
{
    air.trace("Error message:", event.error.message);
    air.trace("Details:", event.error.details);
}
```

The following example adds a row of data to the already-existing employees table, using synchronous execution mode. Note that this listing assumes that there is a SQLConnection instance named `conn` that has already been instantiated and is already connected to a database. It also assumes that the "employees" table has already been created.
Retrieving a database-generated primary key of an inserted row

Often after inserting a row of data into a table, your code needs to know a database-generated primary key or row identifier value for the newly inserted row. For example, once you insert a row in one table, you might want to add rows in a related table. In that case you would want to insert the primary key value as a foreign key in the related table.

The primary key of a newly inserted row can be retrieved using the SQLResult object generated by the statement execution. This is the same object that’s used to access result data after a SELECT statement is executed. As with any SQL statement, when the execution of an INSERT statement completes the runtime creates a SQLResult instance. You access the SQLResult instance by calling the SQLStatement object’s getResult() method if you’re using an event listener or if you’re using synchronous execution mode. Alternatively, if you’re using asynchronous execution mode and you pass a Responder instance to the execute() call, the SQLResult instance is passed as an argument to the result handler function. In any case, the SQLResult instance has a property, lastInsertRowID, that contains the row identifier of the most-recently inserted row if the executed SQL statement is an INSERT statement.

The following example demonstrates accessing the primary key of an inserted row in asynchronous execution mode:
insertStmt.text = "INSERT INTO ...";

insertStmt.addEventListener(air.SQLEvent.RESULT, resultHandler);

insertStmt.execute();

function resultHandler(event)
{
    // get the primary key
    var result = insertStmt.getResult();

    var primaryKey = result.lastInsertRowID;
    // do something with the primary key
}

The following example demonstrates accessing the primary key of an inserted row in synchronous execution mode:

try
{
    insertStmt.execute();

    // get the primary key
    var result = insertStmt.getResult();

    var primaryKey = result.lastInsertRowID;
    // do something with the primary key
}

catch (error)
{
    // respond to the error
}

Note that the row identifier may or may not be the value of the column that is designated as the primary key column in the table definition, according to the following 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. That generated value is the value of the lastInsertRowID property.

- The value is always the row identifier of the most-recently inserted row. If an INSERT statement causes a trigger to fire which in turn inserts a row, the lastInsertRowID property contains the row identifier of the last row inserted by the trigger rather than the row created by the INSERT statement. 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.
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 for HTML Developers.

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. 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 for HTML Developers.

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. This is true regardless of which attached database the statement runs on.

Alternatively, you can also create multiple SQLConnection instances in an application, each of which is connected to one or multiple databases. However, if you do use multiple connections to the same database keep in mind that a database transaction isn’t shared across SQLConnection instances. Consequently, if you connect to the same database file using multiple SQLConnection instances, you can’t rely on both connections’ data changes being applied in the expected manner. For example, if two UPDATE or DELETE statements are run against the same database through different SQLConnection instances, and an application error occurs after one operation takes place, the database data could be left in an intermediate state that would not be reversible and might affect the integrity of the database (and consequently the application).

Handling database errors

In general, database error handling is like other runtime error handling. You should write code that is prepared for errors that may occur, and respond to the errors rather than leave it up to the runtime to do so. In a general sense, the possible database errors can be divided into three categories: connection errors, SQL syntax errors, and constraint errors.
Connection errors
Most database errors are connection errors, and they can occur during any operation. Although there are strategies for preventing connection errors, there is rarely a simple way to gracefully recover from a connection error if the database is a critical part of your application.

Most connection errors have to do with how the runtime interacts with the operating system, the file system, and the database file. For example, a connection error occurs if the user doesn’t have permission to create a database file in a particular location on the file system. The following strategies help to prevent connection errors:

**Use user-specific database files** Rather than using a single database file for all users who use the application on a single computer, give each user their own database file. The file should be located in a directory that’s associated with the user’s account. For example, it could be in the application’s storage directory, the user’s documents folder, the user’s desktop, and so forth.

**Consider different user types** Test your application with different types of user accounts, on different operating systems. Don’t assume that the user has administrator permission on the computer. Also, don’t assume that the individual who installed the application is the user who’s running the application.

**Consider various file locations** If you allow a user to specify where to save a database file or select a file to open, consider the possible file locations that the users might use. In addition, consider defining limits on where users can store (or from where they can open) database files. For example, you might only allow users to open files that are within their user account’s storage location.

If a connection error occurs, it most likely happens on the first attempt to create or open the database. This means that the user is unable to do any database-related operations in the application. For certain types of errors, such as read-only or permission errors, one possible recovery technique is to copy the database file to a different location. The application can copy the database file to a different location where the user does have permission to create and write to files, and use that location instead.

Syntax errors
A syntax error occurs when a SQL statement is incorrectly formed, and the application attempts to execute the statement. Because local database SQL statements are created as strings, compile-time SQL syntax checking is not possible. All SQL statements must be executed to check their syntax. Use the following strategies to prevent SQL syntax errors:

**Test all SQL statements thoroughly** If possible, while developing your application test your SQL statements separately before encoding them as statement text in the application code. In addition, use a code-testing approach such as unit testing to create a set of tests that exercise every possible option and variation in the code.

**Use statement parameters and avoid concatenating (dynamically generating) SQL** Using parameters, and avoiding dynamically built SQL statements, means that the same SQL statement text is used each time a statement is executed. Consequently, it’s much easier to test your statements and limit the possible variation. If you must dynamically generate a SQL statement, keep the dynamic parts of the statement to a minimum. Also, carefully validate any user input to make sure it won’t cause syntax errors.

To recover from a syntax error, an application would need complex logic to be able to examine a SQL statement and correct its syntax. By following the previous guidelines for preventing syntax errors, your code can identify any potential run-time sources of SQL syntax errors (such as user input used in a statement). To recover from a syntax error, provide guidance to the user. Indicate what to correct to make the statement execute properly.
Constraint errors

Constraint errors occur when an `INSERT` or `UPDATE` statement attempts to add data to a column. The error happens if the new data violates one of the defined constraints for the table or column. The set of possible constraints includes:

- **Unique constraint**: Indicates that across all the rows in a table, there cannot be duplicate values in one column. Alternatively, when multiple columns are combined in a unique constraint, the combination of values in those columns must not be duplicated. In other words, in terms of the specified unique column or columns, each row must be distinct.

- **Primary key constraint**: In terms of the data that a constraint allows and doesn’t allow, a primary key constraint is identical to a unique constraint.

- **Not null constraint**: Specifies that a single column cannot store a `NULL` value and consequently that in every row, that column must have a value.

- **Check constraint**: Allows you to specify an arbitrary constraint on one or more tables. A common check constraint is a rule that define that a column’s value must be within certain bounds (for example, that a numeric column’s value must be larger than 0). Another common type of check constraint specifies relationships between column values (for example, that a column’s value must be different from the value of another column in the same row).

- **Data type (column affinity) constraint**: The runtime enforces the data type of columns’ values, and an error occurs if an attempt is made to store a value of the incorrect type in a column. However, in many conditions values are converted to match the column’s declared data type. See “Working with database data types” on page 268 for more information.

The runtime does not enforce constraints on foreign key values. In other words, foreign key values aren’t required to match an existing primary key value.

In addition to the predefined constraint types, the runtime SQL engine supports the use of triggers. A trigger is like an event handler. It is a predefined set of instructions that are carried out when a certain action happens. For example, a trigger could be defined that runs when data is inserted into or deleted from a particular table. One possible use of a trigger is to examine data changes and cause an error to occur if specified conditions aren’t met. Consequently, a trigger can serve the same purpose as a constraint, and the strategies for preventing and recovering from constraint errors also apply to trigger-generated errors. However, the error id for trigger-generated errors is different from the error id for constraint errors.

The set of constraints that apply to a particular table is determined while you’re designing an application. Consciously designing constraints makes it easier to design your application to prevent and recover from constraint errors. However, constraint errors are difficult to systematically predict and prevent. Prediction is difficult because constraint errors don’t appear until application data is added. Constraint errors occur with data that is added to a database after it’s created. These errors are often a result of the relationship between new data and data that already exists in the database. The following strategies can help you avoid many constraint errors:

- **Carefully plan database structure and constraints**: The purpose of constraints is to enforce application rules and help protect the integrity of the database’s data. When you’re planning your application, consider how to structure your database to support your application. As part of that process, identify rules for your data, such as whether certain values are required, whether a value has a default, whether duplicate values are allowed, and so forth. Those rules guide you in defining database constraints.

- **Explicitly specify column names**: An `INSERT` statement can be written without explicitly specifying the columns into which values are to be inserted, but doing so is an unnecessary risk. By explicitly naming the columns into which values are to be inserted, you can allow for automatically generated values, columns with default values, and columns that allow `NULL` values. In addition, by doing so you can ensure that all `NOT NULL` columns have an explicit value inserted.
Use default values Whenever you specify a **not null** constraint for a column, if at all possible specify a default value in the column definition. Application code can also provide default values. For example, your code can check if a String variable is `null` and assign it a value before using it to set a statement parameter value.

Validate user-entered data Check user-entered data ahead of time to make sure that it obeys limits specified by constraints, especially in the case of **not null** and **check** constraints. Naturally, a **unique** constraint is more difficult to check for because doing so would require executing a **select** query to determine whether the data is unique.

Use triggers You can write a trigger that validates (and possibly replaces) inserted data or takes other actions to correct invalid data. This validation and correction can prevent a constraint error from occurring.

In many ways constraint errors are more difficult to prevent than other types of errors. Fortunately, there are several strategies to recover from constraint errors in ways that don’t make the application unstable or unusable:

Use conflict algorithms When you define a constraint on a column, and when you create an **insert** or **update** statement, you have the option of specifying a conflict algorithm. A conflict algorithm defines the action the database takes when a constraint violation occurs. There are several possible actions the database engine can take. The database engine can end a single statement or a whole transaction. It can ignore the error. It can even remove old data and replace it with the data that the code is attempting to store.

For more information see the section “**ON CONFLICT (conflict algorithms)**” in the appendix “**SQL support in local databases**” in the Adobe AIR Language Reference for HTML Developers.

Provide corrective feedback The set of constraints that can affect a particular SQL command can be identified ahead of time. Consequently, you can anticipate constraint errors that a statement could cause. With that knowledge, you can build application logic to respond to a constraint error. For example, suppose an application includes a data entry form for entering new products. If the product name column in the database is defined with a **unique** constraint, the action of inserting a new product row in the database could cause a constraint error. Consequently, the application is designed to anticipate a constraint error. When the error happens, the application alerts the user, indicating that the specified product name is already in use and asking the user to choose a different name. Another possible response is to allow the user to view information about the other product with the same name.

Working with database data types

When a table is created in a database, the SQL statement for creating the table defines the affinity, or data type, for each column in the table. Although affinity declarations can be omitted, it’s a good idea to explicitly declare column affinity in your **create table** SQL statements.

As a general rule, any object that you store in a database using an **insert** statement is returned as an instance of the same data type when you execute a **select** statement. However, the data type of the retrieved value can be different depending on the affinity of the database column in which the value is stored. When a value is stored in a column, if its data type doesn’t match the column’s affinity, the database attempts to convert the value to match the column’s affinity. For example, if a database column is declared with **numeric** affinity, the database attempts to convert inserted data into a numeric storage class (**integer** or **real**) before storing the data. The database throws an error if the data can’t be converted. According to this rule, if the String “12345” is inserted into a **numeric** column, the database automatically converts it to the integer value 12345 before storing it in the database. When it’s retrieved with a **select** statement, the value is returned as an instance of a numeric data type (such as Number) rather than as a String instance.

The best way to avoid undesirable data type conversion is to follow two rules. First, define each column with the affinity that matches the type of data that it is intended to store. Next, only insert values whose data type matches the defined affinity. Following these rules provides two benefits. When you insert the data it isn’t converted unexpectedly (possibly losing its intended meaning as a result). In addition, when you retrieve the data it is returned with its original data type.
For more information about the available column affinity types and using data types in SQL statements, see the section “Data type support” in the appendix “SQL support in local databases” in the Adobe AIR Language Reference for HTML Developers.

Using synchronous and asynchronous database operations

Previous sections have described common database operations such as retrieving, inserting, updating, and deleting data, as well as creating a database file and tables and other objects within a database. The examples have demonstrated how to perform these operations both asynchronously and synchronously.

As a reminder, in asynchronous execution mode, you instruct the database engine to perform an operation. The database engine then works in the background while the application keeps running. When the operation finishes the database engine dispatches an event to alert you to that fact. The key benefit of asynchronous execution is that the runtime performs the database operations in the background while the main application code continues executing. This is especially valuable when the operation takes a notable amount of time to run.

On the other hand, in synchronous execution mode operations don’t run in the background. You tell the database engine to perform an operation. The code pauses at that point while the database engine does its work. When the operation completes, execution continues with the next line of your code.

A single database connection can’t execute some operations or statements synchronously and others asynchronously. You specify whether a SQLConnection operates in synchronous or asynchronous when you open the connection to the database. If you call SQLConnection.open() the connection operates in synchronous execution mode, and if you call SQLConnection.openAsync() the connection operates in asynchronous execution mode. Once a SQLConnection instance is connected to a database using open() or openAsync(), it is fixed to synchronous or asynchronous execution.

Using synchronous database operations

There is little difference in the actual code that you use to execute and respond to operations when using synchronous execution, compared to the code for asynchronous execution mode. The key differences between the two approaches fall into two areas. The first is executing an operation that depends on another operation (such as SELECT result rows or the primary key of the row added by an INSERT statement). The second area of difference is in handling errors.

Writing code for synchronous operations

The key difference between synchronous and asynchronous execution is that in synchronous mode you write the code as a single series of steps. In contrast, in asynchronous code you register event listeners and often divide operations among listener methods. When a database is connected in synchronous execution mode, you can execute a series of database operations in succession within a single code block. The following example demonstrates this technique:
// Include AIRAliases.js to use air.* shortcuts

var conn = new air.SQLConnection();

// The database file is in the application storage directory
var folder = File.applicationStorageDirectory;
var dbFile = folder.resolvePath("DBSample.db");

// open the database
conn.open(dbFile, air.OpenMode.UPDATE);

// start a transaction
conn.begin();

// add the customer record to the database
var insertCustomer = new air.SQLStatement();
insertCustomer.sqlConnection = conn;
insertCustomer.text =
   "INSERT INTO customers (firstName, lastName) " +
   "VALUES ('Bob', 'Jones')";
insertCustomer.execute();

var customerId = insertCustomer.getResult().lastInsertRowID;

// add a related phone number record for the customer
var insertPhoneNumber = new air.SQLStatement();
insertPhoneNumber.sqlConnection = conn;
insertPhoneNumber.text =
   "INSERT INTO customerPhoneNumbers (customerId, number) " +
   "VALUES (:customerId, '800-555-1234')";
insertPhoneNumber.parameters[":customerId"] = customerId;
insertPhoneNumber.execute();

// commit the transaction
conn.commit();

As you can see, you call the same methods to perform database operations whether you’re using synchronous or asynchronous execution. The key differences between the two approaches are executing an operation that depends on another operation and handling errors.

**Executing an operation that depends on another operation**

When you’re using synchronous execution mode, you don’t need to write code that listens for an event to determine when an operation completes. Instead, you can assume that if an operation in one line of code completes successfully, execution continues with the next line of code. Consequently, to perform an operation that depends on the success of another operation, simply write the dependent code immediately following the operation on which it depends. For instance, to code an application to begin a transaction, execute an INSERT statement, retrieve the primary key of the inserted row, insert that primary key into another row of a different table, and finally commit the transaction, the code can all be written as a series of statements. The following example demonstrates these operations:
// Include AIRAliases.js to use air.* shortcuts

var conn = new air.SQLConnection();

// The database file is in the application storage directory
var folder = File.applicationStorageDirectory;
var dbFile = folder.resolvePath("DSample.db");

// open the database
conn.open(dbFile, air.OpenMode.UPDATE);

// start a transaction
conn.begin();

// add the customer record to the database
var insertCustomer = new air.SQLStatement();
insertCustomer.sqlConnection = conn;
insertCustomer.text =
    "INSERT INTO customers (firstName, lastName) " +
    "VALUES ('Bob', 'Jones');"
insertCustomer.execute();

var customerId = insertCustomer.getResult().lastInsertRowID;

// add a related phone number record for the customer
var insertPhoneNumber = new air.SQLStatement();
insertPhoneNumber.sqlConnection = conn;
insertPhoneNumber.text =
    "INSERT INTO customerPhoneNumbers (customerId, number) " +
    "VALUES (:customerId, '800-555-1234');"
insertPhoneNumber.parameters[":customerId"] = customerId;
insertPhoneNumber.execute();

// commit the transaction
conn.commit();

Handling errors with synchronous execution

In synchronous execution mode, you don’t listen for an error event to determine that an operation has failed. Instead, you surround any code that could trigger errors in a set of try..catch..finally code blocks. You wrap the error-throwing code in the try block. Write the actions to perform in response to each type of error in separate catch blocks. Place any code that you want to always execute regardless of success or failure (for example, closing a database connection that’s no longer needed) in a finally block. The following example demonstrates using try..catch..finally blocks for error handling. It builds on the previous example by adding error handling code:
// Include AIRAliases.js to use air.* shortcuts

var conn = new air.SQLConnection();

// The database file is in the application storage directory
var folder = File.applicationStorageDirectory;
var dbFile = folder.resolvePath("DBSample.db");

// open the database
conn.open(dbFile, air.SQLMode.UPDATE);

// start a transaction
conn.begin();

try {
    // add the customer record to the database
    var insertCustomer = new air.SQLStatement();
    insertCustomer.sqlConnection = conn;
    insertCustomer.text = "INSERT INTO customers (firstName, lastName) VALUES ('Bob', 'Jones')";
    insertCustomer.execute();
    var customerId = insertCustomer.getResult().lastInsertRowID;

    // add a related phone number record for the customer
    var insertPhoneNumber = new air.SQLStatement();
    insertPhoneNumber.sqlConnection = conn;
    insertPhoneNumber.text = "INSERT INTO customerPhoneNumbers (customerId, number) VALUES (:customerId, '800-555-1234')";
    insertPhoneNumber.parameters[":customerId"] = customerId;
    insertPhoneNumber.execute();

    // if we've gotten to this point without errors, commit the transaction
    conn.commit();
} catch (error) {
    // rollback the transaction
    conn.rollback();
}

Understanding the asynchronous execution model

One common concern about using asynchronous execution mode is the assumption that you can’t start executing a SQLStatement instance if another SQLStatement is currently executing against the same database connection. In fact, this assumption isn’t correct. While a SQLStatement instance is executing you can’t change the text property of the statement. However, if you use a separate SQLStatement instance for each different SQL statement that you want to execute, you can call the execute() method of a SQLStatement while another SQLStatement instance is still executing, without causing an error.
Internally, when you’re executing database operations using asynchronous execution mode, each database connection (each SQLConnection instance) has its own queue or list of operations that it is instructed to perform. The runtime executes each operation in sequence, in the order they are added to the queue. When you create a SQLStatement instance and call its `execute()` method, that statement execution operation is added to the queue for the connection. If no operation is currently executing on that SQLConnection instance, the statement begins executing in the background. Suppose that within the same block of code you create another SQLStatement instance and also call that method’s `execute()` method. That second statement execution operation is added to the queue behind the first statement. As soon as the first statement finishes executing, the runtime moves to the next operation in the queue. The processing of subsequent operations in the queue happens in the background, even while the result event for the first operation is being dispatched in the main application code. The following code demonstrates this technique:

```javascript
// Using asynchronous execution mode
var stmt1 = new air.SQLStatement();
stmt1.sqlConnection = conn;

// ... Set statement text and parameters, and register event listeners ...
stmt1.execute();

// At this point stmt1's execute() operation is added to conn's execution queue.

var stmt2 = new air.SQLStatement();
stmt2.sqlConnection = conn;

// ... Set statement text and parameters, and register event listeners ...
stmt2.execute();

// At this point stmt2's execute() operation is added to conn's execution queue.
// When stmt1 finishes executing, stmt2 will immediately begin executing
// in the background.
```

There is an important side effect of the database automatically executing subsequent queued statements. If a statement depends on the outcome of another operation, you can’t add the statement to the queue (in other words, you can’t call its `execute()` method) until the first operation completes. This is because once you’ve called the second statement’s `execute()` method, you can’t change the statement’s `text` or `parameters` properties. In that case you must wait for the event indicating that the first operation completes before starting the next operation. For example, if you want to execute a statement in the context of a transaction, the statement execution depends on the operation of opening the transaction. After calling the `SQLConnection.begin()` method to open the transaction, you need to wait for the SQLConnection instance to dispatch its `begin` event. Only then can you call the SQLStatement instance’s `execute()` method. In this example the simplest way to organize the application to ensure that the operations are executed properly is to create a method that’s registered as a listener for the `begin` event. The code to call the `SQLStatement.execute()` method is placed within that listener method.
Using encryption with SQL databases

All Adobe AIR applications share the same local database engine. Consequently, any AIR application can connect to, read from, and write to an unencrypted database file. Starting with Adobe AIR 1.5, AIR includes the capability of creating and connecting to encrypted database files. When you use an encrypted database, in order to connect to the database an application must provide the correct encryption key. If the incorrect encryption key (or no key) is provided, the application is not able to connect to the database. Consequently, the application can’t read data from the database or write to or change data in the database.

To use an encrypted database, you must create the database as an encrypted database. With an existing encrypted database, you can open a connection to the database. You can also change the encryption key of an encrypted database. Other than creating and connecting to encrypted databases, the techniques for working with an encrypted database are the same as for working with an unencrypted one. In particular, executing SQL statements is the same regardless of whether a database is encrypted or not.

Uses for an encrypted database

Encryption is useful any time you want to restrict access to the information stored in a database. The database encryption functionality of Adobe AIR can be used for several purposes. The following are some examples of cases where you would want to use an encrypted database:

- A read-only cache of private application data downloaded from a server
- A local application store for private data that is synchronized with a server (data is sent to and loaded from the server)
- Encrypted files used as the file format for documents created and edited by the application. The files could be private to one user, or could be designed to be shared among all users of the application.
- Any other use of a local data store, such as the ones described in “Uses for local SQL databases” on page 243, where the data must be kept private from people who have access to the machine or the database files.

Understanding the reason why you want to use an encrypted database helps you decide how to architect your application. In particular, it can affect how your application creates, obtains, or stores the encryption key for the database. For more information about these considerations, see “Considerations for using encryption with a database” on page 278.

Other than an encrypted database, an alternative mechanism for keeping sensitive data private is the encrypted local store. With the encrypted local store, you store a single ByteArray value using a String key. Only the AIR application that stored the value can access it, and only on the computer on which it is stored. With the encrypted local store, it isn’t necessary to create your own encryption key. For these reasons, the encrypted local store is most suitable for easily storing a single value or set of values that can easily be encoded in a ByteArray. An encrypted database is most suitable for larger data sets where structured data storage and querying are desirable. For more information about using the encrypted local store, see “Storing encrypted data” on page 290.

Creating an encrypted database

To use an encrypted database, the database file must be encrypted when it is created. Once a database is created as unencrypted, it can’t be encrypted later. Likewise, an encrypted database can’t be unencrypted later. If needed you can change the encryption key of an encrypted database. For details, see “Changing the encryption key of a database” on page 277. If you have an existing database that’s not encrypted and you want to use database encryption, you can create a new encrypted database and copy the existing table structure and data to the new database.
Creating an encrypted database is nearly identical to creating an unencrypted database, as described in “Creating a database” on page 247. You first create a SQLConnection instance that represents the connection to the database. You create the database by calling the SQLConnection object’s open() method or openAsync() method, specifying for the database location a file that doesn’t exist yet. The only difference when creating an encrypted database is that you provide a value for the encryptionKey parameter (the open() method’s fifth parameter and the openAsync() method’s sixth parameter).

A valid encryptionKey parameter value is a ByteArray object containing exactly 16 bytes. The examples below demonstrate creating an encrypted database. For simplicity, in these examples the encryption key is hard-coded in the application code. However, this technique is strongly discouraged because it is not secure.

```javascript
var conn = new air.SQLConnection();

var encryptionKey = new air.ByteArray();
encryptionKey.writeUTFBytes("Some16ByteString"); // This technique is not secure!

// Create an encrypted database in asynchronous mode
conn.openAsync(dbFile, air.SQLMode.CREATE, null, false, 1024, encryptionKey);

// Create an encrypted database in synchronous mode
conn.open(dbFile, air.SQLMode.CREATE, false, 1024, encryptionKey);
```

For an example demonstrating a recommended way to generate an encryption key, see “Example: Generating and using an encryption key” on page 279.

### Connecting to an encrypted database

Like creating an encrypted database, the procedure for opening a connection to an encrypted database is like connecting to an unencrypted database. That procedure is described in greater detail in “Connecting to a database” on page 250. You use the open() method to open a connection in synchronous execution mode, or the openAsync() method to open a connection in asynchronous execution mode. The only difference is that to open an encrypted database, you specify the correct value for the encryptionKey parameter (the open() method’s fifth parameter and the openAsync() method’s sixth parameter).

```javascript
var conn = new air.SQLConnection();

var encryptionKey = new air.ByteArray();
encryptionKey.writeUTFBytes("Some16ByteString"); // This technique is not secure!

// Create an encrypted database in asynchronous mode
conn.openAsync(dbFile, air.SQLMode.CREATE, null, false, 1024, encryptionKey);

// Create an encrypted database in synchronous mode
conn.open(dbFile, air.SQLMode.CREATE, false, 1024, encryptionKey);
```

If the encryption key that’s provided is not correct, an error occurs. For the open() method, a SQLError exception is thrown. For the openAsync() method, the SQLConnection object dispatches a SQLErrorEvent, whose error property contains a SQLError object. In either case, the SQLError object generated by the exception has the errorID property value 3138. That error ID corresponds to the error message “File opened is not a database file”.

// Include AIRAliases.js to use air.* shortcuts
var conn = new air.SQLConnection();
conn.addEventListener(air.SQLEvent.OPEN, openHandler);
conn.addEventListener(air.SQLSErrorEvent.ERROR, errorHandler);

// The database file is in the application storage directory
var folder = File.applicationStorageDirectory;
var dbFile = folder.resolvePath("DBSample.db");

var encryptionKey = new air.ByteArray();
encryptionKey.writeUTFBytes("Some16ByteString"); // This technique is not secure!

conn.openAsync(dbFile, air.SQLMode.UPDATE, null, false, 1024, encryptionKey);

function openHandler(event)
{
    air.trace("the database opened successfully");
}

function errorHandler(event)
{
    if (event.error.errorID == 3138)
    {
        air.trace("Incorrect encryption key");
    }
    else
    {
        air.trace("Error message:", event.error.message);
        air.trace("Details:", event.error.details);
    }
}

The following example demonstrates opening an encrypted database in synchronous execution mode. For simplicity, in this example the encryption key is hard-coded in the application code. However, this technique is strongly discouraged because it is not secure.
// Include AIRAliases.js to use air.* shortcuts

var conn = new air.SQLConnection();

// The database file is in the application storage directory
var folder:File = File.applicationStorageDirectory;
var dbFile:File = folder.resolvePath("DBSample.db");

var encryptionKey = new air.ByteArray();
encryptionKey.writeUTFBytes("Some16ByteString"); // This technique is not secure!

try
{
  conn.open(dbFile, air.SQLMode.UPDATE, false, 1024, encryptionKey);
  air.trace("the database was created successfully");
}
catch (error)
{
  if (error.errorID == 3138)
  {
    air.trace("Incorrect encryption key");
  }
  else
  {
    air.trace("Error message:", error.message);
    air.trace("Details:", error.details);
  }
}

For an example demonstrating a recommended way to generate an encryption key, see "Example: Generating and using an encryption key" on page 279.

**Changing the encryption key of a database**

When a database is encrypted, you can change the encryption key for the database at a later time. To change a database’s encryption key, first open a connection to the database by creating a SQLConnection instance and calling its `open()` or `openAsync()` method. Once the database is connected, call the `reencrypt()` method, passing the new encryption key as an argument.

Like most database operations, the `reencrypt()` method’s behavior varies depending on whether the database connection uses synchronous or asynchronous execution mode. If you use the `open()` method to connect to the database, the `reencrypt()` operation runs synchronously. When the operation finishes, execution continues with the next line of code:

```
var newKey = new air.ByteArray();
// ... generate the new key and store it in newKey
conn.reencrypt(newKey);
```

On the other hand, if the database connection is opened using the `openAsync()` method, the `reencrypt()` operation is asynchronous. Calling `reencrypt()` begins the reencryption process. When the operation completes, the SQLConnection object dispatches a `reencrypt` event. You use an event listener to determine when the reencryption finishes:
The `reencrypt()` operation runs in its own transaction. If the operation is interrupted or fails (for example, if the application is closed before the operation finishes) the transaction is rolled back. In that case, the original encryption key is still the encryption key for the database.

The `reencrypt()` method can't be used to remove encryption from a database. Passing a `null` value or encryption key that's not a 16-byte `ByteArray` to the `reencrypt()` method results in an error.

**Considerations for using encryption with a database**

The section “Uses for an encrypted database” on page 274 presents several cases in which you would want to use an encrypted database. It’s obvious that the usage scenarios of different applications (including these and other scenarios) have different privacy requirements. The way you architect the use of encryption in your application plays an important part in controlling how private a database’s data is. For example, if you are using an encrypted database to keep personal data private, even from other users of the same machine, then each user’s database needs its own encryption key. For the greatest security, your application can generate the key from a user-entered password. Basing the encryption key on a password ensures that even if another person is able to impersonate the user’s account on the machine, the data still can’t be accessed. On the other end of the privacy spectrum, suppose you want a database file to be readable by any user of your application but not to other applications. In that case every installed copy of the application needs access to a shared encryption key.

You can design your application, and in particular the technique used to generate the encryption key, according to the level of privacy that you want for your application data. The following list provides design suggestions for various levels of data privacy:

- To make a database accessible to any user who has access to the application on any machine, use a single key that’s available to all instances of the application. For example, the first time an application runs it can download the shared encryption key from a server using a secure protocol such as SSL. It can then save the key in the encrypted local store for future use. As an alternative, encrypt the data per-user on the machine, and synchronize the data with a remote data store such as a server to make the data portable.

- To make a database accessible to a single user on any machine, generate the encryption key from a user secret (such as a password). In particular, do not use any value that’s tied to a particular computer (such as a value stored in the encrypted local store) to generate the key. As an alternative, encrypt the data per-user on the machine, and synchronize the data with a remote data store such as a server to make the data portable.

- To make a database accessible only to a single individual on a single machine, generate the key from a password and a generated salt. For an example of this technique, see “Example: Generating and using an encryption key” on page 279.
The following are additional security considerations that are important to keep in mind when designing an application to use an encrypted database:

- A system is only as secure as its weakest link. If you are using a user-entered password to generate an encryption key, consider imposing minimum length and complexity restrictions on passwords. A short password that uses only basic characters can be guessed quickly.

- The source code of an AIR application is stored on a user’s machine in plain text (for HTML content) or an easily decompilable binary format (for SWF content). Because the source code is accessible, two points to remember are:
  - Never hard-code an encryption key in your source code
  - Always assume that the technique used to generate an encryption key (such as random character generator or a particular hashing algorithm) can be easily worked out by an attacker

- AIR database encryption uses the Advanced Encryption Standard (AES) with Counter with CBC-MAC (CCM) mode. This encryption cipher requires a user-entered key to be combined with a salt value to be secure. For an example of this, see “Example: Generating and using an encryption key” on page 279.

- When you elect to encrypt a database, all disk files used by the database engine in conjunction with that database are encrypted. However, the database engine holds some data temporarily in an in-memory cache to improve read-and-write-time performance in transactions. Any memory-resident data is unencrypted. If an attacker is able to access the memory used by an AIR application, for example by using a debugger, the data in a database that is currently open and unencrypted would be available.

**Example: Generating and using an encryption key**

This example application demonstrates one technique for generating an encryption key. This application is designed to provide the highest level of privacy and security for users’ data. One important aspect of securing private data is to require the user to enter a password each time the application connects to the database. Consequently, as shown in this example, an application that requires this level of privacy should never directly store the database encryption key.

The application consists of two parts: an ActionScript class that generates an encryption key (the EncryptionKeyGenerator class), and a basic user interface that demonstrates how to use that class. For the complete source code, see “Complete example code for generating and using an encryption key” on page 283.

**Using the EncryptionKeyGenerator class to obtain a secure encryption key**

The section “Understanding the EncryptionKeyGenerator class” on page 281 details the techniques the EncryptionKeyGenerator class uses to generate an encryption key for a database. However, it isn’t necessary to understand these details to use the EncryptionKeyGenerator class in your application.

Follow these steps to use the EncryptionKeyGenerator class in your application:

1. Download the EncryptionKeyGenerator library. The EncryptionKeyGenerator class is included in the open-source ActionScript 3.0 core library (as3corelib) project. You can download the as3corelib package including source code and documentation. You can also download the SWC or source code files from the project page.

2. Extract the SWF file from the SWC. To extract the SWF file, rename the SWC file with the “.zip” filename extension and open the ZIP file. Extract the SWF file from the ZIP file and place it in a location where your application source code can find it. For example, you could place it in the folder containing your application’s main HTML file. You can rename the SWF file if you desire. In this example, the SWF file is named “EncryptionKeyGenerator.swf.”

3. In your application source code, import the SWF code library by adding a `<script>` block linking to the SWF file. This technique is explained in “Using ActionScript libraries within an HTML page” on page 60. The following code makes the SWF file available as a code library:

```html
<script type="application/x-shockwave-flash" src="EncryptionKeyGenerator.swf"/>
```
By default the class is available using the code `window.runtime` followed by the full package and class name. For the EncryptionKeyGenerator, the full name is:

```javascript
window.runtime.com.adobe.air.crypto.EncryptionKeyGenerator
```

You can create an alias for the class to avoid having to type the full name. The following code creates the alias `ekg.EncryptionKeyGenerator` to represent the EncryptionKeyGenerator class:

```javascript
var ekg;
if (window.runtime)
{
    if (!ekg) ekg = {};
    ekg.EncryptionKeyGenerator = window.runtime.com.adobe.air.crypto.EncryptionKeyGenerator;
}
```

4 Before the point where the code creates the database or opens a connection to it, add code to create an EncryptionKeyGenerator instance by calling the EncryptionKeyGenerator() constructor.

```javascript
var keyGenerator = new ekg.EncryptionKeyGenerator();
```

5 Obtain a password from the user:

```javascript
var password = passwordInput.value;
if (!keyGenerator.validateStrongPassword(password))
{
    // display an error message
    return;
}
```

The EncryptionKeyGenerator instance uses this password as the basis for the encryption key (shown in the next step). The EncryptionKeyGenerator instance tests the password against certain strong password validation requirements. If the validation fails, an error occurs. As the example code shows, you can check the password ahead of time by calling the EncryptionKeyGenerator object's validateStrongPassword() method. That way you can determine whether the password meets the minimum requirements for a strong password and avoid an error.

6 Generate the encryption key from the password:

```javascript
var encryptionKey = keyGenerator.getEncryptionKey(password);
```

The getEncryptionKey() method generates and returns the encryption key (a 16-byte ByteArray). You can then use the encryption key to create your new encrypted database or open your existing one.

The getEncryptionKey() method also accepts a second (optional) parameter, the overrideSaltELSKey parameter. The EncryptionKeyGenerator creates a random value (known as a salt) that is used as part of the encryption key. In order to be able to re-create the encryption key, the salt value is stored in the Encrypted Local Store (ELS) of your AIR application. By default, the EncryptionKeyGenerator class uses a particular String as the ELS key. Although unlikely, it’s possible that the key can conflict with another key your application uses. Instead of using the default key, you might want to specify your own ELS key. In that case, specify a custom key by passing it as the second getEncryptionKey() parameter, as shown here:

**Note:** To maintain the highest level of security and privacy for data, an application must require the user to enter a password each time the application connects to the database. Do not store the user’s password or the database encryption key directly. Doing so exposes security risks. Instead, as demonstrated in this example, an application should use the same technique to derive the encryption key from the password both when creating the encrypted database and when connecting to it later.
var customKey = "My custom ELS salt key";
var encryptionKey = keyGenerator.getEncryptionKey(password, customKey);

7 Create or open the database

With an encryption key returned by the `getEncryptionKey()` method, your code can create a new encrypted database or attempt to open the existing encrypted database. In either case you use the SQLConnection class's `open()` or `openAsync()` method, as described in “Creating an encrypted database” on page 274 and “Connecting to an encrypted database” on page 275.

In this example, the application is designed to open the database in asynchronous execution mode. The code sets up the appropriate event listeners and calls the `openAsync()` method, passing the encryption key as the final argument:

```javascript
conn.addEventListener(air.SQLEvent.OPEN, openHandler);
conn.addEventListener(air.SQLErrorEvent.ERROR, openError);
conn.openAsync(dbFile, air.SQLMode.CREATE, null, false, 1024, encryptionKey);
```

In the listener methods, the code removes the event listener registrations. It then displays a status message indicating whether the database was created, opened, or whether an error occurred. The most noteworthy part of these event handlers is in the `openError()` method. In that method an if statement checks if the database exists (meaning that the code is attempting to connect to an existing database) and if the error ID matches the constant `EncryptionKeyGenerator.ENCRYPTED_DB_PASSWORD_ERROR_ID`. If both of these conditions are true, it probably means that the password the user entered is incorrect. (It could also mean that the specified file isn’t a database file at all.) The following is the code that checks the error ID:

```javascript
if (!createNewDB && event.error.errorID ==
  ekg.EncryptionKeyGenerator.ENCRYPTED_DB_PASSWORD_ERROR_ID)
{
  statusMsg.innerHTML = "<p class='error'>Incorrect password!</p>";
}
else
{
  statusMsg.innerHTML = "<p class='error'>Error creating or opening database.</p>";
}
```

For the complete code for the example event listeners, see “Complete example code for generating and using an encryption key” on page 283.

**Understanding the EncryptionKeyGenerator class**

It isn’t necessary to understand the inner workings of the EncryptionKeyGenerator class to use it to create a secure encryption key for your application database. The process for using the class is explained in “Using the EncryptionKeyGenerator class to obtain a secure encryption key” on page 279. However, you might find it valuable to understand the techniques that the class uses. For example, you might want to adapt the class or incorporate some of its techniques for situations where a different level of data privacy is desired.

The EncryptionKeyGenerator class is included in the open-source ActionScript 3.0 core library (as3corelib) project. You can download the as3corelib package including source code and documentation. You can also view the source code on the project site or download it to follow along with the explanations.

When code creates an EncryptionKeyGenerator instance and calls its `getEncryptionKey()` method, several steps are taken to ensure that only the rightful user can access the data. The process is the same to generate an encryption key from a user-entered password before the database is created as well as to re-create the encryption key to open the database.
Obtain and validate a strong password

When code calls the `getEncryptionKey()` method, it passes in a password as a parameter. The password is used as the basis for the encryption key. By using a piece of information that only the user knows, this design ensures that only the user who knows the password can access the data in the database. Even if an attacker accesses the user’s account on the computer, the attacker can’t get into the database without knowing the password. For maximum security, the application never stores the password.

In the example application, `passwordInput` is the id of an HTML `<input>` element in which the user enters the password. Rather than manipulating the element’s value directly, the application copies the password into a variable named `password`.

```javascript
var password = passwordInput.value;
```

The example application then creates an `EncryptionKeyGenerator` instance and calls its `getEncryptionKey()` method, using the `password` variable as an argument:

```javascript
var keyGenerator = new ekg.EncryptionKeyGenerator();
var encryptionKey = keyGenerator.getEncryptionKey(password);
```

The first step the `EncryptionKeyGenerator` class takes when the `getEncryptionKey()` method is called is to check the user-entered password to ensure that it meets the password strength requirements. In this case the password must be 8 - 32 characters long. It must contain a mix of uppercase and lowercase letters and at least one number or symbol character.

Expand the password to 256 bits

Later in the process, the password is required to be 256 bits long. Rather than require each user to enter a password that’s exactly 256 bits (32 characters) long, the code creates a longer password by repeating the password characters.

The following is the code for the `concatenatePassword()` method:

```javascript
If the password is less than 256 bits, the code concatenates the password with itself to make it 256 bits. If the length doesn’t work out exactly, the last repetition is shortened to get exactly 256 bits.

Generate or retrieve a 256-bit salt value

The next step is to get a 256-bit salt value that in a later step is combined with the password. A *salt* is a random value that is added to or combined with a user-entered value to form a password. Using a salt with a password ensures that even if a user chooses a real word or common term as a password, the password-plus-salt combination that the system uses is a random value. This randomness helps guard against a dictionary attack, where an attacker uses a list of words to attempt to guess a password. In addition, by generating the salt value and storing it in the encrypted local store, it is tied to the user’s account on the machine on which the database file is located.

If the application is calling the `getEncryptionKey()` method for the first time, the code creates a random 256-bit salt value. Otherwise, the code loads the salt value from the encrypted local store.

Combine the 256-bit password and salt using the XOR operator

The code now has a 256-bit password and a 256-bit salt value. It next uses a bitwise XOR operation to combine the salt and the concatenated password into a single value. In effect, this technique creates a 256-bit password consisting of characters from the entire range of possible characters. This principle is true even though most likely the actual password input consists of primarily alphanumeric characters. This increased randomness provides the benefit of making the set of possible passwords large without requiring the user to enter a long complex password.
Hash the key
Once the concatenated password and the salt have been combined, the next step is to further secure this value by hashing it using the SHA-256 hashing algorithm. Hashing the value makes it more difficult for an attacker to reverse-engineer it.

Extract the encryption key from the hash
The encryption key must be a ByteArray that is exactly 16 bytes (128 bits) long. The result of the SHA-256 hashing algorithm is always 256 bits long. Consequently, the final step is to select 128 bits from the hashed result to use as the actual encryption key.

It isn’t necessary to use the first 128 bits as the encryption key. You could select a range of bits starting at some arbitrary point, you could select every other bit, or use some other way of selecting bits. The important thing is that the code selects 128 distinct bits, and that the same 128 bits are used each time.

Complete example code for generating and using an encryption key
The following is the complete code for the example application “Generating and using an encryption key.” The code consists of two parts.

The example uses the EncryptionKeyGenerator class to create an encryption key from a password. The EncryptionKeyGenerator class is included in the open-source ActionScript 3.0 core library (as3corelib) project. You can download the as3corelib package including source code and documentation. You can also download the SWC or source code files from the project page.

The application HTML file contains the source code for a simple application that creates or opens a connection to an encrypted database:

```
<html>
  <head>
    <title>Encrypted Database Example (HTML)</title>
    <style type="text/css">
      body
      {
        padding-top: 25px;
        font-family: Verdana, Arial;
        font-size: 14px;
      }
      div
      {
        width: 85%;
        margin-left: auto;
        margin-right: auto;
      }
      .error {color: #990000}
      .success {color: #009900}
    </style>
  </head>
  <script type="text/javascript" src="AIRAliases.js"></script>
  <script type="application/x-shockwave-flash" src="EncryptionKeyGenerator.swf"></script>
  <script type="text/javascript">
    // set up the class shortcut
    var ekg;
    if (window.runtime)
    {
      if (!ekg) ekg = {};
      ekg.EncryptionKeyGenerator =
```
window.runtime.com.adobe.air.crypto.EncryptionKeyGenerator;
}

// app globals
var dbFileName = "encryptedDatabase.db";
var dbFile;
var createNewDB = true;
var conn;

// UI elements
var instructions;
var passwordInput;
var openButton;
var statusMsg;

function init()
{

    // UI elements
    instructions = document.getElementById("instructions");
    passwordInput = document.getElementById("passwordInput");
    openButton = document.getElementById("openButton");
    statusMsg = document.getElementById("statusMsg");

    conn = new air.SQLConnection();
    dbFile = air.File.applicationStorageDirectory.resolvePath(dbFileName);
    if (dbFile.exists)
    {
        createNewDB = false;
        instructions.innerHTML = "<p>Enter your database password to open the encrypted database.</p>"
        openButton.value = "Open Database";
    }
}

function openConnection()
{
    var keyGenerator = new ekg.EncryptionKeyGenerator();

    var password = passwordInput.value;

    if (password == null || password.length <= 0)
    {
        statusMsg.innerHTML = "<p class='error'>Please specify a password.</p>";
        return;
    }

    if (!keyGenerator.validateStrongPassword(password))
    {
        statusMsg.innerHTML = "<p class='error'>The password must be 8-32 characters long. It must contain at least one lowercase letter, at least one uppercase letter, and at least one number or symbol.</p>";
        return;
    }

    passwordInput.value = "";
    passwordInput.disabled = true;
    openButton.disabled = true;
statusMsg.innerHTML = "";
var encryptionKey = keyGenerator.getEncryptionKey(password);
conn.addEventListener(air.SQLEvent.OPEN, openHandler);
conn.addEventListener(air.SQLErrorEvent.ERROR, openError);

conn.openAsync(dbFile, air.SQLMode.CREATE, null, false, 1024, encryptionKey);
}

function openHandler(event)
{
conn.removeEventListener(air.SQLEvent.OPEN, openHandler);
conn.removeEventListener(air.SQLErrorEvent.ERROR, openError);

if (createNewDB)
{
    statusMsg.innerHTML = "<p class='success'>The encrypted database was created successfully.</p>";
} else
{
    statusMsg.innerHTML = "<p class='success'>The encrypted database was opened successfully.</p>";
}
}

function openError(event)
{
conn.removeEventListener(air.SQLEvent.OPEN, openHandler);
conn.removeEventListener(air.SQLErrorEvent.ERROR, openError);

if (!createNewDB & event.error.errorID == ekg.EncryptionKeyGenerator.ENCRIPTED_DB_PASSWORD_ERROR_ID)
{
    statusMsg.innerHTML = "<p class='error'>Incorrect password!</p>";
} else
{
    statusMsg.innerHTML = "<p class='error'>Error creating or opening database.</p>";
}
</script>
</head>

<body onload="init();">

<div id="instructions"><p>Enter a password to create an encrypted database. The next time you open the application, you will need to re-enter the password to open the database again.</p></div>

<input id="passwordInput" type="password"/>
<input id="openButton" type="button" value="Create Database" onclick="openConnection();"/>

</body>
</html>
Strategies for working with SQL databases

There are various ways that an application can access and work with a local SQL database. The application design can vary in terms of how the application code is organized, the sequence and timing of how operations are performed, and so on. The techniques you choose can have an impact on how easy it is to develop your application. They can affect how easy it is to modify the application in future updates. They can also affect how well the application performs from the users’ perspective.

Distributing a pre-populated database

When you use an AIR local SQL database in your application, the application expects a database with a certain structure of tables, columns, and so forth. Some applications also expect certain data to be pre-populated in the database file. One way to ensure that the database has the proper structure is to create the database within the application code. When the application loads it checks for the existence of its database file in a particular location. If the file doesn’t exist, the application executes a set of commands to create the database file, create the database structure, and populate the tables with the initial data.

The code that creates the database and its tables is frequently complex. It is often only used once in the installed lifetime of the application, but still adds to the size and complexity of the application. As an alternative to creating the database, structure, and data programmatically, you can distribute a pre-populated database with your application. To distribute a predefined database, include the database file in the application’s AIR package.

Like all files that are included in an AIR package, a bundled database file is installed in the application directory (the directory represented by the `File.applicationDirectory` property). However, files in that directory are read only.

Use the file from the AIR package as a “template” database. The first time a user runs the application, copy the original database file into the user’s application storage directory (or another location), and use that database within the application.

Improving database performance

Several techniques that are built in to Adobe AIR allow you to improve the performance of database operations in your application.

In addition to the techniques described here, the way a SQL statement is written can also affect database performance. Frequently, there are multiple ways to write a SQL `SELECT` statement to retrieve a particular result set. In some cases, the different approaches 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.

Use one SQLStatement instance for each SQL statement

Before any SQL statement is executed, the runtime prepares (compiles) it to determine the steps that are performed internally to carry out the statement. When you call `SQLStatement.execute()` on a `SQLStatement` instance that hasn’t executed previously, the statement is automatically prepared before it is executed. On subsequent calls to the `execute()` method, as long as the `SQLStatement.text` property hasn’t changed the statement is still prepared. Consequently, it executes faster.

To gain the maximum benefit from reusing statements, if values need to change between statement executions, use statement parameters to customize your statement. (Statement parameters are specified using the `SQLStatement.parameters` associative array property.) Unlike changing the `SQLStatement` instance’s `text` property, if you change the values of statement parameters the runtime isn’t required to prepare the statement again. For more information about using parameters in statements, see “Using parameters in statements” on page 253.
Because preparing and executing a statement is an operation that is potentially demanding, a good strategy is to preload initial data and then execute other statements in the background. Load the data that the application needs first. When the initial start-up operations of your application have completed, or at another “idle” time in the application, execute other statements. For instance, if your application doesn’t access the database at all in order to display its initial screen, wait until that screen displays, then open the database connection, and finally create the SQLStatement instances and execute any that you can. Alternatively, suppose when your application starts up it immediately displays some data, such as the result of a particular query. In that case, go ahead and execute the SQLStatement instance for that query. After the initial data is loaded and displayed, create SQLStatement instances for other database operations and if possible execute other statements that are needed later.

When you’re reusing a SQLStatement instance, your application needs to keep a reference to the SQLStatement instance once it has been prepared. To keep a reference to the instance, declare the variable as a class-scope variable rather than a function-scope variable. One good way to do this is to structure your application so that a SQL statement is wrapped in a single class. A group of statements that are executed in combination can also be wrapped in a single class. By defining the SQLStatement instance or instances as member variables of the class, they persist as long as the instance of the wrapper class exists in the application. At a minimum, you can simply define a variable containing the SQLStatement instance outside of a function so that the instance persists in memory. For example, declare the SQLStatement instance as a member variable in an ActionScript class or as a non-function variable in a JavaScript file. You can then set the statement’s parameter values and call its `execute()` method when you want to actually run the query.

**Group multiple operations in a transaction**

Suppose you’re executing a large number of SQL statements that involve adding or changing data (INSERT or UPDATE statements). You can get a significant increase in performance by executing all the statements within an explicit transaction. If you don’t explicitly begin a transaction, each of the statements runs in its own automatically created transaction. After each transaction (each statement) finishes executing, the runtime writes the resulting data to the database file on the disk. On the other hand, consider what happens if you explicitly create a transaction and execute the statements in the context of that transaction. The runtime makes all the changes in memory, then writes all the changes to the database file at one time when the transaction is committed. Writing the data to disk is usually the most time-intensive part of the operation. Consequently, writing to the disk one time rather than once per SQL statement can improve performance significantly.

**Minimize 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 along with table names in a statement. (Use “main” if it’s the main database). For example, use `SELECT employeeId FROM main.employees` rather than `SELECT employeeId FROM employees`. Explicitly specifying the database name prevents the runtime from having to check each database to find the matching table. It also prevents the possibility of having the runtime choose the wrong database. Follow this rule even if a SQLConnection is only connected to a single database, because behind the scenes the SQLConnection is also connected to a temporary database that is accessible through SQL statements.

- Always explicitly specify column names in a SELECT or INSERT statement.

- Break up the rows returned by a SELECT statement that retrieves a large number of rows: see “Retrieving SELECT results in parts” on page 261.
Avoid schema changes
If possible, avoid changing the schema (table structure) of a database once you've added data into the database's tables. Normally a database file is structured with the table definitions at the start of the file. When you open a connection to a database, the runtime loads those definitions. When you add data to database tables, that data is added to the file after the table definition data. However, if you make schema changes such as adding a column to a table or adding a new table, the new table definition data is mixed in with the table data in the database file. If the table definition data is not all at the start of the database file, it takes longer to open a connection to the database as the runtime reads the table definition data from different parts of the file.

If you do need to make schema changes, you can call the `SQLConnection.compact()` method after completing the changes. This operation restructures the database file so that the table definition data is located together at the start of the file. However, the `compact()` operation can be time-intensive, especially as a database file grows larger.

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. For additional techniques for improving database applications, see "Improving database performance" on page 286.

Pre-create database connections
Even if your application doesn’t execute any statements when it first loads, instantiate a `SQLConnection` object and call its `open()` or `openAsync()` method ahead of time (such as after the initial application startup) to avoid delays when running statements. See “Connecting to a database” on page 250.

Reuse database connections
If you access a certain database throughout the execution time of your application, keep a reference to the `SQLConnection` instance, and reuse it throughout the application, rather than closing and reopening the connection. See “Connecting to a database” on page 250.

Favor asynchronous execution mode
When writing data-access code, it can be tempting to execute operations synchronously rather than asynchronously, because using synchronous operations frequently requires shorter and less complex code. However, as described in “Using synchronous and asynchronous database operations” on page 269, synchronous operations can have a performance impact that is obvious to users and detrimental to their experience with an application. The amount of time a single operation takes varies according to the operation and particularly the amount of data it involves. For instance, a SQL `INSERT` statement that only adds a single row to the database takes less time than a `SELECT` statement that retrieves thousands of rows of data. However, when you’re using synchronous execution to perform multiple operations, the operations are usually strung together. Even if the time each single operation takes is very short, the application is frozen until all the synchronous operations finish. As a result, the cumulative time of multiple operations strung together may be enough to stall your application.

Use asynchronous operations as a standard approach, especially with operations that involve large numbers of rows. There is a technique for dividing up the processing of large sets of `SELECT` statement results, described in "Retrieving `SELECT` results in parts" on page 261. However, this technique can only be used in asynchronous execution mode. Only use synchronous operations when you can’t achieve certain functionality using asynchronous programming, when you’ve considered the performance trade-off that your application’s users will face, and when you’ve tested your application so that you know how your application’s performance is affected. Using asynchronous execution can involve more complex coding. However, remember that you only have to write the code once, but the application’s users have to use it repeatedly, fast or slow.
In many cases, by using a separate SQLStatement instance for each SQL statement to be executed, multiple SQL operations can be queued up at one time, which makes asynchronous code like synchronous code in terms of how the code is written. For more information, see “Understanding the asynchronous execution model” on page 272.

**Use separate SQL statements and don’t change the SQLStatement’s text property**

For any SQL statement that is executed more than once in an application, create a separate SQLStatement instance for each SQL statement. Use that SQLStatement instance each time that SQL command executes. For example, suppose you are building an application that includes four different SQL operations that are performed multiple times. In that case, create four separate SQLStatement instances and call each statement’s `execute()` method to run it. Avoid the alternative of using a single SQLStatement instance for all SQL statements, redefining its `text` property each time before executing the statement. See “Use one SQLStatement instance for each SQL statement” on page 286 for more information.

**Use statement parameters**

Use SQLStatement parameters—never concatenate user input into statement text. Using parameters makes your application more secure because it prevents the possibility of SQL injection attacks. It makes it possible to use objects in queries (rather than only SQL literal values). It also makes statements run more efficiently because they can be reused without needing to be recompiled each time they’re executed. See “Using parameters in statements” on page 253 for more information.

**Use constants for column and parameter names**

When you don’t specify an `itemClass` for a SQLStatement, to avoid spelling errors, define String constants containing a table’s column names. Use those constants in the statement text and for the property names when retrieving values from result objects. Also use constants for parameter names.
Chapter 26: Storing encrypted data

The Adobe® AIR® runtime provides a persistent encrypted local store (ELS) for each AIR application installed on a user's computer. This lets you save and retrieve data that is stored on the user's local hard drive in an encrypted format that cannot easily be deciphered by other users. A separate encrypted local store is used for each AIR application, and each AIR application uses a separate encrypted local store for each user.

**Note:** In addition to the encrypted local store, AIR also provides encryption for content stored in SQL databases. For details, see "Using encryption with SQL databases" on page 274.

You may want to use the encrypted local store to cache information that must be secured, such as login credentials for web services. The ELS is appropriate for storing information that must be kept private from other users. It does not, however, protect the data from other processes run under the same user account. It is thus not appropriate for protecting secret application data, such as DRM or encryption keys.

AIR uses DPAPI on Windows, KeyChain on Mac OS, and KeyRing or KWallet on Linux to associate the encrypted local store to each application and user. The encrypted local store uses AES-CBC 128-bit encryption.

Information in the encrypted local store is only available to AIR application content in the application security sandbox.

If you update an AIR application, the updated version retains access to any existing data in the encrypted local store unless:

- The items were added with the `stronglyBound` parameter set to `true`
- The existing and update versions are both published prior to AIR 1.5.3 and the update is signed with a migration signature (for more information see “Changing certificates” on page 373)

**Limitations of the encrypted local store**

The data in the encrypted local store is protected by the user's operating system account credentials. Other entities cannot access the data in the store unless they can login as that user. However, the data is not secure against access by other applications run by an authenticated user.

Because the user must be authenticated for these attacks to work, the user's private data is still protected (unless the user account itself is compromised). However, data that your application may want to keep secret from users, such as keys used for licensing or digital rights management, is not secure. Thus the ELS is not an appropriate location for storing such information. It is only an appropriate place for storing a user's private data, such as passwords.

Data in the ELS can be lost for a variety of reasons. For example, the user could uninstall the application and delete the encrypted file. Or, the publisher ID could be changed as a result of an update. Thus the ELS should be treated as a private cache, not a permanent data storage.

The `stronglyBound` parameter is deprecated and should not be set to `true`. Setting the parameter to `true` does not provide any additional protection for data. At the same time, access to the data is lost whenever the application is updated — even if the publisher ID stays the same.

The encrypted local store may perform more slowly if the stored data exceeds 10MB.

When you uninstall an AIR application, the uninstaller does not delete data stored in the encrypted local store.

The best practices for using the ELS include:

- Use the ELS to store sensitive user data such as passwords (setting stronglyBound to false)
Do not use the ELS to store applications secrets such as DRM keys or licensing tokens.

Provide a way for your application to recreate the data stored in the ELS if the ELS data is lost. For example, by prompting the user to re-enter their account credentials when necessary.

Do not use the `stronglyBound` parameter.

If you do set `stronglyBound` to `true`, do not migrate stored items during an update. Recreate the data after the update instead.

Only store relatively small amounts of data. For larger amounts of data, use an AIR SQL database with encryption.

### Adding data to the encrypted local store

Use the `setItem()` static method of the EncryptedLocalStore class to store data in the local store. The data is stored in a hash table, using strings as keys, with the data stored as byte arrays.

For example, the following code stores a string in the encrypted local store:

```javascript
var str = "Bob";
var bytes = new air.ByteArray();
bytes.writeUTFBytes(str);
air.EncryptedLocalStore.setItem("firstName", bytes);
```

The third parameter of the `setItem()` method, the `stronglyBound` parameter, is optional. When this parameter is set to `true`, the encrypted local store binds the stored item to the storing AIR application’s digital signature and bits:

```javascript
var str = "Bob";
var bytes = new air.ByteArray();
bytes.writeUTFBytes(str);
air.EncryptedLocalStore.setItem("firstName", bytes, false);
```

For an item that is stored with `stronglyBound` set to `true`, subsequent calls to `getItem()` only succeed if the calling AIR application is identical to the storing application (if no data in files in the application directory have changed). If the calling AIR application is different from the storing application, the application throws an Error exception when you call `getItem()` for a strongly bound item. If you update your application, it will not be able to read strongly bound data previously written to the encrypted local store.

If the `stronglyBound` parameter is set to `false` (the default), only the publisher ID needs to stay the same for the application to read the data. The bits of the application may change (and they need to be signed by the same publisher), but they do not need to be the exact same bits as were in application that stored the data. Updated applications with the same publisher ID as the original can continue to access the data.

**Note:** In practice, setting `stronglyBound` to `true` does not add any additional data protection. A "malicious" user could still alter an application to gain access to items stored in the ELS. Furthermore, data is protected from external, non-user threats just as strongly whether `stronglyBound` is set to `true` or `false`. For these reasons, setting `stronglyBound` to `true` is discouraged.

### Accessing data in the encrypted local store

Use the `getItem()` method to read data in the encrypted local store, passing in the name string assigned when the item was added. The data is returned as a `ByteArray` object. You can use the appropriate read methods of the `ByteArray` class to extract the data inside the byte array.
Removing data from the encrypted local store

You can delete a value from the encrypted local store by using the `EncryptedLocalStore.removeItem()` method, as in the following example:

```javascript
air.EncryptedLocalStore.removeItem("firstName");
```

You can clear all data from the encrypted local store by calling the `EncryptedLocalStore.reset()` method, as in the following example:

```javascript
air.EncryptedLocalStore.reset();
```
Chapter 27: Adding PDF content

Applications running in Adobe® AIR® can render not only SWF and HTML content, but also PDF content. AIR applications render PDF content using the HTMLLoader class, the WebKit engine, and the Adobe® Reader® browser plug-in. In an AIR application, PDF content can either stretch across the full height and width of your application or alternatively as a portion of the interface. The Adobe Reader browser plug-in controls display of PDF files in an AIR application, so modifications to the Reader toolbar interface (such as those for position, anchoring, and visibility) persist in subsequent viewing of PDF files in both AIR applications and the browser.

Important: In order to render PDF content in AIR, the user must have Adobe Reader or Adobe® Acrobat® version 8.1 or higher installed.

Detecting PDF Capability

If the user does not have an installed version of Adobe Reader or Adobe Acrobat 8.1 or higher, PDF content is not displayed in an AIR application. To detect if a user can render PDF content, first check the HTMLLoader.pdfCapability property. This property is set to one of the following constants of the HTMLPDFCapability class:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTMLPDFCapability.STATUS_OK</td>
<td>A sufficient version (8.1 or greater) of Adobe Reader is detected and PDF content can be loaded into an HTMLLoader object.</td>
</tr>
<tr>
<td>HTMLPDFCapability.ERROR_INSTALLED_READER_NOT_FOUND</td>
<td>No version of Adobe Reader is detected. An HTMLLoader object cannot display PDF content.</td>
</tr>
<tr>
<td>HTMLPDFCapability.ERROR_INSTALLED_READER_TOO_OLD</td>
<td>Adobe Reader has been detected, but the version is too old. An HTMLLoader object cannot display PDF content.</td>
</tr>
<tr>
<td>HTMLPDFCapability.ERROR_PREFERRED_READER_TOO_OLD</td>
<td>A sufficient version (8.1 or later) of Adobe Reader is detected, but the version of Adobe Reader that is set up to handle PDF content is older than Reader 8.1. An HTMLLoader object cannot display PDF content.</td>
</tr>
</tbody>
</table>

On Windows, if Adobe Acrobat or Adobe 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 is installed. In this case, if the value of the pdfCapability property is HTMLPDFCapability.STATUS_OK, when an AIR application attempts to load PDF content, the older version of Acrobat or Reader displays an alert (and no exception is thrown in the AIR application). If this is a possible situation for your end users, consider providing them with instructions to close Acrobat while running your application. You may want to display these instructions if the PDF content does not load within an acceptable time frame.

On Linux, AIR looks for Adobe Reader in the PATH exported by the user (if it contains the acroread command) and in the /opt/Adobe/Reader directory.

The following code detects whether a user can display PDF content in an AIR application, and if not traces the error code that corresponds to the HTMLPDFCapability error object:
if(air.HTMLLoader.pdfCapability == air.HTMLPDFCapability.STATUS_OK)
{
    air.trace("PDF content can be displayed");
}
else
{
    air.trace("PDF cannot be displayed. Error code:", HTMLLoader.pdfCapability);
}

## Loading PDF content

You can add a PDF to an AIR application by creating an HTMLLoader instance, setting its dimensions, and loading the path of a PDF.

You can add a PDF to an AIR application just as you would in a browser. For example, you can load PDF into the top-level HTML content of a window, into an object tag, in a frame, or in an iframe.

The following example loads a PDF from an external site. Replace the value of the src property of the iframe with the path to an available external PDF.

```html
<html>
<body>
<h1>PDF test</h1>
<iframe id="pdfFrame"
    width="100%"
    height="100%"
    src="http://www.example.com/test.pdf"/>
</body>
</html>
```

You can also load content from file URLs and AIR-specific URL schemes, such as app and app-storage. For example, the following code loads the test.pdf file in the PDFs subdirectory of the application directory:

`app:/js_api_reference.pdf`

For more information on AIR URL schemes, see "Using AIR URL schemes in URLs" on page 344.

## Scripting PDF content

You can use JavaScript to control PDF content just as you can in a web page in the browser.

JavaScript extensions to Acrobat provide the following features, among others:

- Controlling page navigation and magnification
- Processing forms within the document
- Controlling multimedia events

**HTML-PDF communication basics**

JavaScript in an HTML page can send a message to JavaScript in PDF content by calling the `postMessage()` method of the DOM object representing the PDF content. For example, consider the following embedded PDF content:

```html
<object id="PDFObj" data="test.pdf" type="application/pdf" width="100%" height="100%"/>
```

The following JavaScript code in the containing HTML content sends a message to the JavaScript in the PDF file:

```javascript
pdfObject = document.getElementById("PDFObj");
pdfObject.postMessage(["testMsg", "hello"]);
```

The PDF file can include JavaScript for receiving this message. You can add JavaScript code to PDF files in some contexts, including the document-, folder-, page-, field-, and batch-level contexts. Only the document-level context, which defines scripts that are evaluated when the PDF document opens, is discussed here.

A PDF file can add a `messageHandler` property to the `hostContainer` object. The `messageHandler` property is an object that defines handler functions to respond to messages. For example, the following code defines the function to handle messages received by the PDF file from the host container (which is the HTML content embedding the PDF file):

```javascript
this.hostContainer.messageHandler = {onMessage: myOnMessage};

function myOnMessage(aMessage)
{
    if(aMessage[0] == "testMsg")
    {
        app.alert("Test message: " + aMessage[1]);
    }
    else
    {
        app.alert("Error");
    }
}
```

JavaScript code in the HTML page can call the `postMessage()` method of the PDF object contained in the page. Calling this method sends a message ("Hello from HTML") to the document-level JavaScript in the PDF file:
<html>
<head>
<title>PDF Test</title>
<script>
function init()
{
    pdfObject = document.getElementById("PDFObj");
    try {
        pdfObject.postMessage(["alert", "Hello from HTML"]);
    }
    catch (e)
    {
        alert("Error: \n name = " + e.name + "\n message = " + e.message);
    }
}
</script>
</head>
<body onload='init()'>
<object id="PDFObj" data="test.pdf" type="application/pdf" width="100%" height="100%"/>
</body>
</html>

For a more advanced example, and for information on using Acrobat 8 to add JavaScript a PDF file, see Cross-scripting PDF content in Adobe AIR.

**Known limitations for PDF content in AIR**

PDF content in Adobe AIR has the following limitations:

- PDF content does not display in a window (a NativeWindow object) that is transparent (where the transparent property is set to true).
- The display order of a PDF file operates differently than other display objects in an AIR application. Although PDF content clips correctly according to HTML display order, it will always sit on top of content in the AIR application’s display order.
- PDF content does not display in a window that is in full-screen mode (when the displayState property of the Stage (the window.nativeWindow.stage property) is set to air.StageDisplayState.FULL_SCREEN or air.StageDisplayState.FULL_SCREEN_INTERACTIVE).
- If certain visual properties of an HTMLLoader object that contains a PDF document are changed, the PDF document will become invisible. These properties include the filters, alpha, rotation, and scaling properties. Changing these renders the PDF file invisible until the properties are reset. This is also true if you change these properties of display object containers that contain the HTMLLoader object.
- PDF content is visible only when the scaleMode property of the Stage object of the NativeWindow object containing the PDF content (the window.nativeWindow.stage property) is set to air.StageScaleMode.NO_SCALE. When it is set to any other value, the PDF content is not visible.
• Clicking links to content within the PDF file update the scroll position of the PDF content. Clicking links to content outside the PDF file redirect the HTMLLoader object that contains the PDF (even if the target of a link is a new window).

• PDF commenting workflows do not function in AIR.
298

Last updated 12/9/2009

Chapter 28: Working with sound
The Adobe® AIR® classes include many capabilities not available to HTML content running in the browser, including
capabilities for loading and playing sound content.

Basics of working with sound
Before you can control a sound, you need to load the sound into the Adobe AIR application. There are five ways you
can get audio data into AIR:

• You can load an external sound file such as an MP3 file into the application.
• You can embed the sound information into a SWF file, load it (using <script

src="[swfFile].swf"

type="application/x-shockwave-flash"/>) and play it.

• You can get audio input using a microphone attached to a user’s computer.
• You can access sound data that’s streamed from a server.
• You can dynamically generate sound data.
When you load sound data from an external sound file, you can begin playing back the start of the sound file while the
rest of the sound data is still loading.
Although there are various sound file formats used to encode digital audio, AIR supports sound files that are stored in
the MP3 format. It cannot directly load or play sound files in other formats like WAV or AIFF.
While you’re working with sound in AIR, you’ll likely work with several classes from the runtime.flash.media package.
The Sound class is the class you use to get access to audio information by loading a sound file or assigning a function
to an event that samples sound data and then starting playback. Once you start playing a sound, AIR gives you access
to a SoundChannel object. An audio file that you’ve loaded may only be one of several sounds that an application plays
simultaneously. Each individual sound that’s playing uses its own SoundChannel object; the combined output of all
the SoundChannel objects mixed together is what actually plays over the speakers. You use this SoundChannel
instance to control properties of the sound and to stop its playback. Finally, if you want to control the combined audio,
the SoundMixer class gives you control over the mixed output.
You can also use several other runtime classes to perform more specific tasks when you’re working with sound in AIR.
For more information on all the sound-related classes, see “Understanding the sound architecture” on page 298.
The Adobe AIR developer’s center provides a sample application: Using Sound in an HTML-based Application

Understanding the sound architecture
Your applications can load sound data from four main sources:

• External sound files loaded at run time
• Sound resources embedded within a SWF file
• Sound data from a microphone attached to the user’s system


• Sound data streamed from a remote media server, such as Flash Media Server
• Sound data being generated dynamically through the use of the `sampleData` event handler

Sound data can be fully loaded before it is played back, or it can be streamed, meaning that it is played back while it is still loading.

Adobe AIR supports sound files that are stored in the MP3 format. They cannot directly load or play sound files in other formats like WAV or AIFF. (However, AIR can also load and play AAC audio files using the NetStream class.)

The AIR sound architecture includes the following classes:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sound</td>
<td>The Sound class handles the loading of sound, manages basic sound properties, and starts a sound playing.</td>
</tr>
<tr>
<td>SoundChannel</td>
<td>When an application plays a Sound object, a new SoundChannel object is created to control the playback. The SoundChannel object controls the volume of both the left and right playback channels of the sound. Each sound that plays has its own SoundChannel object.</td>
</tr>
<tr>
<td>SoundLoaderContext</td>
<td>The SoundLoaderContext class specifies how many seconds of buffering to use when loading a sound, and whether the runtime looks for a cross-domain policy file from the server when loading a file. A SoundLoaderContext object is used as a parameter to the <code>Sound.load()</code> method.</td>
</tr>
<tr>
<td>SoundMixer</td>
<td>The SoundMixer class controls playback and security properties that pertain to all sounds in an application. In effect, multiple sound channels are mixed through a common SoundMixer object. Property values in the SoundMixer object affect all SoundChannel objects that are currently playing.</td>
</tr>
<tr>
<td>SoundTransform</td>
<td>The SoundTransform class contains values that control sound volume and panning. A SoundTransform object can be applied to an individual SoundChannel object, to the global SoundMixer object, or to a Microphone object, among others.</td>
</tr>
<tr>
<td>ID3Info</td>
<td>An ID3Info object contains properties that represent ID3 metadata information that is often stored in MP3 sound files.</td>
</tr>
<tr>
<td>Microphone</td>
<td>The Microphone class represents a microphone or other sound input device attached to the user’s computer. Audio input from a microphone can be routed to local speakers or sent to a remote server. The Microphone object controls the gain, sampling rate, and other characteristics of its own sound stream.</td>
</tr>
</tbody>
</table>

Each sound that is loaded and played needs its own instance of the Sound class and the SoundChannel class. During playback, the SoundMixer class mixes the output from multiple SoundChannel instances.

The Sound, SoundChannel, and SoundMixer classes are not used for sound data obtained from a microphone or from a streaming media server like Flash Media Server.

### Loading external sound files

Each instance of the Sound class exists to load and trigger the playback of a specific sound resource. An application can’t reuse a Sound object to load more than one sound. To load a new sound resource, it should create a new Sound object.

### Creating a sound object

If you are loading a small sound file, such as a click sound to be attached to a button, your application can create a Sound and have it automatically load the sound file, as the following example shows:

```javascript
var req = new air.URLRequest("click.mp3");
var s = new air.Sound(req);
```
The `Sound()` constructor accepts a `URLRequest` object as its first parameter. When a value for the `URLRequest` parameter is supplied, the new `Sound` object starts loading the specified sound resource automatically.

In all but the simplest cases, your application should pay attention to the sound’s loading progress and watch for errors during loading. For example, if the click sound is fairly large, the application may not completely load it by the time the user clicks the button that triggers the sound. Trying to play an unloaded sound could cause a run-time error. It’s safer to wait for the sound to load completely before letting users take actions that can start sounds playing.

**About sound events**

A `Sound` object dispatches a number of different events during the sound loading process. Your application can listen for these events to track loading progress and make sure that the sound loads completely before playing. The following table lists the events that can be dispatched by a `Sound` object:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>open (air.Event.OPEN)</td>
<td>Dispatched right before the sound loading operation begins.</td>
</tr>
<tr>
<td>progress (air.ProgressEvent.PROGRESS)</td>
<td>Dispatched periodically during the sound loading process when data is received from the file or stream.</td>
</tr>
<tr>
<td>id3 (air.Event.ID3)</td>
<td>Dispatched when ID3 data is available for an MP3 sound.</td>
</tr>
<tr>
<td>complete (air.Event.COMPLETE)</td>
<td>Dispatched when all of the sound resource’s data has been loaded.</td>
</tr>
<tr>
<td>ioError (air.IOErrorEvent.IO_ERROR)</td>
<td>Dispatched when a sound file cannot be located or when the loading process is interrupted before all sound data can be received.</td>
</tr>
</tbody>
</table>

The following code illustrates how to play a sound after it has finished loading:

```javascript
var s = new air.Sound();
s.addEventListener(air.Event.COMPLETE, onSoundLoaded);
var req = new air.URLRequest("bigSound.mp3");
s.load(req);

function onSoundLoaded(event)
{
    var localSound = event.target;
    localSound.play();
}
```

First, the code sample creates a new `Sound` object without giving it an initial value for the `URLRequest` parameter. Then, it listens for the `complete` event from the `Sound` object, which causes the `onSoundLoaded()` method to execute when all the sound data is loaded. Next, it calls the `Sound.load()` method with a new `URLRequest` value for the sound file.

The `onSoundLoaded()` method executes when the sound loading is complete. The `target` property of the `Event` object is a reference to the `Sound` object. Calling the `play()` method of the `Sound` object then starts the sound playback.
Monitoring the sound loading process

Sound files can be very large and take a long time to load, especially if they are loaded from the Internet. An application can play sounds before they are fully loaded. You might want to give the user an indication of how much of the sound data has been loaded and how much of the sound has already been played.

The Sound class dispatches two events that make it relatively easy to display the loading progress of a sound: progress and complete. The following example shows how to use these events to display progress information about the sound being loaded:

```javascript
var s = new Sound();
s.addEventListener(air.ProgressEvent.PROGRESS, onLoadProgress);
s.addEventListener(air.Event.COMPLETE, onLoadComplete);
s.addEventListener(air.IOErrorEvent.IO_ERROR, onIOError);

var req = new air.URLRequest("bigSound.mp3");
s.load(req);

function onLoadProgress(event)
{
    var loadedPct = Math.round(100 * (event.bytesLoaded / event.bytesTotal));
    air.trace("The sound is " + loadedPct + ":% loaded.");
}

function onLoadComplete(event)
{
    var localSound = event.target;
    localSound.play();
}

function onIOError(event)
{
    air.trace("The sound could not be loaded: " + event.text);
}
```

This code first creates a Sound object and then adds listeners to that object for the progress and complete events. After the Sound.load() method has been called and the first data is received from the sound file, a progress event occurs and triggers the onLoadProgress() method.

The fraction of the sound data that has been loaded is equal to the value of the bytesLoaded property of the ProgressEvent object divided by the value of the bytesTotal property. The same bytesLoaded and bytesTotal properties are available on the Sound object as well.

This example also shows how an application can recognize and respond to an error when loading sound files. For example, if a sound file with the given filename cannot be located, the Sound object dispatches an ioError event. In the previous code, the onIOError() method executes and displays a brief error message when an error occurs.

Working with embedded sounds

In AIR, you can use JavaScript to access sounds embedded in SWF files. You can load these SWF files into the application using any of the following means:

- By loading the SWF file with a `<script>` tag in the HTML page
By loading a SWF file using the runtime.flash.display.Loader class

The exact method of embedding a sound file into your application’s SWF file varies according to your SWF development environment. For information on embedding media in SWF files, see the documentation for your SWF development environment.

To use the embedded sound, you reference the class name for that sound in ActionScript. For example, the following code starts by creating an instance of the automatically generated DrumSound class:

```actionscript
var drum = new DrumSound();
var channel = drum.play();
```

DrumSound is a subclass of the flash.media.Sound class, so it inherits the methods and properties of the Sound class. This includes the play() method, as the preceding example shows.

### Working with streaming sound files

When a sound file or video file is playing back while its data is still being loaded, it is said to be *streaming*. Sound files loaded from a remote server are often streamed so that the user doesn’t have to wait for all the sound data to load before listening to the sound.

The `SoundMixer.bufferTime` property represents the number of milliseconds of sound data that an application gathers before letting the sound play. In other words, if the `bufferTime` property is set to 5000, the application loads at least 5000 milliseconds worth of data from the sound file before the sound begins to play. The default `SoundMixer.bufferTime` value is 1000.

Your application can override the global `SoundMixer.bufferTime` value for an individual sound by explicitly specifying a new `bufferTime` value when loading the sound. To override the default buffer time, first create an instance of the SoundLoaderContext class, set its `bufferTime` property, and then pass it as a parameter to the `Sound.load()` method. The following example shows this:

```actionscript
var s = new air.Sound();
var url = "http://www.example.com/sounds/bigSound.mp3";
var req = new air.URLRequest(url);
var context = new air.SoundLoaderContext(8000, true);
s.load(req, context);
s.play();
```

As playback continues, AIR tries to keep the sound buffer at the same size or greater. If the sound data loads faster than the playback speed, playback continues without interruption. However, if the data loading rate slows down because of network limitations, the playhead could reach the end of the sound buffer. If this happens, playback is suspended, though it automatically resumes once more sound data has been loaded.

To find out if playback is suspended because AIR is waiting for data to load, use the `Sound.isBuffering` property.

### Working with dynamically generated audio

Instead of loading or streaming an existing sound, you can generate audio data dynamically. You can generate audio data when you assign an event listener for the `sampleData` event of a Sound object. (The `sampleData` event is defined in the SampleDataEvent class.) In this environment, the Sound object doesn’t load sound data from a file. Instead, it acts as a socket for sound data that is being streamed to it through the use of the function you assign to this event.
When you add a `sampleData` event listener to a `Sound` object, the object periodically requests data to add to the sound buffer. This buffer contains data for the `Sound` object to play. When you call the `play()` method of the `Sound` object, it dispatches the `sampleData` event when requesting new sound data. (This is true only when the `Sound` object has not loaded mp3 data from a file.)

The `SampleDataEvent` object includes a `data` property. In your event listener, you write `ByteArray` objects to this `data` object. The byte arrays you write to this object add to buffered sound data that the `Sound` object plays. The byte array in the buffer is a stream of floating-point values from -1 to 1. Each floating-point value represents the amplitude of one channel (left or right) of a sound sample. `Sound` is sampled at 44,100 samples per second. Each sample contains a left and right channel, interleaved as floating-point data in the byte array.

In your handler function, you use the `ByteArray.writeFloat()` method to write to the `data` property of the `sampleData` event. For example, the following code generates a sine wave:

```javascript
var mySound = new air.Sound();
mySound.addEventListener(air.SampleDataEvent.SAMPLE_DATA, sineWaveGenerator);
mySound.play();

function sineWaveGenerator(event)
{
    for (i = 0; i < 8192; i++)
    {
        var n = Math.sin((i + event.position) / Math.PI / 4);
        event.data.writeFloat(n);
        event.data.writeFloat(n);
    }
}
```

When you call `Sound.play()`, the application starts calling your event handler, requesting sound sample data. The application continues to send events as the sound plays back until you stop providing data, or until you call `SoundChannel.stop()`.

The latency of the event varies from platform to platform, and could change in future versions of AIR. Do not depend on a specific latency; calculate it instead. To calculate the latency, use the following formula:

\[(\text{SampleDataEvent.position} / 44.1) - \text{SoundChannelObject.position}\]

Provide from 2048 through 8192 samples to the `data` property of the `SampleDataEvent` object (for each call to the event listener). For best performance, provide as many samples as possible (up to 8192). The fewer samples you provide, the more likely it is that clicks and pops will occur during playback. This behavior can differ on various platforms and can occur in various situations—for example, when resizing the browser. Code that works on one platform when you provide only 2048 sample might not work as well when run on a different platform. If you require the lowest latency possible, consider making the amount of data user-selectable.

If you provide fewer than 2048 samples (per call to the `sampleData` event listener), the application stops after playing the remaining samples. It then dispatches a `SoundComplete` event.

### Modifying sound from mp3 data

You use the `Sound.extract()` method to extract data from a `Sound` object. You can use (and modify) that data to write to the dynamic stream of another `Sound` object for playback. For example, the following code uses the bytes of a loaded MP3 file and passes them through a filter function, `upOctave()`:

```javascript
```
var mySound = new air.Sound();
var sourceSnd = new air.Sound();
var urlReq = new air.URLRequest("test.mp3");
sourceSnd.load(urlReq);
sourceSnd.addEventListener(air.Event.COMPLETE, loaded);
function loaded(event)
{
    mySound.addEventListener(SampleDataEvent.SAMPLE_DATA, processSound);
    mySound.play();
}

function processSound(event)
{
    var bytes = new air.ByteArray();
    sourceSnd.extract(bytes, 8192);
    event.data.writeBytes(upOctave(bytes));
}

function upOctave(bytes)
{
    var returnBytes = new air.ByteArray();
    bytes.position = 0;
    while(bytes.bytesAvailable > 0)
    {
        returnBytes.writeFloat(bytes.readFloat());
        returnBytes.writeFloat(bytes.readFloat());
        if (bytes.bytesAvailable > 0)
        {
            bytes.position += 8;
        }
    }
    return returnBytes;
}

**Limitations on generated sounds**

When you use a `sampleData` event listener with a `Sound` object, the only other `Sound` methods that are enabled are `Sound.extract()` and `Sound.play()`. Calling any other methods or properties results in an exception. All methods and properties of the `SoundChannel` object are still enabled.

**Playing sounds**

Playing a loaded sound can be as simple as calling the `Sound.play()` method for a `Sound` object, as follows:

var req = new air.URLRequest("smallSound.mp3");
var snd = new air.Sound(req);
snd.play();

**Sound playback operations**

When playing back sounds, you can perform the following operations:

- Play a sound from a specific starting position
- Pause a sound and resume playback from the same position later
- Know exactly when a sound finishes playing
- Track the playback progress of a sound
- Change volume or panning while a sound plays

To perform these operations during playback, use the SoundChannel, SoundMixer, and SoundTransform classes.

The SoundChannel class controls the playback of a single sound. The SoundChannel.position property can be thought of as a playhead, indicating the current point in the sound data that’s being played.

When an application calls the Sound.play() method, a new instance of the SoundChannel class is created to control the playback.

Your application can play a sound from a specific starting position by passing that position, in terms of milliseconds, as the startTime parameter of the Sound.play() method. It can also specify a fixed number of times to repeat the sound in rapid succession by passing a numeric value in the loops parameter of the Sound.play() method.

When the Sound.play() method is called with both a startTime parameter and a loops parameter, the sound is played back repeatedly from the same starting point each time. The following code shows this:

```javascript
var req = new air.URLRequest("repeatingSound.mp3");
var snd = new air.Sound();
snd.play(1000, 3);
```

In this example, the sound is played from a point one second after the start of the sound, three times in succession.

**Pausing and resuming a sound**

If your application plays long sounds, like songs or podcasts, you probably want to let users pause and resume the playback of those sounds. A sound cannot literally be paused during playback; it can only be stopped. However, a sound can be played starting from any point. You can record the position of the sound at the time it was stopped, and then replay the sound starting at that position later.

For example, let’s say your code loads and plays a sound file like this:

```javascript
var req = new air.URLRequest("bigSound.mp3");
var snd = new air.Sound(req);
var channel = snd.play();
```

While the sound plays, the position property of the channel object indicates the point in the sound file that is currently being played. Your application can store the position value before stopping the sound from playing, as follows:

```javascript
var pausePosition = channel.position;
channel.stop();
```

To resume playing the sound, pass the previously stored position value to restart the sound from the same point it stopped at before.

```javascript
channel = snd.play(pausePosition);
```

**Monitoring playback**

Your application might want to know when a sound stops playing. Then it can start playing another sound or clean up some resources used during the previous playback. The SoundChannel class dispatches a soundComplete event when its sound finishes playing. Your application can listen for this event and take appropriate action, as the following example shows:
The SoundChannel class does not dispatch progress events during playback. To report on playback progress, your application can set up its own timing mechanism and track the position of the sound playhead.

To calculate what percentage of a sound has been played, you can divide the value of the SoundChannel.position property by the length of the sound data that’s being played:

```javascript
var playbackPercent = 100 * (channel.position / snd.length);
```

However, this code only reports accurate playback percentages if the sound data was fully loaded before playback began. The Sound.length property shows the size of the sound data that is currently loaded, not the eventual size of the entire sound file. To track the playback progress of a streaming sound that is still loading, your application should estimate the eventual size of the full sound file and use that value in its calculations. You can estimate the eventual length of the sound data using the bytesLoaded and bytesTotal properties of the Sound object, as follows:

```javascript
var estimatedLength = Math.ceil(snd.length / (snd.bytesLoaded / snd.bytesTotal));
var playbackPercent = 100 * (channel.position / estimatedLength);
```

The following code loads a larger sound file and uses the setInterval() function as its timing mechanism for showing playback progress. It periodically reports on the playback percentage, which is the current position value divided by the total length of the sound data:

```javascript
var snd = new air.Sound();
var url = "http://www.example.com/sounds/test.mp3";
var req = new air.URLRequest(url);
snd.load(req);
var channel = snd.play();
var timer = setInterval(monitorProgress, 100);
snd.addEventListener(air.Event.SOUND_COMPLETE, onPlaybackComplete);

function monitorProgress(event)
{
    var estimatedLength = Math.ceil(snd.length / (snd.bytesLoaded / snd.bytesTotal));
    var playbackPercent = Math.round(100 * (channel.position / estimatedLength));
    air.trace("Sound playback is " + playbackPercent + ">% complete.");
}

function onPlaybackComplete(event)
{
    air.trace("The sound has finished playing.");
    clearInterval(timer);
}
```

After the sound data starts loading, this code calls the snd.play() method and stores the resulting SoundChannel object in the channel variable. Then it adds a monitorProgress() method, which the setInterval() function calls repeatedly. The code uses an event listener to the SoundChannel object for the soundComplete event that occurs when playback is complete.
The `monitorProgress()` method estimates the total length of the sound file based on the amount of data that has already been loaded. It then calculates and displays the current playback percentage.

When the entire sound has been played, the `onPlaybackComplete()` function executes. This function removes the callback method for the `setInterval()` function, so that the application doesn’t display progress updates after playback is done.

**Stopping streaming sounds**

There is a quirk in the playback process for sounds that are streaming—that is, for sounds that are still loading while they are being played. When you call the `stop()` method on a SoundChannel instance that is playing back a streaming sound, the sound playback stops and then it restarts from the beginning of the sound. This occurs because the sound loading process is still underway. To stop both the loading and the playback of a streaming sound, call the `Sound.close()` method.

**Controlling sound volume and panning**

An individual SoundChannel object controls both the left and the right stereo channels for a sound. If an MP3 sound is a monaural sound, the left and right stereo channels of the SoundChannel object contain identical waveforms.

You can find out the amplitude of each stereo channel of the sound being played using the `leftPeak` and `rightPeak` properties of the SoundChannel object. These properties show the peak amplitude of the sound waveform itself. They do not represent the actual playback volume. The actual playback volume is a function of the amplitude of the sound wave and the volume values set in the SoundChannel object and the SoundMixer class.

The pan property of a SoundChannel object can be used to specify a different volume level for each of the left and right channels during playback. The pan property can have a value ranging from -1 to 1. A value of -1 means the left channel plays at top volume while the right channel is silent. A value of 1 means the right channel plays at top volume while the left channel is silent. Values in between -1 and 1 set proportional values for the left and right channel values. A value of 0 means that both channels play at a balanced, mid-volume level.

The following code example creates a SoundTransform object with a volume value of 0.6 and a pan value of -1 (top left channel volume and no right channel volume). It passes the SoundTransform object as a parameter to the `play()` method. The `play()` method applies that SoundTransform object to the new SoundChannel object that is created to control the playback.

```javascript
var req = new air.URLRequest("bigSound.mp3");
var snd = new air.Sound(req);
var trans = new air.SoundTransform(0.6, -1);
var channel = snd.play(0, 1, trans);
```

You can alter the volume and panning while a sound plays. Set the `pan` or `volume` properties of a SoundTransform object and then apply that object as the `soundTransform` property of a SoundChannel object.

You can also set global volume and pan values for all sounds at once, using the `soundTransform` property of the SoundMixer class. The following example shows this:

```javascript
SoundMixer.soundTransform = new air.SoundTransform(1, -1);
```

You can also use a SoundTransform object to set volume and pan values for a Microphone object (see “Capturing sound input” on page 312).

The following example alternates the panning of the sound from the left channel to the right channel and back while the sound plays:
This code starts by loading a sound file and then creating a SoundTransform object with volume set to 1 (full volume) and pan set to 0 (evenly balanced between left and right). Then it calls the `snd.play()` method, passing the SoundTransform object as a parameter.

While the sound plays, the `panner()` method executes repeatedly. The `panner()` method uses the `Math.sin()` function to generate a value between -1 and 1. This range corresponds to the acceptable values of the `SoundTransform.pan` property. The SoundTransform object’s `pan` property is set to the new value, and then the channel’s `soundTransform` property is set to use the altered SoundTransform object.

To run this example, replace the filename `bigSound.mp3` with the name of a local MP3 file. Then run the example. You should hear the left channel volume getting louder while the right channel volume gets softer, and vice versa.

In this example, the same effect could be achieved by setting the `soundTransform` property of the SoundMixer class. However, that would affect the panning of all sounds currently playing, not just the single sound this SoundChannel object plays.

### Working with sound metadata

Sound files that use the MP3 format can contain additional data about the sound in the form of ID3 tags.

Not every MP3 file contains ID3 metadata. When a Sound object loads an MP3 sound file, it dispatches an `Event.ID3` event if the sound file contains ID3 metadata. To prevent run-time errors, your application should wait to receive the `Event.ID3` event before accessing the `Sound.id3` property for a loaded sound.

The following code shows how to recognize when the ID3 metadata for a sound file has been loaded:
var s = new air.Sound();
s.addEventListener(air.Event.ID3, onID3InfoReceived);
var urlReq = new air.URLRequest("mySound.mp3");
s.load(urlReq);

function onID3InfoReceived(event)
{
    var id3 = event.target.id3;

    air.trace("Received ID3 Info:");
    for (propName in id3)
    {
        air.trace(propName + " = " + id3[propName]);
    }
}

This code starts by creating a Sound object and telling it to listen for the id3 event. When the sound file’s ID3 metadata is loaded, the onID3InfoReceived() method is called. The target of the Event object that is passed to the onID3InfoReceived() method is the original Sound object. The method then gets the Sound object’s id3 property and iterates through its named properties to trace their values.

## Accessing raw sound data

The SoundMixer.computeSpectrum() method lets an application read the raw sound data for the waveform that is currently being played. If more than one SoundChannel object is currently playing, the SoundMixer.computeSpectrum() method shows the combined sound data of every SoundChannel object mixed together.

### How sound data is returned

The sound data is returned as a ByteArray object containing 512 four-byte sets of data, each of which represents a floating point value between -1 and 1. These values represent the amplitude of the points in the sound waveform being played. The values are delivered in two groups of 256, the first group for the left stereo channel and the second group for the right stereo channel.

The SoundMixer.computeSpectrum() method returns frequency spectrum data rather than waveform data if the FFTMode parameter is set to true. The frequency spectrum shows amplitude arranged by sound frequency, from lowest frequency to highest. A Fast Fourier Transform (FFT) is used to convert the waveform data into frequency spectrum data. The resulting frequency spectrum values range from 0 to roughly 1.414 (the square root of 2).
The following diagram compares the data returned from the `computeSpectrum()` method when the `FFTMode` parameter is set to `true` and when it is set to `false`. The sound used for this diagram contains a loud bass sound in the left channel and a drum hit sound in the right channel.

The `computeSpectrum()` method can also return data that has been resampled at a lower bit rate. Generally, this results in smoother waveform data or frequency data at the expense of detail. The `stretchFactor` parameter controls the rate at which the `computeSpectrum()` method data is sampled. When the `stretchFactor` parameter is set to 0, the default, the sound data is sampled at a rate of 44.1 kHz. The rate is halved at each successive value of the `stretchFactor` parameter. So a value of 1 specifies a rate of 22.05 kHz, a value of 2 specifies a rate of 11.025 kHz, and so on. The `computeSpectrum()` method still returns 256 floating point values per stereo channel when a higher `stretchFactor` value is used.

**Building a simple sound visualizer**

The following example uses the `SoundMixer.computeSpectrum()` method to show a chart of the sound waveform that animates periodically:
const PLOT_WIDTH = 600;
const CHANNEL_LENGTH = 256;

var snd = new air.Sound();
var req = new air.URLRequest("test.mp3");
var bytes = new air.ByteArray();
var divStyles = new Array;

/**
 * Initializes the application. It draws 256 DIV elements to the document body,
 * and sets up a divStyles array that contains references to the style objects of
 * each DIV element. It then calls the playSound() function.
 */
function init()
{
    var div;
    for (i = 0; i < CHANNEL_LENGTH; i++)
    {
        div = document.createElement("div");
        div.style.height = "1px";
        div.style.width = "0px";
        div.style.backgroundColor = "blue";
        document.body.appendChild(div);
        divStyles[i] = div.style;
    }
    playSound();
}

/**
 * Plays a sound, and calls setInterval() to call the setMeter() function
 * periodically, to display the sound spectrum data.
 */
function playSound()
{
    if (snd.url !== null)
    {
        snd.close();
    }
    snd.load(req);
    var channel = snd.play();
timer = setInterval(setMeter, 100);
snd.addEventListener(air.Event.SOUND_COMPLETE, onPlaybackComplete);
}

/**
 * Computes the width of each of the 256 colored DIV tags in the document,
 * based on data returned by the call to SoundMixer.computeSpectrum(). The
 * first 256 floating point numbers in the byte array represent the data from
 * the left channel, and then next 256 floating point numbers represent the
 * data from the right channel.
 */
function setMeter()
{
    air.SoundMixer.computeSpectrum(bytes, false, 0);
var n;
for (var i = 0; i < CHANNEL_LENGTH; i++)
{
    bytes.position = i * 4;
    n = Math.abs(bytes.readFloat());
    bytes.position = 256*4 + i * 4;
    n += Math.abs(bytes.readFloat());
    divStyles[i].width = n * PLOT_WIDTH;
}

/**
 * When the sound is done playing, remove the intermediate process
 * started by setInterval().
 */
function onPlaybackComplete(event)
{
    clearInterval(interval);
}
</script>
<body onload="init()">
</body>
</html>

This example first loads and plays a sound file and then uses the `setInterval()` function to monitor the `SoundMixer.computeSpectrum()` method, which stores the sound wave data in the bytes `ByteArray` object.

The sound waveform is plotted by setting the width of `div` elements representing a bar graph.

## Capturing sound input

The Microphone class lets your application connect to a microphone or other sound input device on the user’s system.

An application can broadcast the input audio to that system’s speakers or send the audio data to a remote server, such as the Flash Media Server. You cannot access raw audio data from the microphone; you can only send audio to the system’s speakers or send compressed audio data to a remote server. You can use either Speex or Nellymoser codec for data sent to a remote server. (The Speex codec is available in AIR 1.5.)

### Accessing a microphone

The Microphone class does not have a constructor method. Instead, you use the static `Microphone.getMicrophone()` method to obtain a new Microphone instance, as the following example shows:

```javascript
var mic = air.Microphone.getMicrophone();
```

Calling the `Microphone.getMicrophone()` method without a parameter returns the first sound input device discovered on the user’s system.

A system can have more than one sound input device attached to it. Your application can use the `Microphone.names` property to get an array of the names of all available sound input devices. Then it can call the `Microphone.getMicrophone()` method with an index parameter that matches the index value of a device’s name in the array.
A system might not have a microphone or other sound input device attached to it. You can use the `Microphone.names` property or the `Microphone.getMicrophone()` method to check whether the user has a sound input device installed. If the user doesn’t have a sound input device installed, the names array has a length of zero, and the `getMicrophone()` method returns a value of `null`.

### Routing microphone audio to local speakers

Audio input from a microphone can be routed to the local system speakers by calling the `Microphone.setLoopback()` method with a parameter value of `true`.

When sound from a local microphone is routed to local speakers, there is a risk of creating an audio feedback loop. This can cause loud squealing sounds and can potentially damage sound hardware. Calling the `Microphone.setUseEchoSuppression()` method with a parameter value of `true` reduces, but does not completely eliminate, the risk that audio feedback will occur. Adobe recommends that you always call `Microphone.setUseEchoSuppression(true)` before calling `Microphone.setLoopback(true)`, unless you are certain that the user is playing back the sound using headphones or something other than speakers.

The following code shows how to route the audio from a local microphone to the local system speakers:

```javascript
var mic = air.Microphone.getMicrophone();
mic.setUseEchoSuppression(true);
mic.setLoopBack(true);
```

### Altering microphone audio

Your application can alter the audio data that comes from a microphone in two ways. First, it can change the gain of the input sound, which effectively multiplies the input values by a specified amount. This creates a louder or quieter sound. The `Microphone.gain` property accepts numeric values from 0 through 100. A value of 50 acts like a multiplier of one and specifies normal volume. A value of zero acts like a multiplier of zero and effectively silences the input audio. Values above 50 specify higher than normal volume.

Your application can also change the sample rate of the input audio. Higher sample rates increase sound quality, but they also create denser data streams that use more resources for transmission and storage. The `Microphone.rate` property represents the audio sample rate measured in kilohertz (kHz). The default sample rate is 8 kHz. You can set the `Microphone.rate` property to a value higher than 8 kHz if your microphone supports the higher rate. For example, setting the `Microphone.rate` property to 11 sets the sample rate to 11 kHz; setting it to 22 sets the sample rate to 22 kHz, and so on. The sample rates available depend on the selected codec. When you use the Nellymoser codec, you can specify 5, 8, 11, 16, 22 and 44 kHz as the sample rate. When you use Speex codec (available in AIR 1.5), you can only use 16 kHz.

### Detecting microphone activity

To conserve bandwidth and processing resources, the runtime tries to detect when a microphone transmits no sound. When the microphone’s activity level stays below the silence level threshold for a period of time, the runtime stops transmitting the audio input and dispatches an `activity` event. If you use the Speex codec (available in AIR 1.5), set the silence level to 0, to ensure that the application continuously transmits audio data. Speex voice activity detection automatically reduces bandwidth.

Three properties of the `Microphone` class monitor and control the detection of activity:

- The read-only `activityLevel` property indicates the amount of sound the microphone is detecting, on a scale from 0 to 100.
The `silenceLevel` property specifies the amount of sound needed to activate the microphone and dispatch an `activity` event. The `silenceLevel` property also uses a scale from 0 to 100, and the default value is 10.

The `silenceTimeout` property describes the number of milliseconds that the activity level must stay below the silence level before an `activity` event is dispatched. The default `silenceTimeout` value is 2000.

Both the `Microphone.silenceLevel` property and the `Microphone.silenceTimeout` property are read only, but their values can be changed by using the `Microphone.setSilenceLevel()` method.

In some cases, the process of activating the microphone when new activity is detected can cause a short delay. Keeping the microphone active at all times can remove such activation delays. Your application can call the `Microphone.setSilenceLevel()` method with the `silenceLevel` parameter set to zero. This keeps the microphone active and gathering audio data, even when no sound is detected. Conversely, setting the `silenceLevel` parameter to 100 prevents the microphone from being activated at all.

The following example displays information about the microphone and reports on `activity` events and `status` events dispatched by a `Microphone` object:

```javascript
var deviceArray = air.Microphone.names;
air.trace("Available sound input devices:");
for (i = 0; i < deviceArray.length; i++)
{
    air.trace("   " + deviceArray[i]);
}

var mic = air.Microphone.getMicrophone();
mic.gain = 60;
mic.rate = 11;
mic.setUseEchoSuppression(true);
mic.setLoopBack(true);
mic.setSilenceLevel(5, 1000);

mic.addEventListener(air.ActivityEvent.ACTIVITY, this.onMicActivity);

var micDetails = "Sound input device name: " + mic.name + '\n';
micDetails += "Gain: " + mic.gain + '\n';
micDetails += "Rate: " + mic.rate + " kHz" + '\n';
micDetails += "Muted: " + mic.muted + '\n';
micDetails += "Silence level: " + mic.silenceLevel + '\n';
micDetails += "Silence timeout: " + mic.silenceTimeout + '\n';
micDetails += "Echo suppression: " + mic.useEchoSuppression + '\n';
air.trace(micDetails);

function onMicActivity(event)
{
    air.trace("activating=" + event.activating + ", activityLevel=" +
    microactivityLevel);
}
```

When you run the preceding example, speak or make noises into your system microphone and watch the resulting trace statements appear in the console.

### Sending audio to and from a media server

Additional audio capabilities are available when using a streaming media server such as Flash Media Server.
In particular, your application can attach a Microphone object to a runtime.flash.net.NetStream object and transmit data directly from the user's microphone to the server. Audio data can also be streamed from the server to an AIR application.

AIR 1.5 introduces support for the Speex codec. To set the codec used for compressed audio sent to the media server, set the `codec` property of the Microphone object. This property can have two values, which are enumerated in the SoundCodec class. Setting the codec property to `SoundCodec.SPEEX` selects the Speex codec for compressing audio. Setting the property to `SoundCodec.NELLYMOSE` (the default) selects the Nellymoser codec for compressing audio.

For more information, see the Flash Media Server documentation online at http://www.adobe.com/support/documentation.
Chapter 29: Using digital rights management

Adobe® Flash® Media Rights Management Server (FMRMS) provides media publishers the ability to distribute content, specifically FLV and MP4 files, and to recuperate production costs through direct (user-paid) or indirect (advertising-paid) compensation by their consumers. The publishers distribute media as encrypted FLVs that can be downloaded and played in Adobe® Media Player™, or any Adobe® AIR® application that makes use of the digital rights management (DRM) API.

With FMRMS, the content providers can use identity-based licensing to protect their content with user credentials. For example, a consumer wants to view a television program, but does not want to watch the accompanying advertisements. To avoid watching the advertisements, the consumer registers and pays the content publisher a premium. The user can then use their authentication credential to gain access and play the program without the commercials. Another consumer may want to view the content offline while traveling with no internet access. After registering and paying the content publisher for the premium service, the user’s authentication credential allows them to access and download the program from the publisher’s website. The user can then view the content offline during the permitted period. The content is also protected by the user credentials and cannot be shared with other users.

When a user tries to play a DRM-encrypted file, the application contacts the FMRMS which in turn contacts the content publisher’s system through their service provider interface (SPI) to authenticate the user and retrieve the license, a voucher that determines whether the user is allowed access to the content and, if so, for how long. The voucher also determines whether the user can access the content offline and, if so, for how long. As such, user credentials are needed to determine access to the encrypted content.

Identity-based licensing also supports anonymous access. For example, anonymous access can be used by the provider to distribute ad-supported content or to allow free access to the current content for a specified number of days. The archive material might be considered premium content that must be paid for and requires user credentials. The content provider can also specify and restrict the type and version of the player needed for their content.

How to enable your AIR application to play content protected with digital rights management encryption is described here. It is not necessary to understand how to encrypt content using DRM, but it is assumed that you have access to DRM-encrypted content and are communicating with FMRMS to authenticate the user and retrieve the voucher.

For an overview of FMRMS, including creating policies, see the documentation included with FMRMS.

For information on Adobe Media Player, see Adobe Media Player Help available within Adobe Media Player.

Additional online information about digital rights management

You can find more information about digital rights management from these sources:

Language Reference

Adobe Developer Connection Articles and Samples

- Adobe AIR Developer Connection for HTML and Ajax (search for ‘digital rights management’ or ‘drm’)
Understanding the encrypted FLV workflow

There are four types of events, StatusEvent, DRMAuthenticateEvent, DRMErrorEvent, and DRMStatusEvent, that may be dispatched when an AIR application attempts to play a DRM-encrypted file. To support these files, the application should add event listeners for handling the DRM events.

The following is the workflow of how the AIR application can retrieve and play the content protected with DRM-encryption:

1. The application, using a NetStream object, attempts to play an FLV or MP4 file. If the content is encrypted, an events.StatusEvent event is dispatched with the code, DRM.encryptedFLV, indicating the FLV is encrypted.
   
   **Note:** If an application does not want to play the DRM-encrypted file, it can listen to the status event dispatched when it encounters an encrypted content, then let the user know that the file is not supported and close the connection.

2. If the file is anonymously encrypted, meaning that all users are allowed to view the content without inputting authentication credentials, the AIR application proceeds to the last step of this workflow. However, if the file requires an identity-based license, meaning that the user credential is required, then the NetStream object dispatches a DRMAuthenticateEvent object. The user must provide their authentication credentials before playback can begin.

3. The AIR application must provide some mechanism for gathering the necessary authentication credentials. The usernamePrompt, passwordPrompt, and urlPrompt properties of DRMAuthenticationEvent class, provided by the content server, can be used to instruct the end user with information about the data that is required. You can use these properties in constructing a user interface for retrieving the needed user credentials. For example, the usernamePrompt value string may state that the user name must be in the form of an e-mail address.
   
   **Note:** AIR does not supply a default user interface for gathering authentication credentials. The application developer must write the user interface and handle the DRMAuthenticateEvent events. If the application does not provide an event listener for DRMAuthenticateEvent objects, the DRM-encrypted object remains in a “waiting for credentials” state and the content is therefore not available.

4. Once the application obtains the user credentials, it passes the credentials with the setDRMAuthenticationCredentials() method to the NetStream object. This signals to the NetStream object that it should try authenticating the user at the next available opportunity. AIR then passes the credential to the FMRMS for authentication. If the user was authenticated, then the application proceeds to the next step.

   If authentication failed, the NetStream object dispatches a new DRMAuthenticateEvent object and the application returns to step 3. This process repeats indefinitely. The application should provide a mechanism to handle and limit the repeated authentication attempts. For example, the application could allow the user to cancel the attempt which can close the NetStream connection.

5. Once the user is authenticated, or if anonymous encryption is used, then the DRM subsystem retrieves the voucher. The voucher is used to check if the user is authorized to view the content. The information in the voucher can apply to both the authenticated and the anonymous users. For example, both the authenticated and anonymous users may have access to the content for a specified period of time before the content expires or they may not have access to the content because the content provider may not support the version of the viewing application.

   If an error has not occurred and the user was successfully authorized to view the content, The NetStream object dispatches a DRMStatusEvent object and the AIR application begins playback. The DRMStatusEvent object holds the related voucher information, which identifies the user’s policy and permissions. For example, it holds information regarding whether the content can be made available offline or when the voucher expires and the content can no longer be viewed. The application can use this data to inform the user of the status of their policy. For example, the application can display the number of remaining days the user has for viewing the content in a status bar.
If the user is allowed offline access, the voucher is cached and the encrypted content is downloaded to the user’s machine and made accessible for the duration defined in the offline lease period. The detail property in the event contains “DRM.voucherObtained”. The application decides where to store the content locally in order for it to be available offline. In AIR 1.5, you can also preload vouchers using the DRMManager class.

All DRM-related errors result in the application dispatching a DRMErrorEvent event object. AIR handles the authentication failures encountered when using the NetStream setDRMAuthenticationCredentials() method by re-dispatching the DRMAuthenticationEvent object. All other error events should be explicitly handled by the application. This includes cases where user inputs valid credentials, but the voucher protecting the encrypted content restricts the access to the content. For example, an authenticated user may still not have access to the content because the rights have not been paid for. This could also occur where two users, both registered members with the same media publisher, are attempting to share content that only one of the members has paid for. The application should inform the user of the error, such as the restrictions to the content, as well as provide an alternative, such as instructions in how to register and pay for the rights to view the content.

Preloading vouchers for offline playback

You can preload the vouchers required to play DRM-protected content. Preloaded vouchers allow users to view the content whether or not they have an active Internet connection. (The preload process itself obviously requires an Internet connection.) Use the NetStream class preloadEmbeddedMetadata() method and the AIR 1.5 DRMManager class to preload vouchers.

The following steps describe the workflow for preloading the voucher for a DRM-protected media file:

1. Download and store the media file. (DRM metadata can only be preloaded from locally stored files.)
2. Create the NetConnection and NetStream objects, supplying implementations for the onDRMContentData() and onPlayStatus() callback functions of the NetStream client object.
3. Create a NetStreamPlayOptions object and set the stream property to the URL of the local media file.
4. Call the NetStream preloadEmbeddedMetadata(), passing in the NetStreamPlayOptions object identifying the media file to parse.
5. If the media file contains DRM metadata, then the onDRMContentData() callback function is invoked. The metadata is passed to this function as a DRMContentData object.
6. Use the DRMContentData object to obtain the voucher using the DRMManager loadVoucher() method.

   If the value of the authenticationMethod property of the DRMContentData object is userNameAndPassword, then you must authenticate the user on the media rights server before loading the voucher. The serverURL and domain properties of the DRMContentData object can be passed to the DRMManager authenticate() method, along with the user’s credentials.

7. The onPlayStatus() callback function is invoked when file parsing is complete. If the onDRMContentData() function has not been called, then the file does not contain the metadata required to obtain a voucher (and may not be DRM-protected).

The following code example illustrates how to preload a DRM voucher for a local media file:
package
{
import flash.display.Sprite;
import flash.events.DRMAuthenticationCompleteEvent;
import flash.events.DRMAuthenticationErrorEvent;
import flash.events.DRMErrorEvent;
import flash.events.DRMStatusEvent;
import flash.net.NetConnection;
import flash.net.NetStream;
import flash.net.drm.AuthenticationMethod;
import flash.net.drm.DRMContentData;
import flash.net.drm.DRMManager;
import flash.net.drm.LoadVoucherSetting;
public class DRMPreloader extends Sprite
{
    private var videoURL:String = "app-storage:/video.flv";
    private var userName:String = "user";
    private var password:String = "password";
    private var preloadConnection:NetConnection;
    private var preloadStream:NetStream;
    private var drmManager:DRMManager = DRMManager.getDRMManager();
    private var drmContentData:DRMContentData;
    public function DRMPreloader():void {
        drmManager.addEventListener( DRMAuthenticationCompleteEvent.AUTHENTICATION_COMPLETE,
            onAuthenticationComplete );
        drmManager.addEventListener( DRMAuthenticationErrorEvent.AUTHENTICATION_ERROR,
            onAuthenticationError );
        drmManager.addEventListener(DRMStatusEvent.DRM_STATUS, onDRMStatus);
        drmManager.addEventListener(DRMErrorEvent.DRM_ERROR, onDRMError);
        preloadConnection = new NetConnection();
        preloadConnection.addEventListener(NetStatusEvent.NET_STATUS, onConnect);
        preloadConnection.connect(null);
    }
    private function onConnect( event:NetStatusEvent ):void {
        preloadMetadata();
    }
    private function preloadMetadata():void {
        preloadStream = new NetStream( preloadConnection );
        preloadStream.client = this;
        var options:NetStreamPlayOptions = new NetStreamPlayOptions();
        options.streamName = videoURL;
        preloadStream.preloadEmbeddedData( options );
    }
    public function onDRMContentData( drmMetadata:DRMContentData ):void {
        drmContentData = drmMetadata;
        if ( drmMetadata.authenticationMethod == AuthenticationMethod.USERNAME_AND_PASSWORD ) {
            authenticateUser();
        }
        else {

getVoucher();
}

private function getVoucher():void
{
    drmManager.loadVoucher( drmContentData, LoadVoucherSetting.ALLOW_SERVER );
}

private function authenticateUser():void
{
    drmManager.authenticate( drmContentData.serverURL, drmContentData.domain, userName, password );
}

private function onAuthenticationError( event:DRMAuthenticationErrorEvent ):void
{
    trace( "Authentication error: " + event.errorID + ", " + event.subErrorID );
}

private function onAuthenticationComplete( event:DRMAuthenticationCompleteEvent ):void
{
    trace( "Authenticated to: " + event.serverURL + ", domain: " + event.domain );
    getVoucher();
}

private function onDRMStatus( event:DRMStatusEvent ):void
{
    trace( "DRM Status: " + event.detail);
    trace( "--Voucher allows offline playback = " + event.isAvailableOffline );
    trace( "--Voucher already cached = " + event.isLocal );
    trace( "--Voucher required authentication = " + !event.isAnonymous );
}

private function onDRMError( event:DRMErrorEvent ):void
{
    trace( "DRM error event: " + event.errorID + ", " + event.subErrorID + ", " + event.text );
}

public function onPlayStatus( info:Object ):void
{
    preloadStream.close();
}

---

DRM-related members and events of the NetStream class

The NetStream class provides a one-way streaming connection between Flash Player or an AIR application, and either Flash Media Server or the local file system. (The NetStream class also supports progressive download.) A NetStream object is a channel within a NetConnection object. In an AIR application, the NetStream class dispatches four DRM-related events:
The NetStream class includes the following DRM-specific methods:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>drmAuthenticate</td>
<td>Defined in the DRMAuthenticateEvent class, this event is dispatched when a NetStream object tries to play a digital rights management (DRM) encrypted content that requires a user credential for authentication before play back. The properties of this event include header, usernamePrompt, passwordPrompt, and urlPrompt properties that can be used in obtaining and setting the user’s credentials. This event occurs repeatedly until the NetStream object receives valid user credentials.</td>
</tr>
<tr>
<td>drmError</td>
<td>Defined in the DRMErrorEvent class and dispatched when a NetStream object, trying to play a digital rights management (DRM) encrypted file, encounters a DRM-related error. For example, DRM error event object is dispatched when the user authorization fails. This may be because the user has not purchased the rights to view the content or because the content provider does not support the viewing application.</td>
</tr>
<tr>
<td>drmStatus</td>
<td>Defined in DRMStatusEvent class, is dispatched when the digital rights management (DRM) encrypted content begins playing (when the user is authenticated and authorized to play the content). The DRMStatusEvent object contains information related to the voucher, such as whether the content can be made available offline or when the voucher expires and the content can no longer be viewed.</td>
</tr>
<tr>
<td>status</td>
<td>Defined in events.StatusEvent and only dispatched when the application attempts to play content encrypted with digital rights management (DRM), by invoking the NetStream.play() method. The value of the status code property is &quot;DRM.encryptedFLV&quot;.</td>
</tr>
</tbody>
</table>

In addition, a NetStream object invokes the onDRMContentData() and onPlayStatus() callback functions as a result of a call to the preloadEmbeddedMetaData() method. The onDRMContentData() function is called when DRM metadata is encountered in a media file. The onPlayStatus() function is called when the file has been completely parsed. The onDRMContentData() and onPlayStatus() functions must be defined on the client object assigned to the NetStream instance. If you use the same NetStream object to preload vouchers and play content, you must wait for the onPlayStatus() call generated by preloadEmbeddedMetaData() before starting playback.

In the following code, username ("administrator"), password ("password") and the "drm" authentication type are set for authenticating the user. The setDRMAuthenticationCredentials() method must provide credentials that match credentials known and accepted by the content provider (the same user credentials that provided permission to view the content). The code for playing the video and making sure that a successful connection to the video stream has been made is not included here.
Using the DRMStatusEvent class

A NetStream object dispatches a DRMStatusEvent object when the content protected using digital rights management (DRM) begins playing successfully (when the voucher is verified, and when the user is authenticated and authorized to view the content). The DRMStatusEvent is also dispatched for anonymous users if they are permitted access. The voucher is checked to verify whether anonymous user, who do not require authentication, are allowed access to play the content. Anonymous users maybe denied access for a variety of reasons. For example, an anonymous user may not have access to the content because it has expired.

The DRMStatusEvent object contains information related to the voucher, such as whether the content can be made available offline or when the voucher expires and the content can no longer be viewed. The application can use this data to convey the user's policy status and its permissions.

DRMStatusEvent properties

The DRMStatusEvent class includes the following properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contentData</td>
<td>A DRMContentData object containing the DRM metadata embedded in the content.</td>
</tr>
<tr>
<td>detail</td>
<td>A string explaining the context of the status event. In DRM 1.0, the only valid value is DRM.voucherObtained.</td>
</tr>
<tr>
<td>isAnonymous</td>
<td>Indicates whether the content, protected with DRM encryption, is available without requiring a user to provide authentication credentials (true) or not (false). A false value means user must provide a username and password that matches the one known and expected by the content provider.</td>
</tr>
<tr>
<td>isAvailableOffline</td>
<td>Indicates whether the content, protected with DRM encryption, can be made available offline (true) or not (false). In order for digitally protected content to be available offline, its voucher must be cached to the user's local machine.</td>
</tr>
<tr>
<td>isLocal</td>
<td>Indicates whether the voucher needed to play the content is cached locally.</td>
</tr>
<tr>
<td>offlineLeasePeriod</td>
<td>The remaining number of days that content can be viewed offline.</td>
</tr>
<tr>
<td>policies</td>
<td>A custom object that may contain custom DRM properties.</td>
</tr>
<tr>
<td>voucher</td>
<td>The DRMVoucher.</td>
</tr>
<tr>
<td>voucherEndDate</td>
<td>The absolute date on which the voucher expires and the content is no longer viewable.</td>
</tr>
</tbody>
</table>

Creating a DRMStatusEvent handler

The following example creates an event handler that outputs the DRM content status information for the NetStream object that originated the event. Add this event handler to a NetStream object that points to DRM-encrypted content.

Using the DRMAuthenticateEvent class

The DRMAuthenticateEvent object is dispatched when a NetStream object tries to play a digital rights management (DRM) encrypted content that requires a user credential for authentication before play back.
The DRMAuthenticateEvent handler is responsible for gathering the required credentials (user name, password, and type) and passing the values to the `NetStream.setDRMAuthenticationCredentials()` method for validation. Each AIR application must provide some mechanism for obtaining user credentials. For example, the application could provide a user with a simple user interface to enter the username and password values, and optionally the type value as well. The AIR application should also provide a mechanism for handling and limiting the repeated authentication attempts.

**DRMAuthenticateEvent properties**

The DRMAuthenticateEvent class includes the following properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>authenticationType</td>
<td>Indicates whether the supplied credentials are for authenticating against the FMRMS (&quot;drm&quot;) or a proxy server (&quot;proxy&quot;). For example, the &quot;proxy&quot; option allows the application to authenticate against a proxy server if an enterprise requires such a step before the user can access the Internet. Unless anonymous authentication is used, after the proxy authentication, the user still must authenticate against the FMRMS in order to obtain the voucher and play the content. You can use <code>setDRMAuthenticationCredentials()</code> a second time, with &quot;drm&quot; option, to authenticate against the FMRMS.</td>
</tr>
<tr>
<td>header</td>
<td>The encrypted content file header provided by the server. It contains information about the context of the encrypted content.</td>
</tr>
<tr>
<td>netstream</td>
<td>The NetStream object that initiated this event.</td>
</tr>
<tr>
<td>passwordPrompt</td>
<td>A prompt for a password credential, provided by the server. The string can include instruction for the type of password required.</td>
</tr>
<tr>
<td>urlPrompt</td>
<td>A prompt for a URL string, provided by the server. The string can provide the location where the username and password is sent.</td>
</tr>
<tr>
<td>usernamePrompt</td>
<td>A prompt for a user name credential, provided by the server. The string can include instruction for the type of user name required. For example, a content provider may require an e-mail address as the user name.</td>
</tr>
</tbody>
</table>

**Creating a DRMAuthenticateEvent handler**

The following example creates an event handler that passes a set of hard-coded authentication credentials to the NetStream object that originated the event. (The code for playing the video and making sure that a successful connection to the video stream has been made is not included here.)

```javascript
var connection = new air.NetConnection();
connection.connect(null);

var videoStream = new air.NetStream();
videoStream.addEventListener(air.DRMAuthenticateEvent.DRM_AUTHENTICATE, drmAuthenticateEventHandler)

function drmAuthenticateEventHandler(event)
{
    videoStream.setDRMAuthenticationCredentials("administrator", "password", "drm");
}
```

**Creating an interface for retrieving user credentials**

In the case where DRM content requires user authentication, the AIR application usually needs to retrieve the user’s authentication credentials via a user interface.
Using the DRMErrorEvent class

AIR dispatches a DRMErrorEvent object when a NetStream object, trying to play a digital rights management (DRM) encrypted file, encounters a DRM related error. In the case of invalid user credentials, the DRMAuthenticateEvent object handles the error by repeatedly dispatching until the user enters valid credentials, or the AIR application denies further attempts. The application should listen to any other DRM error events to detect, identify, and handle the DRM-related errors.

If a user enters valid credentials, they still may not be allowed to view the encrypted content, depending on the terms of the DRM voucher. For example, if the user is attempting to view the content in an unauthorized application, that is, an application that is not validated by the publisher of the encrypted content. In this case, a DRMErrorEvent object is dispatched. The error events can also be fired if the content is corrupted or if the application’s version does not match what is specified by the voucher. The application must provide appropriate mechanism for handling errors.

**DRMErrorEvent properties**

The following table lists the errors that the DRMErrorEvent object reports:

<table>
<thead>
<tr>
<th>Major Error Code</th>
<th>Minor Error Code</th>
<th>Error Details</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>not used</td>
<td>User authentication failed.</td>
<td></td>
</tr>
<tr>
<td>1002</td>
<td>not used</td>
<td>Flash Media Rights Management Server (FMRMS) is not supporting Secure Sockets Layer (SSL).</td>
<td></td>
</tr>
<tr>
<td>1003</td>
<td>not used</td>
<td>The content has expired and is no longer available for viewing.</td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td>not used</td>
<td>User authorization failure. This can occur, for example, if the user has not purchased the content and therefore does not have the rights to view it.</td>
<td></td>
</tr>
<tr>
<td>1005</td>
<td>not used</td>
<td>Server URL Cannot connect to the server.</td>
<td></td>
</tr>
<tr>
<td>1006</td>
<td>not used</td>
<td>A client update is required, that is, Flash Media Rights Management Server (FMRMS) requires a new digital rights management (DRM) engine.</td>
<td></td>
</tr>
<tr>
<td>1007</td>
<td>not used</td>
<td>Generic internal failure.</td>
<td></td>
</tr>
<tr>
<td>1008</td>
<td>Detailed decrypting error code</td>
<td>An incorrect license key.</td>
<td></td>
</tr>
<tr>
<td>1009</td>
<td>not used</td>
<td>FLV content is corrupted.</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>not used</td>
<td>publisherID:applicationID The ID of the viewing application does not match a valid ID supported by the content publisher.</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>not used</td>
<td>Application version does not match what is specified in the policy.</td>
<td></td>
</tr>
<tr>
<td>1012</td>
<td>not used</td>
<td>Verification of the voucher associated with the encrypted content failed, indicating that the content may be corrupted.</td>
<td></td>
</tr>
<tr>
<td>1013</td>
<td>not used</td>
<td>The voucher associated with the encrypted content could not be saved.</td>
<td></td>
</tr>
<tr>
<td>1014</td>
<td>not used</td>
<td>Verification of the FLV header integrity failed, indicating that the content may be corrupted.</td>
<td></td>
</tr>
<tr>
<td>Major Error Code</td>
<td>Minor Error ID</td>
<td>Error Details</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>3300</td>
<td>Adobe Policy Server error code</td>
<td>The application detected an invalid voucher associated with the content.</td>
<td></td>
</tr>
<tr>
<td>3301</td>
<td>not used</td>
<td>User authentication failed.</td>
<td></td>
</tr>
<tr>
<td>3302</td>
<td>not used</td>
<td>Secure Sockets Layer (SSL) is not supported by the Flash Media Rights Management Server (FMRMS).</td>
<td></td>
</tr>
<tr>
<td>3303</td>
<td>not used</td>
<td>The content has expired and is no longer available for viewing.</td>
<td></td>
</tr>
<tr>
<td>3304</td>
<td>not used</td>
<td>User authorization failure. This can occur even if the user is authenticated, for example, if the user has not purchased the rights to view the content.</td>
<td></td>
</tr>
<tr>
<td>3305</td>
<td>not used</td>
<td>Cannot connect to the server.</td>
<td></td>
</tr>
<tr>
<td>3306</td>
<td>not used</td>
<td>A client update is required, that is, Flash Media Rights Management Server (FMRMS) requires a new digital rights management client engine.</td>
<td></td>
</tr>
<tr>
<td>3307</td>
<td>not used</td>
<td>Generic internal digital rights management failure.</td>
<td></td>
</tr>
<tr>
<td>3308</td>
<td>Detailed decrypting error code</td>
<td>An incorrect license key.</td>
<td></td>
</tr>
<tr>
<td>3309</td>
<td>not used</td>
<td>Flash video content is corrupted.</td>
<td></td>
</tr>
<tr>
<td>3310</td>
<td>not used</td>
<td>The ID of the viewing application does not match a valid ID supported by the content publisher. In other words, the viewing application is not supported by the content provider.</td>
<td></td>
</tr>
<tr>
<td>3311</td>
<td>not used</td>
<td>Application version does not match what is specified in the voucher.</td>
<td></td>
</tr>
<tr>
<td>3312</td>
<td>not used</td>
<td>Verification of the voucher associated with the encrypted content failed, indicating that the content may be corrupted.</td>
<td></td>
</tr>
<tr>
<td>3313</td>
<td>not used</td>
<td>The voucher associated with the encrypted content could not be saved to Microsafe.</td>
<td></td>
</tr>
<tr>
<td>3314</td>
<td>not used</td>
<td>Verification of the FLV header integrity failed, indicating that the content may be corrupted.</td>
<td></td>
</tr>
<tr>
<td>3315</td>
<td>not used</td>
<td>Remote playback of the DRM protected content is not allowed.</td>
<td></td>
</tr>
<tr>
<td>3316</td>
<td>not used</td>
<td>Missing AdobeCP module.</td>
<td></td>
</tr>
<tr>
<td>3317</td>
<td>not used</td>
<td>Load AdobeCP module failed.</td>
<td></td>
</tr>
<tr>
<td>3318</td>
<td>not used</td>
<td>Incompatible AdobeCP Version found.</td>
<td></td>
</tr>
<tr>
<td>3319</td>
<td>not used</td>
<td>Missing AdobeCP API entry point.</td>
<td></td>
</tr>
<tr>
<td>3320</td>
<td>not used</td>
<td>AdobeCP module is not authenticated.</td>
<td></td>
</tr>
</tbody>
</table>
Creating a DRMErroEvent handler

The following example creates an event handler for the NetStream object that originated the event. It is called when the NetStream encounters an error while attempting to play the DRM-encrypted content. Normally, when an application encounters an error, it performs any number of clean-up tasks, informs the user of the error, and provides options for solving the problem.

```javascript
function drmErrorEventHandler(event)
{
   air.trace(event.toString());
}
```

Using the DRMManager class

Use the DRMManager class to manage vouchers and media rights server sessions in an AIR application. The DRMManager class is available in AIR version 1.5 or higher.

Voucher management

Whenever a user plays an online DRM-protected media file, AIR obtains and caches the license voucher required to view the content. If the application saves the file locally, and the voucher allows offline playback, then the user can view the content even if a connection to the media rights server is not available. Using the DRMManager and the NetStream `preloadEmbeddedMetadata()` method, you can pre-cache the voucher so that the application does not have to initiate playback to obtain the license necessary to view the content. For example, your application could download the media file and then obtain the voucher while the user is still online.

To preload a voucher, use the NetStream `preloadEmbeddedMetadata()` method to obtain a DRMContentData object. The DRMContentData object contains the URL and domain of the media rights server that can provide the license and describes whether user authentication is required. With this information, you can call the DRMManager `loadVoucher()` method to obtain and cache the voucher. The workflow for preloading vouchers is described in more detail in “Preloading vouchers for offline playback” on page 318.

Session management

You can also use the DRMManager to authenticate the user to a media rights server and to manage persistent sessions. Call the DRMManager `authenticate()` method to establish a session with the media rights server. When authentication is completed successfully, the DRMManager dispatches a DRMAuthenticationCompleteEvent object. This object contains a session token. You can save this token to establish future sessions so that the user does not have to provide their account credentials. Pass the token to the `setAuthenticationToken()` method to establish a new authenticated session. (Token expiration, and other attributes, are determined by the settings of the server which generates the token. The token data structure is not intended to be interpreted by AIR application code and may change in future AIR updates.)

Authentication tokens can be transferred to other computers. To protect tokens, you can store them in the AIR Encrypted Local Store. See “Storing encrypted data” on page 290 for more information.

DRMStatus Events

The DRMManager dispatches a DRMStatusEvent object when after a call to the `loadVoucher()` method completes successfully.

If a voucher is obtained, then the detail property of the event object will have the value: “DRM.voucherObtained”, and the `voucher` property will contain the DRMVoucher object.
If a voucher is not obtained, then the detail property still has the value: "DRM.voucherObtained"; however, the voucher property is null. A voucher may not be obtained if, for example, you use the LoadVoucherSetting of localOnly and there is no locally cached voucher.

If the loadVoucher() call does not complete successfully, perhaps because of an authentication or communication error, then the DRMManager dispatches a DRMErrorEvent object instead.

**DRMAuthenticationComplete events**

The DRMManager dispatches a DRMAuthenticationCompleteEvent object when a user is successfully authenticated through a call to the authenticate() method.

In AIR 1.5, the DRMAuthenticationCompleteEvent object contains a reusable token that can be used to persist user authentication across application sessions. Pass this token to DRMManager setAuthenticationToken() method to re-establish the session. (Token attributes, such as expiration, are set by the token creator. AIR does not provide an API for examining token attributes.)

**DRMAuthenticationError events**

The DRMManager dispatches a DRMAuthenticationErrorEvent object when a user cannot be successfully authenticated through a call to the authenticate() or setAuthenticationToken() methods.

**Using the DRMContentData class**

The DRMContentData object contains the metadata properties of a DRM-protected media file. The DRMContentData properties contain the information necessary to obtain a license voucher for viewing the content.
Chapter 30: Application launching and exit options

This section discusses options and considerations for launching an installed Adobe® AIR® application, as well as options and considerations for closing a running application.

Additional online information about launch and exit options

You can find more information about the APIs used when launching and closing applications from these sources:

Quick Starts (Adobe AIR Developer Connection)
- Startup Options

Language Reference
- NativeApplication
- InvokeEvent
- InvokeEventReason
- BrowserInvokeEvent

Adobe Developer Connection Articles and Samples
- Adobe AIR Developer Connection for HTML and Ajax

Application invocation

An AIR application is invoked when the user (or the operating system):
- Launches the application from the desktop shell.
- Uses the application as a command on a command line shell.
- Opens a type of file for which the application is the default opening application.
- (Mac OS X) clicks the application icon in the dock taskbar (whether or not the application is currently running).
- Chooses to launch the application from the installer (either at the end of a new installation process, or after double-clicking the AIR file for an already installed application).
- Begins an update of an AIR application when the installed version has signaled that it is handling application updates itself (by including a `<customUpdateUI>true</customUpdateUI>` declaration in the application descriptor file).
Visits a web page hosting a Flash badge or application that calls `com.adobe.air.AIR launchApplication()` method specifying the identifying information for the AIR application. (The application descriptor must also include a `<allowBrowserInvocation>true</allowBrowserInvocation>` declaration for browser invocation to succeed.) See “Launching an installed AIR application from the browser” on page 367.

Whenever an AIR application is invoked, AIR dispatches an InvokeEvent object of type `invoke` through the singleton `NativeApplication` object. To allow an application time to initialize itself and register an event listener, `invoke` events are queued instead of discarded. As soon as a listener is registered, all the queued events are delivered.

**Note:** When an application is invoked using the browser invocation feature, the `NativeApplication` object only dispatches an `invoke` event if the application is not already running. See “Launching an installed AIR application from the browser” on page 367.

To receive `invoke` events, call the `addEventListener()` method of the `NativeApplication` object (`NativeApplication.nativeApplication`). When an event listener registers for an `invoke` event, it also receives all `invoke` events that occurred before the registration. Queued `invoke` events are dispatched one at a time on a short interval after the call to `addEventListener()` returns. If a new `invoke` event occurs during this process, it may be dispatched before one or more of the queued events. This event queuing allows you to handle any `invoke` events that have occurred before your initialization code executes. Keep in mind that if you add an event listener later in execution (after application initialization), it will still receive all `invoke` events that have occurred since the application started.

Only one instance of an AIR application is started. When an already running application is invoked again, AIR dispatches a new `invoke` event to the running instance. It is the responsibility of an AIR application to respond to an `invoke` event and take the appropriate action (such as opening a new document window).

An `InvokeEvent` object contains any arguments passed to the application, as well as the directory from which the application has been invoked. The `InvokeEvent.arguments` property contains an array of the arguments passed by the operating system when an AIR application is 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.

When multiple files are opened in one operation a single `InvokeEvent` object is dispatched on Mac OS X. Each file is included in the `arguments` array. On Windows and Linux, a separate `InvokeEvent` object is dispatched for each file.

Your application can handle `invoke` events by registering a listener with its `NativeApplication` object:

```java
NativeApplication.nativeApplication.addEventListener(InvokeEvent.INVOKE, onInvokeEvent);
air.NativeApplication.nativeApplication.addEventListener(air.InvokeEvent.INVOKE, onInvokeEvent);
```

And defining an event listener:

```java
var arguments;
var currentDir;
function onInvokeEvent(invocation) {
    arguments = invocation.arguments;
    currentDir = invocation.currentDirectory;
}
```

## Capturing command line arguments

The command line arguments associated with the invocation of an AIR application are delivered in the `invoke` event dispatched by the `NativeApplication` object. The `InvokeEvent.arguments` property contains an array of the arguments passed by the operating system when an AIR application is invoked. If the arguments contain relative file paths, you can typically resolve the paths using the `currentDirectory` property.
The arguments passed to an AIR program are treated as white-space delimited strings, unless enclosed in double quotes:

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>tick tock</td>
<td>(tick,tick)</td>
</tr>
<tr>
<td>&quot;tick tock&quot;</td>
<td>(tick,tick)</td>
</tr>
<tr>
<td>&quot;tick tock&quot;</td>
<td>(tick,tick)</td>
</tr>
<tr>
<td>&quot;tick&quot;&quot;tock&quot;</td>
<td>(&quot;tick&quot;,&quot;tock&quot;)</td>
</tr>
</tbody>
</table>

The `InvokeEvent.currentDirectory` property contains a File object representing the directory from which the application was launched.

When an application is invoked because a file of a type registered by the application is opened, the native path to the file is included in the command line arguments as a string. (Your application is responsible for opening or performing the intended operation on the file.) Likewise, when an application is programmed to update itself (rather than relying on the standard AIR update user interface), the native path to the AIR file is included when a user double-clicks an AIR file containing an application with a matching application ID.

You can access the file using the `resolve()` method of the `currentDirectory` File object:

```javascript
if((invokeEvent.currentDirectory != null)&&(invokeEvent.arguments.length > 0)){
  dir = invokeEvent.currentDirectory;
  fileToOpen = dir.resolvePath(invokeEvent.arguments[0]);
}
```

You should also validate that an argument is indeed a path to a file.

**Example: Invocation event log**

The following example demonstrates how to register listeners for and handle the `invoke` event. The example logs all the invocation events received and displays the current directory and command line arguments.

*Note: This example uses the AIRAliases.js file, which you can find in the frameworks folder of the SDK.*

```html
<html>
<head>
<title>Invocation Event Log</title>
<script src="AIRAliases.js" />
<script type="text/javascript">
function appLoad() {
  air.trace("Invocation Event Log.");
  air.NativeApplication.nativeApplication.addEventListener(
    air.InvokeEvent.INVOKE, onInvoke);
}

function onInvoke(invokeEvent) {
  logEvent("Invoke event received.");
  if (invokeEvent.currentDirectory) {
    logEvent("Current directory=" + invokeEvent.currentDirectory.nativePath);
  } else {
    logEvent("--no directory information available--");
  }

  if (invokeEvent.arguments.length > 0) {
```
logEvent("Arguments: " + invokeEvent.arguments.toString());
} else {
    logEvent("--no arguments--");
}
}

function logEvent(message) {
    var logger = document.getElementById('log');
    var line = document.createElement('p');
    line.innerHTML = message;
    logger.appendChild(line);
    air.trace(message);
}

window.unload = function() {
    air.NativeApplication.nativeApplication.removeEventListener(
        air.InvokeEvent.INVOKE, onInvoke);
}
</script>
</head>
<body onLoad="appLoad();">
    <div id="log" />
</body>
</html>

Launching on login

An AIR application can be set to launch automatically when the current user logs in by setting the NativeApplication
startAtLogin property to true. Once set, the application automatically starts whenever the user logs in. It continues
to launch at login until the setting is changed to false, the user manually changes the setting through the operating
system, or the application is uninstalled. Launching at login is a run-time setting. The setting only applies to the
current user. The application must be installed to successfully set the startAtLogin property to true. An error is
thrown if the property is set when an application is not installed (such as when it is launched with ADL).

Note: The application does not launch when the computer system starts. It launches when the user logs in.

To determine whether an application has launched automatically or as a result of a user action, you can examine the
reason property of the InvokeEvent object. If the property is equal to InvokeEventReason.LOGIN, then the
application started automatically. For any other invocation path, the reason property equals
InvokeEventReason.STANDARD. To access the reason property, your application must target AIR 1.5.1 (by setting
the correct namespace value in the application descriptor file).

The following, simplified application uses the InvokeEvent reason property to decide how to behave when an invoke
event occurs. If the reason property is "login", then the application remains in the background. Otherwise, it makes the
main application visible. An application using this pattern typically starts at login so that it can carry out background
processing or event monitoring and opens a window in response to a user-triggered invoke event.
<html>
<head>
<script src="AIRAliases.js"></script>
<script language="javascript">
try {
    air.NativeApplication.nativeApplication.startAtLogin = true;
}
catch ( e ) {
    air.trace( "Cannot set startAtLogin: " + e.message );
}
air.NativeApplication.nativeApplication.addEventListener( air.InvokeEvent.INVOKE, onInvoke );

function onInvoke( event ) {
    if( event.reason == air.InvokeEventReason.LOGIN ) {
        //do background processing...
        air.trace( "Running in background..." );
    } else {
        window.nativeWindow.activate();
    }
}
</script>
</head>
<body>
</body>
</html>

Note: To see the difference in behavior, package and install the application. The startAtLogin property can only be set for installed applications.

Browser invocation

Using the browser invocation feature, a website can launch an installed AIR application to be launched from the browser. Browser invocation is only permitted if the application descriptor file sets allowBrowserInvocation to true:

<allowBrowserInvocation>true</allowBrowserInvocation>

For more information on the application descriptor file, see “Setting AIR application properties” on page 116.

When the application is invoked via the browser, the application’s NativeApplication object dispatches a BrowserInvokeEvent object.
To receive BrowserInvokeEvent events, call the addEventListener() method of the NativeApplication object (NativeApplication.nativeApplication) in the AIR application. When an event listener registers for a BrowserInvokeEvent event, it also receives all BrowserInvokeEvent events that occurred before the registration. These events are dispatched after the call to addEventListener() returns, but not necessarily before other BrowserInvokeEvent events that might be received after registration. This allows you to handle BrowserInvokeEvent events that have occurred before your initialization code executes (such as when the application was initially invoked from the browser). Keep in mind that if you add an event listener later in execution (after application initialization) it still receives all BrowserInvokeEvent events that have occurred since the application started.

The BrowserInvokeEvent object includes the following properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arguments</td>
<td>An array of arguments (strings) to pass to the application.</td>
</tr>
<tr>
<td>isHTTPS</td>
<td>Whether the content in the browser uses the https URL scheme (true) or not (false).</td>
</tr>
<tr>
<td>isUserEvent</td>
<td>Whether the browser invocation resulted in a user event (such as a mouse click). In AIR 1.0, this is always set to true; AIR requires a user event to the browser invocation feature.</td>
</tr>
</tbody>
</table>
| sandboxType  | The sandbox type for the content in the browser. Valid values are defined the same as those that can be used in the Security.sandboxType property, and can be one of the following:  
  * Security.APPLICATION — The content is in the application security sandbox.  
  * Security.LOCAL_TRUSTED — The content is in the local-with-filesystem security sandbox.  
  * Security.LOCAL_WITH_FILE — The content is in the local-with-filesystem security sandbox.  
  * Security.LOCAL_WITH_NETWORK — The content is in the local-with-networking security sandbox.  
  * Security.REMOTE — The content is in a remote (network) domain. |
| securityDomain | The security domain for the content in the browser, such as “www.adobe.com” or “www.example.org”. This property is only set for content in the remote security sandbox (for content from a network domain). It is not set for content in a local or application security sandbox. |

If you use the browser invocation feature, be sure to consider security implications. When a website launches an AIR application, it can send data via the arguments property of the BrowserInvokeEvent object. Be careful using this data in any sensitive operations, such as file or code loading APIs. The level of risk depends on what the application is doing with the data. If you expect only a specific website to invoke the application, the application should check the securityDomain property of the BrowserInvokeEvent object. You can also require the website invoking the application to use HTTPS, which you can verify by checking the isHTTPS property of the BrowserInvokeEvent object.

The application should validate the data passed in. For example, if an application expects to be passed URLs to a specific domain, it should validate that the URLs really do point to that domain. This can prevent an attacker from tricking the application into sending it sensitive data.

No application should use BrowserInvokeEvent arguments that might point to local resources. For example, an application should not create File objects based on a path passed from the browser. If remote paths are expected to be passed from the browser, the application should ensure that the paths do not use the file:/// protocol instead of a remote protocol.

For details on invoking an application from the browser, see “Launching an installed AIR application from the browser” on page 367.
Application termination

The quickest way to terminate an application is to call `NativeApplication.nativeApplication.exit()` and this works fine when your application has no data to save or external resources to clean up. Calling `exit()` closes all windows and then terminates the application. However, to allow windows or other components of your application to interrupt the termination process, perhaps to save vital data, dispatch the proper warning events before calling `exit()`.

Another consideration in gracefully shutting down an application is providing a single execution path, no matter how the shut-down process starts. The user (or operating system) can trigger application termination in the following ways:

- By closing the last application window when `NativeApplication.nativeApplication.autoExit` is true.
- By selecting the application exit command from the operating system; for example, when the user chooses the exit application command from the default menu. This only happens on Mac OS; Windows and Linux do not provide an application exit command through system chrome.
- By shutting down the computer.

When an exit command is mediated through the operating system by one of these routes, the NativeApplication dispatches an `exiting` event. If no listeners cancel the `exiting` event, any open windows are closed. Each window dispatches a `closing` and then a `close` event. If any of the windows cancel the `closing` event, the shut-down process stops.

If the order of window closure is an issue for your application, listen for the `exiting` event from the `NativeApplication` and close the windows in the proper order yourself. This might be the case, for example, if you have a document window with tool palettes. It might be inconvenient, or worse, if the system closed the palettes, but the user decided to cancel the exit command to save some data. On Windows, the only time you will get the `exiting` event is after closing the last window (when the `autoExit` property of the `NativeApplication` object is set to true).

To provide consistent behavior on all platforms, whether the exit sequence is initiated via operating system chrome, menu commands, or application logic, observe the following good practices for exiting the application:

1. Always dispatch an `exiting` event through the `NativeApplication` object before calling `exit()` in application code and check that another component of your application doesn’t cancel the event.

```javascript
function applicationExit(){
    var exitingEvent = new air.Event(air.Event.EXITING, false, true);
    air.NativeApplication.nativeApplication.dispatchEvent(exitingEvent);
    if (!exitingEvent.isDefaultPrevented()) {
        air.NativeApplication.nativeApplication.exit();
    }
}
```

2. Listen for the application `exiting` event from the `NativeApplication` object and, in the handler, close any windows (dispatching a `closing` event first). Perform any needed clean-up tasks, such as saving application data or deleting temporary files, after all windows have been closed. Only use synchronous methods during cleanup to ensure that they finish before the application quits.

If the order in which your windows are closed doesn’t matter, then you can loop through the `NativeApplication.openedWindows` array and close each window in turn. If order does matter, provide a means of closing the windows in the correct sequence.
function onExiting(exitingEvent) {
    var winClosingEvent;
    for (var i = 0; i < air.NativeApplication.nativeApplication.openedWindows.length; i++) {
        var win = air.NativeApplication.nativeApplication.openedWindows[i];
        winClosingEvent = new air.Event(air.Event.CLOSING,false,true);
        win.dispatchEvent(winClosingEvent);
        if (!winClosingEvent.isDefaultPrevented()) {
            win.close();
        } else {
            exitingEvent.preventDefault();
        }
    }

    if (!exitingEvent.isDefaultPrevented()) {
        //perform cleanup
    }
}

3 Windows should always handle their own clean up by listening for their own closing events.

4 Only use one exiting listener in your application since handlers called earlier cannot know whether subsequent handlers will cancel the exiting event (and it would be unwise to rely on the order of execution).

**More Help topics**

“Setting AIR application properties” on page 116

“Presenting a custom application update user interface” on page 378
Chapter 31: Reading application settings

At runtime, you can get properties of the application descriptor file as well as the publisher ID for an application. These are set in the \texttt{applicationDescriptor} and \texttt{publisherID} properties of the \texttt{NativeApplication} object.

\textbf{Note:} As of AIR 1.5.3, the publisher ID is deprecated. Application updates can specify a publisher ID for backwards compatibility with the earlier application version. New applications should not specify a publisher ID. In applications published prior to AIR 1.5.3, the publisher ID is computed based on the signing certificate and is not specified in the application descriptor.

Reading the application descriptor file

You can read the application descriptor file of the currently running application, by getting the \texttt{applicationDescriptor} property of the \texttt{NativeApplication} object, as in the following:

\begin{verbatim}
var appXml = air.nativeApplication.nativeApplication.applicationDescriptor;
\end{verbatim}

You can use a DOMParser object to parse the data, as in the following:

\begin{verbatim}
var xmlString = air.NativeApplication.nativeApplication.applicationDescriptor;
var appXml = new DOMParser();
var xmlobject = appXml.parseFromString(xmlString, "text/xml");
var root = xmlobject.getElementsByTagName('application')[0];
var appId = root.getElementsByTagName("id")[0].firstChild.data;
var appVersion = root.getElementsByTagName("version")[0].firstChild.data;
var appName = root.getElementsByTagName("filename")[0].firstChild.data;
air.trace("appId: " + appId);
air.trace("version: " + appVersion);
air.trace("filename: " + appName);
\end{verbatim}

For more information, see “The application descriptor file structure” on page 116.

Getting the application and publisher identifiers

The application ID and optional publisher id together uniquely identify an AIR application. You specify the application ID in the \texttt{id} element of the application descriptor and the publisher ID in the \texttt{publisherID} element. (However, prior to AIR 1.5.3, the publisher ID was derived from the certificate used to sign the AIR installation package and is not found in the application descriptor.)

The application ID can be read from the \texttt{NativeApplication} object’s \texttt{id} property, as illustrated in the following code:

\begin{verbatim}
air.trace(air.NativeApplication.nativeApplication.applicationID);
\end{verbatim}

The publisher ID can be read from the \texttt{NativeApplication} \texttt{publisherID} property:

\begin{verbatim}
air.trace(air.NativeApplication.nativeApplication.publisherID);
\end{verbatim}

In applications published with AIR 1.5.2 or earlier, the publisher ID for an installed application can also be found in the \texttt{META-INF/AIR/publisherid} file within the application install directory.

For more information, see “About AIR publisher identifiers” on page 370.
Chapter 32: Working with runtime and operating system information

This section discusses ways that an AIR application can manage operating system file associations, detect user activity, and get information about the Adobe® AIR® runtime.

Managing file associations

Associations between your application and a file type must be declared in the application descriptor. During the installation process, the AIR application installer associates the AIR application as the default opening application for each of the declared file types, unless another application is already the default. The AIR application install process does not override an existing file type association. To take over the association from another application, call the NativeApplication.setAsDefaultApplication() method at run time.

It is a good practice to verify that the expected file associations are in place when your application starts up. This is because the AIR application installer does not override existing file associations, and because file associations on a user’s system can change at any time. When another application has the current file association, it is also a polite practice to ask the user before taking over an existing association.

The following methods of the NativeApplication class let an application manage file associations. Each of the methods takes the file type extension as a parameter:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isSetAsDefaultApplication()</td>
<td>Returns true if the AIR application is currently associated with the specified file type.</td>
</tr>
<tr>
<td>setAsDefaultApplication()</td>
<td>Creates the association between the AIR application and the open action of the file type.</td>
</tr>
<tr>
<td>removeAsDefaultApplication()</td>
<td>Removes the association between the AIR application and the file type.</td>
</tr>
<tr>
<td>getDefaultApplication()</td>
<td>Reports the path of the application that is currently associated with the file type.</td>
</tr>
</tbody>
</table>

AIR can only manage associations for the file types originally declared in the application descriptor. You cannot get information about the associations of a non-declared file type, even if a user has manually created the association between that file type and your application. Calling any of the file association management methods with the extension for a file type not declared in the application descriptor causes the application to throw a runtime exception.

For information about declaring file types in the application descriptor, see “Declaring file type associations” on page 124.

Getting the runtime version and patch level

The NativeApplication object has a runtimeVersion property, which is the version of the runtime in which the application is running (a string, such as "1.0.5"). The NativeApplication object also has a runtimePatchLevel property, which is the patch level of the runtime (a number, such as 2960). The following code uses these properties:

```javascript
air.trace(air.NativeApplication.nativeApplication.runtimeVersion);
air.trace(air.NativeApplication.nativeApplication.runtimePatchLevel);
```
Detecting AIR capabilities

For a file that is bundled with the Adobe AIR application, the Security.sandboxType property is set to the value defined by the Security.APPLICATION constant. You can load content (which may or may not contain APIs specific to AIR) based on whether a file is in the Adobe AIR security sandbox, as illustrated in the following code:

```javascript
if (window.runtime)
{
    if (air.Security.sandboxType == air.Security.APPLICATION)
    {
        alert("In AIR application security sandbox.");
    }
    else
    {
        alert("Not in AIR application security sandbox.");
    }
}
else
{
    alert("Not in the Adobe AIR runtime.")
}
```

All resources that are not installed with the AIR application are put in security sandboxes based on their domains of origin. For example, content served from www.example.com is put in a security sandbox for that domain.

You can check if the window.runtime property is set to see if content is executing in the runtime.

For more information, see “AIR security” on page 100.

Tracking user presence

The NativeApplication object dispatches two events that help you detect when a user is actively using a computer. If no mouse or keyboard activity is detected in the interval determined by the NativeApplication.idleThreshold property, the NativeApplication dispatches a userIdle event. When the next keyboard or mouse input occurs, the NativeApplication object dispatches a userPresent event. The idleThreshold interval is measured in seconds and has a default value of 300 (5 minutes). You can also get the number of seconds since the last user input from the NativeApplication.nativeApplication.lastUserInput property.

The following lines of code set the idle threshold to 2 minutes and listen for both the userIdle and userPresent events:

```javascript
air.NativeApplication.nativeApplication.idleThreshold = 120;
air.NativeApplication.nativeApplication.addEventListener(air.Event.USER_IDLE, function(event) {
    air.trace("Idle");
});
air.NativeApplication.nativeApplication.addEventListener(air.Event.USER_PRESENT, function(event) {
    air.trace("Present");
});
```

**Note:** Only a single userIdle event is dispatched between any two userPresent events.
Chapter 33: Monitoring network connectivity

Adobe® AIR® provides the means to check for changes to the network connectivity of the computer on which an AIR application is installed. This information is useful if an application uses data obtained from the network. Also, an application can check the availability of a network service.

Detecting network connectivity changes

Your 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 application’s NativeApplication object dispatches the network change event. To react to this event, add a listener:

```javascript
air.NativeApplication.nativeApplication.addEventListener(air.Event.NETWORK_CHANGE, onNetworkChange);
```

And define an event handler function:

```javascript
function onNetworkChange(event)
{
  //Check resource availability
}
```

The `Event.NETWORK_CHANGE` event does not indicate a change in all network activity, but 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 functions separately from the AIR framework. For HTML-based AIR applications, the servicemonitor.swf must be included in your AIR application package. The servicemonitor.swf must also be included in your AIR application code, as follows:

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

The servicemonitor.swf file is included in the frameworks directory of the AIR SDK.
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 dispatches events regarding network connectivity. The ServiceMonitor object dispatches these events when the instance is created and whenever a network change is detected by Adobe 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. A ServiceMonitor object does not check network connectivity until the start() method is called.

The URLMonitor class, a subclass of the ServiceMonitor class, detects changes in HTTP connectivity for a specified URLRequest.

The SocketMonitor class, also a subclass of 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:

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

<script>
    var monitor;
    function test() {
        monitor = new air.URLMonitor(new air.URLRequest('http://www.adobe.com'));
        monitor.addEventListener(air.StatusEvent.STATUS, announceStatus);
        monitor.start();
    }

    function announceStatus(e) {
        air.trace("Status change. Current status: " + monitor.available);
    }
</script>
```

### 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:

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

<script>
    var monitor;
    function test() {
        monitor = new air.SocketMonitor('6667');
        monitor.addEventListener(air.StatusEvent.STATUS, announceStatus);
        monitor.start();
    }

    function announceStatus(e) {
        air.trace("Status change. Current status: " + monitor.available);
    }
</script>
```
Monitoring network connectivity

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

<script>
    function test()
    {
        socketMonitor = new air.SocketMonitor('www.adobe.com', 6667);
        socketMonitor.addEventListener(air.StatusEvent.STATUS, socketStatusChange);
        socketMonitor.start();
    }
    function announceStatus(e) {
        air.trace("Status change. Current status: " + socketMonitor.available);
    }
</script>
```
Chapter 34: URL requests and networking

This section explains Adobe® AIR® features that let applications communicate with the network. It also explains how to load data from external sources, send messages between a server and an AIR application, and perform file uploads and downloads using the FileReference and FileReferenceList classes.

This section describes the AIR networking and communication API—functionality uniquely provided to applications running in the 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 XMLHttpRequest class).

Basics of networking and communication

When you build more complex AIR applications, you may need to communicate with server-side scripts, or load data from XML or text files on the internet. The runtime contains classes to send and receive data across the Internet—for example, to load content from remote URLs, to communicate with other AIR applications, and to connect to remote websites.

In AIR, you can load external files with the URLLoader and URLStream classes. You then use a specific class to access the data, depending on the type of data that was loaded. For instance, if the remote content is formatted as name-value pairs, you use the URLVariables class to parse the server results. Alternatively, if the file loaded using the URLLoader and URLStream classes is a remote XML document, you can parse the XML document using the DOMParser class. This allows you to simplify your code because the code for loading external files is the same whether you use the URLVariables, DOMParser, or some other class to parse and work with the remote data.

The runtime also contains classes for other types of remote communication. These include the FileReference class for uploading and downloading files from a server, the Socket and XMLSocket classes that allow you to communicate directly with remote computers over socket connections, and the NetConnection class, which is used for communicating with a remote application server, such as Flash Media Server 2.

Finally, the runtime includes a LocalConnection class, which allows you to communicate among AIR applications (and SWF files in a browser) running on a single computer.

Networking and communication terms

The following list contains important AIR networking and communication terms:

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>External data</td>
<td>Data that is stored in some form outside of the AIR application, and loaded into the application when needed. This data could be stored in a file that’s loaded directly from a server, or stored in a database or other form that is retrieved by calling scripts or programs running on a server.</td>
</tr>
<tr>
<td>URL-encoded variables</td>
<td>The URL-encoded format provides a way to represent several variables (pairs of variable names and values) in a single string of text. Individual variables are written in the format name=value. Each variable (that is, each name-value pair) is separated by ampersand characters, like this: variable1=value1&amp;variable2=value2. In this way, an indefinite number of variables can be sent as a single message.</td>
</tr>
<tr>
<td>MIME type</td>
<td>A standard code used to identify the type of a given file in Internet communication. Any given file type has a specific code that is used to identify it. When sending a file or message, a computer (such as a web server or an AIR application) specifies the type of file being sent.</td>
</tr>
</tbody>
</table>
Using the URLRequest class

Many networking-related runtime APIs use the URLRequest class to define how a network request is made. The URLRequest class lets you define more than simply the URL string. Content in the runtime can define URLs using new URL schemes (in addition to standard schemes like file and http).

URLRequest properties

The URLRequest class includes the following properties which are available to content only in the AIR application security sandbox:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>followRedirects</td>
<td>Specifies whether redirects are to be followed (true, the default value) or not (false). This is only supported in the runtime.</td>
</tr>
<tr>
<td>manageCookies</td>
<td>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.</td>
</tr>
<tr>
<td>authenticate</td>
<td>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 &quot;Setting URLRequest defaults&quot; on page 344.</td>
</tr>
<tr>
<td>cacheResponse</td>
<td>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).</td>
</tr>
<tr>
<td>useCache</td>
<td>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.</td>
</tr>
<tr>
<td>userAgent</td>
<td>Specifies the user-agent string to be used in the HTTP request.</td>
</tr>
</tbody>
</table>

The following properties of a URLRequest object can be set by content in any sandbox (not just the AIR application security sandbox):

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contentType</td>
<td>The MIME content type of any data sent with the URL request.</td>
</tr>
<tr>
<td>data</td>
<td>An object containing data to be transmitted with the URL request.</td>
</tr>
<tr>
<td>digest</td>
<td>A secure &quot;digest&quot; to a cached file to track Adobe® Flash® Player cache.</td>
</tr>
</tbody>
</table>
344 DEVELOPING ADOBE AIR 1.5 APPLICATIONS WITH HTML AND AJAX

URL requests and networking

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>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 “AIR security” on page 100.)</td>
</tr>
<tr>
<td>requestHeaders</td>
<td>The array of HTTP request headers to be appended to the HTTP request.</td>
</tr>
<tr>
<td>url</td>
<td>Specifies the URL to be requested.</td>
</tr>
</tbody>
</table>

Note: The HTMLLoader class has related properties for settings pertaining to content loaded by an HTMLLoader object. For details, see About the HTMLLoader class.

Setting URLRequest defaults

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:

```javascript
air.URLRequestDefaults.manageCookies = false;
air.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:

```javascript
air.URLRequestDefaults.setLoginCredentialsForHost("www.example.com", "Ada", "love1816$X");
```

Each property of URLRequestDefaults settings applies 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 in to 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 Adobe AIR Language Reference for HTML Developers (http://www.adobe.com/go/learn_air_html_jslr).

Using AIR URL schemes in URLs

The standard URL schemes, such as the following, are available when defining URLs in any security sandbox in AIR:

- **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:

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

  You can also use the following schemes when defining a URL for content running in the application security sandbox:
app:
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:/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 for each user, which is a useful place to store data specific to that application. For example, the following path points to a prefs.xml file in a settings subdirectory of the application store directory:

app-storage:/settings/prefs.xml

The application storage directory location is based on the user name, the application ID, and the publisher ID:

- On Mac OS—in:
  
  /Users/user name/Library/Preferences/applicationID.publisherID/Local Store/

  For example:

  /Users/babbage/Library/Preferences/com.example.TestApp.02D88EEED35F84C264A183921344EEA353A629FD.1/Local Store

- On Windows—in the documents and Settings directory, in:
  
  user name/Application Data/applicationID.publisherID/Local Store/

  For example:

  C:\Documents and Settings\babbage\Application Data\com.example.TestApp.02D88EEED35F84C264A183921344EEA353A629FD.1\Local Store

- On Linux—in:

  /home/user name/.appdata/applicationID.publisherID/Local Store/

  For example:

  /home/babbage/.appdata/com.example.TestApp.02D88EEED35F84C264A183921344EEA353A629FD.1\Local Store

The URL (and url property) for a File object created with File.applicationStorageDirectory uses the app-storage URL scheme, as in the following:

```javascript
var dir = air.File.applicationStorageDirectory;
dir = dir.resolvePath("prefs.xml");
air.trace(dir.url); // app-storage:/preferences
```

mailto:
You can use the mailto scheme in URLRequest objects passed to the navigateToURL() function. See “Opening a URL in the default system web browser” on page 354.
Using URL schemes in AIR
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-specific URL schemes (app: and app-storage:) in content in the application security sandbox. For more information, see “AIR security” on page 100.

Prohibited URL schemes
Some APIs allow you to launch content in a web browser. For security reasons, some URL schemes are prohibited when using these APIs in AIR. The list of prohibited schemes depends on the security sandbox of the code using the API. For details, see “Opening a URL in the default system web browser” on page 354.

Working with external data
The runtime includes mechanisms for loading data from external sources. Those sources can be static content such as text files, or dynamic content, such as content generated by a web script that retrieves data from a database. The data can be formatted in a variety of ways, and the runtime provides functionality for decoding and accessing the data. You can also send data to the external server as part of the process of retrieving data.

Using the URLLoader and URLVariables classes
The runtime includes the URLLoader and URLVariables classes for loading external data. The URLLoader class downloads data from a URL as text, binary data, or URL-encoded variables. The URLLoader class is useful for downloading text files, XML, or other information to use in AIR applications. The URLLoader class takes advantage of the runtime event-handling model, which allows you to listen for such events as complete, httpStatus, ioError, open, progress, and securityError.

The URLLoader data is not available until the download has completed. You can monitor the progress of the download (bytes loaded and bytes total) by listening for the progress event to be dispatched, although if a file loads too quickly a progress event may not be dispatched. When a file has successfully downloaded, the complete event is dispatched. The loaded data is decoded from UTF-8 or UTF-16 encoding into a string.

Note: If no value is set for URLRequest.contentType, values are sent as application/x-www-form-urlencoded.

The URLLoader.load() method (and optionally the URLLoader class’s constructor) takes a single parameter, request, which is a URLRequest instance. A URLRequest instance contains all of the information for a single HTTP request, such as the target URL, request method (such as GET or POST), additional header information, and the MIME type (for example, when you upload XML content).

For example, to upload an XML packet to a server-side script, you could use the following code:
DEVELOPING ADOBE AIR 1.5 APPLICATIONS WITH HTML AND AJAX

URL requests and networking

```javascript
var secondsUTC = new Date().getTime();
var dataXML = "<login>
  + "<time>" + secondsUTC + "</time>
  + "<username>Ernie</username>"
  + "<password>guru</password>"
  + "</login>";
var request = new air.URLRequest("http://www.example.com/login.cfm");
request.contentType = "text/xml";
request.data = dataXML;
request.method = air.URLRequestMethod.POST;
var loader = new air.URLLoader();
loader.load(request);
```

The previous code creates an XML instance named `dataXML` that contains an XML packet to be sent to the server. Next, you set the `URLRequest` `contentType` property to "text/xml" and set the `URLRequest` `data` property to the contents of the XML packet. Finally, you create a `URLLoader` instance and send the request to the remote script by using the `URLLoader.load()` method.

There are three ways in which you can specify parameters to pass in a URL request:

- Within the `URLVariables` constructor
- Within the `URLVariables.decode()` method
- As specific properties within the `URLVariables` object itself

When you define variables within the `URLVariables` constructor or within the `URLVariables.decode()` method, you need to make sure that you URL-encode the ampersand character because it has a special meaning and acts as a delimiter. For example, when you pass an ampersand, you need to URL-encode the ampersand by changing it from `&` to `%26` because the ampersand acts as a delimiter for parameters.

### Loading data from external network documents

The following snippet creates a `URLRequest` and `URLLoader` object, which loads the contents of a text file on the network:

```javascript
var request = new air.URLRequest("http://www.example.com/data/params.txt");
var loader = new air.URLLoader();
loader.load(request);
```

By default, if you do not define a request method, the runtime loads the content using the HTTP GET method. If you want to send the data using the POST method, you need to set the `request.method` property to `POST` using the static constant `URLRequestMethod.POST`, as the following code shows:

```javascript
var request = new air.URLRequest("http://www.example.com/sendfeedback.cfm");
request.method = air.URLRequestMethod.POST;
```

The external document, `params.txt`, that is loaded at run time contains the following data:

```plaintext
monthNames=January,February,March,April,May,June,July,August,September,October,November,December
dayNames=Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday
```

The file contains two parameters, `monthNames` and `dayNames`. Each parameter contains a comma-separated list that is parsed as strings. You can split this list into an array using the `String.split()` method.

**Tip:** Avoid using reserved words or language constructs as variable names in external data files, because doing so makes reading and debugging your code more difficult.

Once the data has loaded, the `Event.COMPLETE` event is dispatched, and the contents of the external document are available to use in the `URLLoader`'s `data` property, as the following code shows:
function completeHandler(event)
{
    var loader2 = event.target;
    air.trace(loader2.data);
}

If the remote document contains name-value pairs, you can parse the data using the URLVariables class by passing in the contents of the loaded file, as follows:

function completeHandler(event)
{
    var loader2 = event.target;
    var variables = new air.URLVariables(loader2.data);
    air.trace(variables.dayNames);
}

Each name-value pair from the external file is created as a property in the URLVariables object. Each property within the variables object in the previous code sample is treated as a string. If the value of the name-value pair is a list of items, you can convert the string into an array by calling the String.split() method, as follows:

var dayNameArray = variables.dayNames.split(",");

If you are loading numeric data from external text files, you need to convert the values into numeric values by using a top-level function, such as parseInt(), parseFloat(), and Number().

Instead of loading the contents of the remote file as a string and creating a URLVariables object, you could instead set the URLLoader.dataFormat property to one of the static properties found in the URLLoaderDataFormat class. The three possible values for the URLLoader.dataFormat property are as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>air.URLLoaderDataFormat.BINARY</td>
<td>The URLLoader.data property contains binary data stored in a ByteArray object.</td>
</tr>
<tr>
<td>air.URLLoaderDataFormat.TEXT</td>
<td>The URLLoader.data property contains text in a String object.</td>
</tr>
<tr>
<td>air.URLLoaderDataFormat.VARIABLES</td>
<td>The URLLoader.data property contains URL-encoded variables stored in a URLVariables object.</td>
</tr>
</tbody>
</table>

The following code demonstrates how setting the URLLoader.dataFormat property to air.URLLoaderDataFormat.VARIABLES allows you to automatically parse loaded data into a URLVariables object:

var request = new air.URLRequest("http://www.example.com/params.txt");
var variables = new air.URLLoader();
variables.dataFormat = air.URLLoaderDataFormat.VARIABLES;
variables.addEventListener(air.Event.COMPLETE, completeHandler);
try
{
    variables.load(request);
}
catch (error)
{
    air.trace("Unable to load URL: " + error);
}

function completeHandler(event)
{
    var loader = event.target;
    air.trace(loader.data.dayNames);
}
Note: The default value for URLLoader.dataFormat is air.URLLoaderDataFormat.TEXT.

As the following example shows, loading XML from an external file is the same as loading URLVariables. You can create a URLRequest instance and a URLLoader instance and use them to download a remote XML document. When the file has completely downloaded, the complete event is dispatched and the trace() function outputs the contents of the file to the command line.

```javascript
var request = new air.URLRequest("http://www.example.com/data.xml");
var loader = new air.URLLoader();
loader.addEventListener(air.Event.COMPLETE, completeHandler);
loader.load(request);

function completeHandler(event)
{
    var dataXML = event.target.data;
    air.trace(dataXML);
}
```

**Communicating with external scripts**

In addition to loading external data files, you can also use the URLVariables class to send variables to a server and process the server’s response. This is useful, for example, if you are programming a game and want to send the user’s score to a server to calculate whether it should be added to the high scores list, or even send a user’s login information to a server for validation. A server-side script can process the user name and password, validate it against a database, and return confirmation of whether the user-supplied credentials are valid.

The following snippet creates a URLVariables object named `variables`, which creates a variable called `name`. Next, a URLRequest object is created that specifies the URL of the server-side script to send the variables to. Then you set the method property of the URLRequest object to send the variables as an HTTP POST request. To add the URLVariables object to the URL request, you set the data property of the URLRequest object to the URLVariables object created earlier. Finally, the URLLoader instance is created and the URLLoader.load() method is invoked, which initiates the request.

```javascript
var variables = new air.URLVariables("name=Franklin");
var request = new air.URLRequest();
request.url = "http://www.example.com/greeting.cfm";
request.method = air.URLRequestMethod.POST;
request.data = variables;
var loader = new air.URLLoader();
loader.dataFormat = air.URLLoaderDataFormat.VARIABLES;
loader.addEventListener(air.Event.COMPLETE, completeHandler);
try
{
    loader.load(request);
}
catch (error)
{
    air.trace("Unable to load URL");
}

function completeHandler(event)
{
    air.trace(event.target.data.welcomeMessage);
}
```
The following code contains the contents of the Adobe® ColdFusion® greeting.cfm document used in the previous example:

```cfml
<cfif NOT IsDefined("Form.name") OR Len(Trim(Form.Name)) EQ 0>
   <cfset Form.Name = "Stranger" />
</cfif>
<cfoutput>
   welcomeMessage=#UrlEncodedFormat("Welcome, " & Form.name)#
</cfoutput>
```

### Using the URLStream class

The URLStream class provides low-level access to downloading URLs. Data is made available immediately as it is downloaded, instead of waiting until the entire data is sent as with URLLoader. The URLStream class also lets you close a stream before it finishes downloading. The contents of the downloaded file are made available as raw binary data.

The read operations in URLStream are non-blocking. This means that you must use the `bytesAvailable` property to determine whether sufficient data is available before reading it. An EOFError exception is thrown if insufficient data is available.

The URLStream class dispatches an `httpResponseStatus` event before any response data is delivered. 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.

### Socket connections

There are two different types of socket connections possible in the runtime: XML socket connections and binary socket connections.

An XML socket lets you connect to a remote server and create a server connection that remains open until explicitly closed. This lets you exchange XML data between a server and client without having to continually open new server connections. Another benefit of using an XML socket server is that the user doesn’t need to explicitly request data. You can send data from the server without requests, and you can send data to every client connected to the XML socket server.

A binary socket connection is similar to an XML socket except that the client and server don’t need to exchange XML packets specifically. Instead, the connection can transfer data as binary information. This allows you to connect to a wide range of services, including mail servers (POP3, SMTP, and IMAP), and news servers (NNTP).

### Socket class

The Socket class enables AIR applications to make socket connections and to read and write raw binary data. It is similar to the XMLSocket class, but does not dictate the format of the received and transmitted data. The Socket class is useful for interoperating with servers that use binary protocols. By using binary socket connections, you can write code that allows interaction with several different Internet protocols, such as POP3, SMTP, IMAP, and NNTP. This in turn enables AIR applications to connect to mail and news servers.
AIR applications can interface with a server by using the binary protocol of that server directly. Some servers use the big-endian byte order, and some use the little-endian byte order. Most servers on the Internet use the big-endian byte order because "network byte order" is big-endian. You should use the endian byte order that matches the byte order of the server that is sending or receiving data. All operations are encoded by default in big-endian format; that is, with the most significant byte first. This is done to match Java and official network byte order. To change whether big-endian or little-endian byte order is used, you can set the \texttt{endian} property to \texttt{air.Endian.BIG_ENDIAN} or \texttt{air.Endian.LITTLE_ENDIAN}.

\textbf{Tip:} The \texttt{Socket} class inherits all the methods implemented by the \texttt{IDataInput} and \texttt{IDataOutput} interfaces (located in the \texttt{flash.utils} package), and those methods should be used to write to and read from the \texttt{Socket}.

\section*{XMLSocket class}

The runtime provides a built-in XMLSocket class, which lets you open a continuous connection with a server. This open connection removes latency issues and is commonly used for real-time applications such as chat applications or multiplayer games. A traditional HTTP-based chat solution frequently polls the server and downloads new messages using an HTTP request. In contrast, an XMLSocket chat solution maintains an open connection to the server, which lets the server immediately send incoming messages without a request from the client.

To create a socket connection, you must create a server-side application to wait for the socket connection request and send a response to the AIR application. This type of server-side application can be written in a programming language such as Java, Python, or Perl. To use the XMLSocket class, the server computer must run a daemon that understands the protocol used by the XMLSocket class. The protocol is described in the following list:

\begin{itemize}
  \item XML messages are sent over a full-duplex TCP/IP stream socket connection.
  \item Each XML message is a complete XML document, terminated by a zero (0) byte.
  \item An unlimited number of XML messages can be sent and received over a single XMLSocket connection.
\end{itemize}

\textbf{Note:} The XMLSocket class cannot tunnel through firewalls automatically because, unlike the Real-Time Messaging Protocol (RTMP), XMLSocket has no HTTP tunneling capability. If you need to use HTTP tunneling, consider using Flash Remoting or Adobe Flash Media Server (which supports RTMP) instead.

The following restrictions apply to how and where content outside of the application security sandbox can use an XMLSocket object to connect to the server:

\begin{itemize}
  \item For content outside of the application security sandbox, the \texttt{XMLSocket.connect()} method can connect only to TCP port numbers greater than or equal to 1024. One consequence of this restriction is that the server daemons that communicate with the XMLSocket object must also be assigned to port numbers greater than or equal to 1024. Port numbers below 1024 are often used by system services such as FTP (21), Telnet (23), SMTP (25), HTTP (80), and POP3 (110), so XMLSocket objects are barred from these ports for security reasons. The port number restriction limits the possibility that these resources will be inappropriately accessed and abused.
  \item For content outside of the application security sandbox, the \texttt{XMLSocket.connect()} method can connect only to computers in the same domain where the content resides. (This restriction is identical to the security rules for \texttt{URLLoader.load()}. To connect to a server daemon running in a domain other than the one where the content resides, you can create a cross-domain policy file on the server that allows access from specific domains. For details on cross-domain policy files, see “AIR security” on page 100.
\end{itemize}

\textbf{Note:} Setting up a server to communicate with the XMLSocket object can be challenging. If your application does not require real-time interactivity, use the \texttt{URLLoader} class instead of the XMLSocket class.
You can use the `XMLSocket.connect()` and `XMLSocket.send()` methods of the XMLSocket class to transfer XML to and from a server over a socket connection. The `XMLSocket.connect()` method establishes a socket connection with a web server port. The `XMLSocket.send()` method passes an XML object to the server specified in the socket connection.

When you invoke the `XMLSocket.connect()` method, the runtime opens a TCP/IP connection to the server and keeps that connection open until one of the following occurs:

- The `XMLSocket.close()` method of the XMLSocket class is called.
- No more references to the XMLSocket object exist.
- The connection is broken (for example, the modem disconnects).

### Creating and connecting to a Java XML socket server

The following code demonstrates a simple XMLSocket server written in Java that accepts incoming connections and displays the received messages in the command prompt window. By default, a new server is created on port 8080 of your local machine, although you can specify a different port number when starting your server from the command line.

Create a text document and add the following code:

```java
import java.io.*;
import java.net.*;

class SimpleServer
{
    private static SimpleServer server;
    ServerSocket socket;
    Socket incoming;
    BufferedReader readerIn;
    PrintStream printOut;

    public static void main(String[] args)
    {
        int port = 8080;
        try
        {
            port = Integer.parseInt(args[0]);
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
            // Catch exception and keep going.
        }

        server = new SimpleServer(port);
    }

    private SimpleServer(int port)
    {
        System.out.println(">> Starting SimpleServer");
        try
        {
            socket = new ServerSocket(port);
            incoming = socket.accept();
        }
    }
}
```
readerIn = new BufferedReader(new InputStreamReader(incoming.getInputStream()));
printOut = new PrintStream(incoming.getOutputStream());
printOut.println("Enter EXIT to exit.\r");
out("Enter EXIT to exit.\r");
boolean done = false;
while (!done) {
    String str = readerIn.readLine();
    if (str == null) {
        done = true;
    } else {
        out("Echo: " + str + "\r");
        if (str.trim().equals("EXIT")) {
            done = true;
        }
    }
    incoming.close();
}
catch (Exception e) {
    System.out.println(e);
}

private void out(String str) {
    printOut.println(str);
    System.out.println(str);
}

Save the document to your hard disk as SimpleServer.java and compile it using a Java compiler, which creates a Java class file named SimpleServer.class.

You can start the XMLSocket server by opening a command prompt and typing java SimpleServer. The SimpleServer.class file can be located anywhere on your local computer or network; it doesn’t need to be placed in the root directory of your web server.

Tip: If you’re unable to start the server because the files are not located within the Java classpath, try starting the server with java -classpath . SimpleServer.

To connect to the XMLSocket from your AIR application, you need to create an instance of the XMLSocket class, and call the XMLSocket.connect() method while passing a host name and port number, as follows:

var xmlsock = new air.XMLSocket();
xmlsock.connect("127.0.0.1", 8080);

Whenever you receive data from the server, the data event is dispatched:
xmlsock.addEventListener(window.runtime.flash.events.DataEvent.DATA, onData);
function onData(event) {
    air.trace("[" + event.type + "] " + event.data);
}

To send data to the XMLSocket server, you use the XMLSocket.send() method and pass a string. The runtime sends the content to the XMLSocket server followed by a zero (0) byte:
xmlsock.send(xmlFormattedData);

The XMLSocket.send() method does not return a value that indicates whether the data was successfully transmitted. If an error occurred while trying to send data, an IOError error is thrown.

Tip: Each message you send to the XML socket server must be terminated by a new line (\n) character.

Opening a URL in the default system web browser

You can use the navigateToURL() function to open a URL in the default system web browser. For the URLRequest object you pass as the request parameter of this function, only the url property is used.

The first parameter of the navigateToURL() function, the navigate parameter, is a URLRequest object (see “Using the URLRequest class” on page 343). The second is an optional window parameter, in which you can specify the window name. For example, the following code opens the www.adobe.com web site in the default system browser:

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

Note: When using the navigateToURL() function, the runtime treats a URLRequest object that uses the POST method (one that has its method property set to URLRequestMethod.POST) as using the GET method.

When using the navigateToURL() function, URL schemes are permitted based on the security sandbox of the code calling the navigateToURL() function.

Some APIs allow you to launch content in a web browser. For security reasons, some URL schemes are prohibited when using these APIs in AIR. The list of prohibited schemes depends on the security sandbox of the code using the API. (For details on security sandboxes, see “AIR security” on page 100.)

Application sandbox
The following schemes are allowed. Use these as you would use them in a web browser.

- http:
- https:
- file:
- mailto: — AIR directs these requests to the registered system mail application
- app:
- app-storage:

All other URL schemes are prohibited.
Remote sandbox
The following schemes are allowed. Use these as you would use them in a web browser.

- http:
- https:
- mailto: — AIR directs these requests to the registered system mail application

All other URL schemes are prohibited.

Local-with-file sandbox
The following schemes are allowed. Use these as you would use them in a web browser.

- file:
- mailto: — AIR directs these requests to the registered system mail application

All other URL schemes are prohibited.

Local-with-networking sandbox
The following schemes are allowed. Use these as you would use them in a web browser.

- http:
- https:
- mailto: — AIR directs these requests to the registered system mail application

All other URL schemes are prohibited.

Local-trusted sandbox
The following schemes are allowed. Use these as you would use them in a web browser.

- file:
- http:
- https:
- mailto: — AIR directs these requests to the registered system mail application

All other URL schemes are prohibited.

Sending a URL to a server

You can use the sendToURL() function to send a URL request to a server. This function ignores any server response. The sendToURL() function takes one argument, request, which is a URLRequest object (see "Using the URLRequest class" on page 343). Here is an example:
var url = "http://www.example.com/application.jsp";
var variables = new air.URLVariables();
variables.sessionId = new Date().getTime();
variables.userLabel = "Your Name";
var request = new air.URLRequest(url);
request.data = variables;
air.sendToURL(request);

This example uses the URLVariables class to include variable data in the URLRequest object. For more information, see “Using the URLLoader and URLVariables classes” on page 346.
Chapter 35: Inter-application communication

The LocalConnection class enables communications between Adobe® AIR® applications, as well as among AIR applications and SWF content running in the browser.

About the LocalConnection class

LocalConnection objects can communicate only among AIR applications and SWF files that are running on the same client computer, but they can run in different applications. For example, two AIR applications can communicate using the LocalConnection class, as can an AIR application and a SWF file running in a browser.

The simplest way to use a LocalConnection object is to allow communication only between LocalConnection objects located in the same domain or the same AIR application. That way, you won’t have to worry about security issues. However, if you need to allow communication between domains, you have several ways to implement security measures. For more information, see the discussion of the connectionName parameter of the send() method and the allowDomain() and domain entries in the LocalConnection class listing in the Adobe AIR Language Reference.

To add callback methods to your LocalConnection objects, set the LocalConnection.client property to an object that has member methods, as the following code shows:

```javascript
var lc = new air.LocalConnection();
var clientObject = new Object();
clientObject.doMethod1 = function() {
  air.trace("doMethod1 called.");
}
clientObject.doMethod2 = function(param1) {
  air.trace("doMethod2 called with one parameter: " + param1);
  air.trace("The square of the parameter is: " + param1 * param1);
}
lc.client = clientObject;
```

The LocalConnection.client property includes all callback methods that can be invoked.

Sending messages between two applications

You use the LocalConnection class to communicate between different AIR applications and between AIR applications and Adobe® Flash® Player (SWF) applications running in a browser.

The following code defines a LocalConnection object that acts as a server and accepts incoming LocalConnection calls from other applications:
var lc = new air.LocalConnection();
lc.connect("connectionName");
var clientObject = new Object();
clientObject.echoMsg = function(msg) {
    air.trace("This message was received: " + msg);
}
lc.client = clientObject;

This code first creates a LocalConnection object named lc and sets the client property to an object, clientObject. When another application calls a method in this LocalConnection instance, AIR looks for that method in the clientObject object.

If you already have a connection with the specified name, an ArgumentError exception is thrown, indicating that the connection attempt failed because the object is already connected.

The following snippet demonstrates how to create a LocalConnection with the name conn1:

    connection.connect("conn1");

Connecting to the primary application from a secondary application requires that you first create a LocalConnection object in the sending LocalConnection object, and then call the LocalConnection.send() method with the name of the connection and the name of the method to execute. For example, to connect to the LocalConnection object that you created earlier, use the following code:

    sendingConnection.send("conn1", "echoMsg", "Bonjour.");

This code connects to an existing LocalConnection object with the connection name conn1 and invokes the doMessage() method in the remote application. If you want to send parameters to the remote application, you specify additional arguments after the method name in the send() method, as the following snippet shows:

    sendingConnection.send("conn1", "doMessage", "Hello world");

Connecting to content in different domains and to other AIR applications

To allow communications only from specific domains, you call the allowDomain() or allowInsecureDomain() method of the LocalConnection class and pass a list of one or more domains that are allowed to access this LocalConnection object, passing one or more names of domains to be allowed.

There are two special values that you can pass to the LocalConnection.allowDomain() and LocalConnection.allowInsecureDomain() methods: * and localhost. The asterisk value (*) allows access from all domains. The string localhost allows calls to the application from content locally installed, but outside of the application resource directory.

If the LocalConnection.send() method attempts to communicate with an application from a security sandbox to which the calling code does not have access, a securityError event(SecurityErrorEvent.SECURITY_ERROR) is dispatched. To work around this error, you can specify the caller's domain in the receiver's LocalConnection.allowDomain() method.

If you implement communication only between content in the same domain, you can specify a connectionName parameter that does not begin with an underscore (_) and does not specify a domain name (for example, myDomain:connectionName). Use the same string in the LocalConnection.connect(connectionName) command.
If you implement communication between content in different domains, you specify a `connectionName` parameter that begins with an underscore. Specifying the underscore makes the content with the receiving LocalConnection object more portable between domains. Here are the two possible cases:

- If the string for `connectionName` does not begin with an underscore, the runtime adds a prefix with the superdomain name and a colon (for example, `myDomain:connectionName`). Although this ensures that your connection does not conflict with connections of the same name from other domains, any sending LocalConnection objects must specify this superdomain (for example, `myDomain:connectionName`). If you move the HTML or SWF file with the receiving LocalConnection object to another domain, the runtime changes the prefix to reflect the new superdomain (for example, `anotherDomain:connectionName`). All sending LocalConnection objects have to be manually edited to point to the new superdomain.

- If the string for `connectionName` begins with an underscore (for example, `_connectionName`), the runtime does not add a prefix to the string. This means the receiving and sending LocalConnection objects use identical strings for `connectionName`. If the receiving object uses `LocalConnection.allowDomain()` to specify that connections from any domain will be accepted, you can move the HTML or SWF file with the receiving LocalConnection object to another domain without altering any sending LocalConnection objects.

A downside to using underscore names in `connectionName` is the potential for collisions, such as when two applications both try to connect using the same `connectionName` value. A second, related downside is security-related. Connection names that use underscore syntax do not identify the domain of the listening application. For these reasons, domain-qualified names are preferred.

For content running in the AIR application security sandbox (content installed with the AIR application), in place of the domain used by SWF content running in the browser, AIR uses the string `app#` followed by the application ID for the AIR application (defined in the application descriptor file), followed by a dot (.) character, followed by the publisher ID for the application (if defined). For example, a `connectionName` for an application with the application ID `com.example.air.MyApp` and the publisher ID `B146A943FBD637B68C33D022304CEA236D129B4.1` resolves to "app#com.example.air.MyApp.B146A943FBD637B68C33D022304CEA236D129B4.1:myConnection". If the application does not have a publisher ID, the string is: "app#com.example.air.MyApp:myConnection". (For more information, see “Defining the application identity” on page 118 and “Getting the application and publisher identifiers” on page 336.)

When you allow another AIR application to communicate with your application through the local connection, you must call the `allowDomain()` of the LocalConnection object, passing in the local connection domain name. For an AIR application, this domain name is formed from the application id and publisher ID in the same fashion as the connection string. For example, if the sending AIR application has an application ID of `com.example.air.FriendlyApp` and a publisher ID of `214649436BD677B62C3303043EA236D13934.1`, then the domain string that you would use to allow this application to connect is: "app#com.example.air.FriendlyApp.214649436BD677B62C3303043EA236D13934.1". As of AIR 1.5.3, not all applications have publisher IDs. In this case, omit the publisher ID (and the dot separator after the application ID).
Chapter 36: Distributing, Installing, and Running AIR applications

AIR applications are distributed as a single AIR installation file, which contains the application code and all assets. You can distribute this file through any of the typical means, such as by download, by e-mail, or by physical media such as a CD-ROM. Users can install the application by double-clicking the AIR file. You can use the seamless install feature, which lets users install your AIR application (and Adobe® AIR®, if needed) by clicking a single link on a web page.

Before it can be distributed, an AIR installation file must be packaged and signed with a code-signing certificate and private key. Digitally signing the installation file provides assurance that your application has not been altered since it was signed. In addition, if a trusted certification authority issued the digital certificate, your users can confirm your identity as the publisher and signer. The AIR file is signed when the application is packaged with the AIR Developer Tool (ADT).

For information about how to package an application into an AIR file using Adobe® Dreamweaver®, see “Creating an AIR application in Dreamweaver” on page 18.

For information about how to package an application into an AIR file using the Adobe® AIR® SDK, see “Packaging an AIR installation file using the AIR Developer Tool (ADT)” on page 25.

Installing and running an AIR application from the desktop

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.
   The Adobe AIR must already be installed on the computer.

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 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.

3 When the installation is complete, click Finish.

On Mac OS, to install an updated version of an application, the user needs adequate system privileges to install to the application directory. On Windows and Linux, a user needs administrative privileges.
An application can also install a new version via ActionScript or JavaScript. For more information, see “Updating AIR applications” on page 376.

Once the AIR application is installed, a user simply double-clicks the application icon to run it, just like any other desktop application.

- On Windows, double-click the application’s icon (which is either installed on the desktop or in a folder) or select the application from the Start menu.
- On Linux, double-click the application’s icon (which is either installed on the desktop or in a folder) or select the application from the applications menu.
- On Mac OS, double-click the application in the folder in which it was installed. The default installation directory is the /Applications directory.

The AIR seamless install feature lets a user install an AIR application by clicking a link in a web page. The AIR browser invocation features lets a user run an installed AIR application by clicking a link in a web page. These features are described in the following section.

Installing and running AIR applications from a web page

The seamless install feature lets you embed a SWF file in a web page that lets the user install an AIR application from the browser. If the runtime is not installed, the seamless install feature installs the runtime. The seamless install feature lets users install the AIR application without saving the AIR file to their computer. Included in the AIR SDK is a badge.swf file, which lets you easily use the seamless install feature. For details, see “Using the badge.swf file to install an AIR application” on page 362.

For a demonstration of how to use the seamless install feature, see the Distributing an AIR Application via the Web (http://www.adobe.com/go/learn_air_qs_seamless_install_en) quick start sample article.

About customizing the seamless install badge.swf

In addition to using the badge.swf file provided with the SDK, you can create your own SWF file for use in a browser page. Your custom SWF file can interact with the runtime in the following ways:

- It can install an AIR application. See “Installing an AIR application from the browser” on page 367.
- It can check to see if a specific AIR application is installed. See “Checking from a web page if an AIR application is installed” on page 366.
- It can check to see if the runtime is installed. See “Checking if the runtime is installed” on page 365.
- It can launch an installed AIR application on the user’s system. See “Launching an installed AIR application from the browser” on page 367.

These capabilities are all provided by calling APIs in a SWF file hosted at adobe.com: air.swf. This section describes how to use and customize the badge.swf file and how to call the air.swf APIs from your own SWF file.

Additionally, a SWF file running in the browser can communicate with a running AIR application by using the LocalConnection class. For more information, see “Inter-application communication” on page 357.
**Important:** The features described in this section (and the APIs in the air.swf file) require the end user to have Adobe® Flash® Player 9 update 3 installed in the web browser on Windows or Mac OS. On Linux, the seamless install feature requires Flash Player 10 (version 10,0,12,36 or later). You can write code to check the installed version of Flash Player and provide an alternate interface to the user if the required version of Flash Player is not installed. For instance, if an older version of Flash Player is installed, you could provide a link to the download version of the AIR file (instead of using the badge.swf file or the air.swf API to install an application).

### Using the badge.swf file to install an AIR application

Included in the AIR SDK is a badge.swf file which lets you easily use the seamless install feature. The badge.swf can install the runtime and an AIR application from a link in a web page. The badge.swf file and its source code are provided to you for distribution on your webiste.

The instructions in this section provide information on setting parameters of the badge.swf file provided by Adobe. We also provide the source code for the badge.swf file, which you can customize.

### Embedding the badge.swf file in a web page

1. Locate the following files, provided in the samples/badge directory of the AIR SDK, and add them to your web server.
   - badge.swf
   - default_badge.html
   - AC_RunActiveContent.js

2. Open the default_badge.html page in a text editor.

3. In the default_badge.html page, in the `AC_FL_RunContent()` JavaScript function, adjust the `FlashVars` parameter definitions for the following:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>appname</td>
<td>The name of the application, displayed by the SWF file when the runtime is not installed.</td>
</tr>
<tr>
<td>appurl</td>
<td>(Required). The URL of the AIR file to be downloaded. You must use an absolute, not relative, URL.</td>
</tr>
<tr>
<td>airversion</td>
<td>(Required). For the 1.0 version of the runtime, set this to 1.0.</td>
</tr>
<tr>
<td>imageurl</td>
<td>The URL of the image (optional) to display in the badge.</td>
</tr>
<tr>
<td>buttoncolor</td>
<td>The color of the download button (specified as a hex value, such as FFCC00).</td>
</tr>
<tr>
<td>messagecolor</td>
<td>The color of the text message displayed below the button when the runtime is not installed (specified as a hex value, such as FFCC00).</td>
</tr>
</tbody>
</table>

4. The minimum size of the badge.swf file is 217 pixels wide by 180 pixels high. Adjust the values of the `width` and `height` parameters of the `AC_FL_RunContent()` function to suit your needs.

5. Rename the default_badge.html file and adjust its code (or include it in another HTML page) to suit your needs.

   **Note:** For the HTML `embed` tag that loads the badge.swf file, do not set the `wmode` attribute; leave it set to the default setting ("window"). Other `wmode` settings will prevent installation on some systems. Also, using other `wmode` settings produces an error: "Error #2044: Unhandled ErrorEvent:: text=Error #2074: The stage is too small to fit the download ui."

You can also edit and recompile the badge.swf file. For details, see “Modifying the badge.swf file” on page 363.
Installing the AIR application from a seamless install link in a web page

Once you have added the seamless install link to a page, the 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 9 update 3 or later on Windows and Mac OS, or version 10 on Linux) installed.

2. In the web page, click the link in the badge.swf file.
   - If you have installed the runtime, skip to the next step.
   - If you have not installed the runtime, a dialog box is displayed asking whether you would like to install it. Install the runtime (see "Adobe AIR installation" on page 1), 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 `c:\Program Files\`
     - Creates a desktop shortcut for application
     - Creates a Start Menu shortcut
     - Adds an entry for application in the Add/Remove Programs Control Panel
   - On Mac OS, the installer adds the application to the Applications directory (for example, in the `/Applications` directory in Mac OS).
   - On a Linux computer, AIR automatically does the following:
     - Installs the application into `/opt`.
     - Creates a desktop shortcut for application
     - Creates a Start Menu shortcut
     - Adds an entry for application in the system package manager

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

5. When the installation is complete, click Finish.

Modifying the badge.swf file

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

<table>
<thead>
<tr>
<th>Source files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>badge.fla</td>
<td>The source Flash file used to compile the badge.swf file. The badge.fla file compiles into a SWF 9 file (which can be loaded in Flash Player).</td>
</tr>
<tr>
<td>AIRBadge.as</td>
<td>An ActionScript 3.0 class that defines the base class used in the basdge.fla file.</td>
</tr>
</tbody>
</table>

You can use Flash CS3 or Flash CS4 to redesign the visual interface of the badge.fla file.

The `AIRBadge()` constructor function, defined in the AIRBadge class, loads the air.swf file hosted at `http://airdownload.adobe.com/air/browserapi/air.swf`. The air.swf file includes code for using the seamless install feature.

The `onInit()` method (in the AIRBadge class) is invoked when the air.swf file is loaded successfully:
private function onInit(e:Event):void {
    _air = e.target.content;
    switch (_air.getStatus()) {
        case "installed":
            root.statusMessage.text = "";
            break;
        case "available":
            if (_appName && _appName.length > 0) {
                root.statusMessage.htmlText = "<p align='center'><font color='#" + _messageColor + ">In order to run " + _appName + ", this installer will also set up Adobe® AIR®.</font></p>";
            } else {
                root.statusMessage.htmlText = "<p align='center'><font color='#" + _messageColor + ">In order to run this application, " + "this installer will also set up Adobe® AIR®.</font></p>";
            }
            break;
        case "unavailable":
            root.statusMessage.htmlText = "<p align='center'><font color='#" + _messageColor + ">Adobe® AIR® is not available for your system.</font></p>";
            root.buttonBg_mc.enabled = false;
            break;
    }
}

The code sets the global _air variable to the main class of the loaded air.swf file. This class includes the following public methods, which the badge.swf file accesses to call seamless install functionality:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getStatus()</td>
<td>Determines whether the runtime is installed (or can be installed) on the computer. For details, see &quot;Checking if the runtime is installed&quot; on page 365.</td>
</tr>
<tr>
<td>installApplication()</td>
<td>Installs the specified application on the user's machine. For details, see &quot;Installing an AIR application from the browser&quot; on page 367.</td>
</tr>
<tr>
<td></td>
<td>• url—A string defining the URL. You must use an absolute, not relative, URL path.</td>
</tr>
<tr>
<td></td>
<td>• runtimeVersion—A string indicating the version of the runtime (such as &quot;1.0.56&quot;) required by the application to be installed.</td>
</tr>
<tr>
<td></td>
<td>• arguments—Arguments to be passed to the application if it is launched upon installation. The application is launched upon installation if the allowBrowserInvocation element is set to true in the application descriptor file. (For more information on the application descriptor file, see “Setting AIR application properties” on page 116.) If the application is launched as the result of a seamless install from the browser (with the user choosing to launch upon installation), the application's NativeApplication object dispatches a BrowserInvokeEvent object only if arguments are passed. Consider the security implications of data that you pass to the application. For details, see &quot;Launching an installed AIR application from the browser&quot; on page 367.</td>
</tr>
</tbody>
</table>

The settings for url and runtimeVersion are passed into the SWF file via the FlashVars settings in the container HTML page.

If the application starts automatically upon installation, you can use LocalConnection communication to have the installed application contact the badge.swf file upon invocation. For details, see "Inter-application communication" on page 357.
You may also call the `getApplicationVersion()` method of the `air.swf` file to check if an application is installed. You can call this method either before the application installation process or after the installation is started. For details, see “Checking from a web page if an AIR application is installed” on page 366.

**Loading the air.swf file**

You can create your own SWF file that uses the APIs in the `air.swf` file to interact with the runtime and AIR applications from a web page in a browser. The `air.swf` file is hosted at http://airdownload.adobe.com/air/browserapi/air.swf. To reference the `air.swf` APIs from your SWF file, load the `air.swf` file into the same application domain as your SWF file. The following code shows an example of loading the `air.swf` file into the application domain of the loading SWF file:

```actionscript
var airSWF:Object; // This is the reference to the main class of air.swf
var airSWFLoader:Loader = new Loader(); // Used to load the SWF
var loaderContext:LoaderContext = new LoaderContext();
    // Used to set the application domain
loaderContext.applicationDomain = ApplicationDomain.currentDomain;

airSWFLoader.contentLoaderInfo.addEventListener(Event.INIT, onInit);
airSWFLoader.load(new URLRequest("http://airdownload.adobe.com/air/browserapi/air.swf"), loaderContext);

function onInit(e:Event):void
{
    airSWF = e.target.content;
}
```

Once the `air.swf` file is loaded (when the `Loader` object’s `contentLoaderInfo` object dispatches the `init` event), you can call any of the `air.swf` APIs. These APIs are described in these sections:

- “Checking if the runtime is installed” on page 365
- “Checking from a web page if an AIR application is installed” on page 366
- “Installing an AIR application from the browser” on page 367
- “Launching an installed AIR application from the browser” on page 367

**Note:** The badge.swf file, provided with the AIR SDK, automatically loads the `air.swf` file. See “Using the badge.swf file to install an AIR application” on page 362. The instructions in this section apply to creating your own SWF file that loads the `air.swf` file.

**Checking if the runtime is installed**

A SWF file can check if the runtime is installed by calling the `getStatus()` method in the `air.swf` file loaded from http://airdownload.adobe.com/air/browserapi/air.swf. For details, see “Loading the air.swf file” on page 365.

Once the `air.swf` file is loaded, the SWF file can call the `air.swf` file’s `getStatus()` method as in the following:

```actionscript
var status:String = airSWF.getStatus();
```

The `getStatus()` method returns one of the following string values, based on the status of the runtime on the computer:
The `getStatus()` method throws an error if the required version of Flash Player (version 9 update 3 or later on Windows and Mac OS, or version 10 on Linux) is not installed in the browser.

### Checking from a web page if an AIR application is installed

A SWF file can check if an AIR application (with a matching application ID and publisher ID) is installed by calling the `getApplicationVersion()` method in the `air.swf` file loaded from `http://airdownload.adobe.com/air/browserapi/air.swf`. For details, see “Loading the air.swf file” on page 365.

Once the `air.swf` file is loaded, the SWF file can call the `air.swf` file's `getApplicationVersion()` method as in the following:

```actionscript
var appID:String = "com.example.air.myTestApplication";
var pubID:String = "02D88EEED35F84C264A183921344EAA353A629FD.1";
airSWF.getApplicationVersion(appID, pubID, versionDetectCallback);

function versionDetectCallback(version:String):void
{
    if (version == null)
    {
        trace("Not installed.");
        // Take appropriate actions. For instance, present the user with
        // an option to install the application.
    }
    else
    {
        trace("Version", version, "installed.");
        // Take appropriate actions. For instance, enable the
        // user interface to launch the application.
    }
}
```

The `getApplicationVersion()` method has the following parameters:

<table>
<thead>
<tr>
<th>String value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;available&quot;</td>
<td>The runtime can be installed on this computer but currently it is not installed.</td>
</tr>
<tr>
<td>&quot;unavailable&quot;</td>
<td>The runtime cannot be installed on this computer.</td>
</tr>
<tr>
<td>&quot;installed&quot;</td>
<td>The runtime is installed on this computer.</td>
</tr>
</tbody>
</table>

The `getApplicationVersion()` method throws an error if the required version of Flash Player (version 9 update 3 or later on Windows and Mac OS, or version 10 on Linux) is not installed in the browser.
As of AIR 1.5.3, the publisher ID is deprecated. Publisher IDs are no longer assigned to an application automatically. For backward compatibility, applications can continue to specify a publisher ID.

Installing an AIR application from the browser

A SWF file can install an AIR application by calling the `installApplication()` method in the air.swf file loaded from http://airdownload.adobe.com/air/browserapi/air.swf. For details, see “Loading the air.swf file” on page 365.

Once the air.swf file is loaded, the SWF file can call the air.swf file’s `installApplication()` method, as in the following code:

```javascript
var url:String = "http://www.example.com/myApplication.air";
var runtimeVersion:String = "1.0";
var arguments:Array = ["launchFromBrowser"]; // Optional
airSWF.installApplication(url, runtimeVersion, arguments);
```

The `installApplication()` method installs the specified application on the user’s machine. This method has the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>A string defining the URL of the AIR file to install. You must use an absolute, not relative, URL path.</td>
</tr>
<tr>
<td>runtimeVersion</td>
<td>A string indicating the version of the runtime (such as “1.0”) required by the application to be installed.</td>
</tr>
<tr>
<td>arguments</td>
<td>An array of arguments to be passed to the application if it is launched upon installation. Only alphanumerical characters are recognized in the arguments. If you need to pass other values, consider using an encoding scheme. The application is launched upon installation if the <code>allowBrowserInvocation</code> element is set to <code>true</code> in the application descriptor file. (For more information on the application descriptor file, see “Setting AIR application properties” on page 116.) If the application is launched as the result of a seamless install from the browser (with the user choosing to launch upon installation), the application’s NativeApplication object dispatches a BrowserInvokeEvent object only if arguments have been passed. For details, see “Launching an installed AIR application from the browser” on page 367.</td>
</tr>
</tbody>
</table>

The `installApplication()` method can only operate when called in the event handler for a user event, such as a mouse click.

The `installApplication()` method throws an error if the required version of Flash Player (version 9 update 3 or later on Windows and Mac OS, or version 10 on Linux) is not installed in the browser.

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

You may also call the `getApplicationVersion()` method of the air.swf file to check if an application is already installed. You can call this method either before the application installation process begins or after the installation is started. For details, see “Checking from a web page if an AIR application is installed” on page 366. Once the application is running, it can communicate with the SWF content in the browser by using the LocalConnection class. For details, see “Inter-application communication” on page 357.

Launching an installed AIR application from the browser

To use the browser invocation feature (allowing it to be launched from the browser), the application descriptor file of the target application must include the following setting:
<allowBrowserInvocation>true</allowBrowserInvocation>

For more information on the application descriptor file, see “Setting AIR application properties” on page 116.

A SWF file in the browser can launch an AIR application by calling the launchApplication() method in the air.swf file loaded from http://airdownload.adobe.com/air/browserapi/air.swf. For details, see “Loading the air.swf file” on page 365.

Once the air.swf file is loaded, the SWF file can call the air.swf file’s launchApplication() method, as in the following code:

```javascript
var appID:String = "com.example.air.myTestApplication";
var pubID:String = "02D88EEED35F84C264A183921344EEA353A629FD.1";
var arguments:Array = ["launchFromBrowser"]; // Optional
airSWF.launchApplication(appID, pubID, arguments);
```

The launchApplication() method is defined at the top level of the air.swf file (which is loaded in the application domain of the user interface SWF file). Calling this method causes AIR to launch the specified application (if it is installed and browser invocation is allowed, via the allowBrowserInvocation setting in the application descriptor file). The method has the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>appID</td>
<td>The application ID for the application to launch. For details, see “Defining the application identity” on page 118.</td>
</tr>
<tr>
<td>pubID</td>
<td>The publisher ID for the application to launch. For details, see “About AIR publisher identifiers” on page 370. If the application in question does not have a publisher ID, set the pubID parameter to an empty string (“”).</td>
</tr>
<tr>
<td>arguments</td>
<td>An array of arguments to pass to the application. The NativeApplication object of the application dispatches a BrowserInvokeEvent event that has an arguments property set to this array. Only alphanumerical characters are recognized in the arguments. If you need to pass other values, consider using an encoding scheme.</td>
</tr>
</tbody>
</table>

The launchApplication() method can only operate when called in the event handler for a user event, such as a mouse click.

The launchApplication() method throws an error if the required version of Flash Player (version 9 update 3 or later on Windows and Mac OS, or version 10 on Linux) is not installed in the browser.

If the allowBrowserInvocation element is set to false in the application descriptor file, calling the launchApplication() method has no effect.

Before presenting the user interface to launch the application, you may want to call the getApplicationVersion() method in the air.swf file. For details, see “Checking from a web page if an AIR application is installed” on page 366.

When the application is invoked via the browser invocation feature, the application’s NativeApplication object dispatches a BrowserInvokeEvent object. For details, see “Browser invocation” on page 332.

If you use the browser invocation feature, be sure to consider security implications, described in “Browser invocation” on page 332.

Once the application is running, it can communicate with the SWF content in the browser by using the LocalConnection class. For details, see “Inter-application communication” on page 357.

**Note:** As of AIR 1.5.3, the publisher ID is deprecated. Publisher IDs are no longer assigned to an application automatically. For backward compatibility, applications can continue to specify a publisher ID.
Enterprise deployment

IT administrators can install the Adobe AIR runtime and AIR applications silently using standard desktop deployment tools. IT administrators can do the following:

- Silently install the Adobe AIR runtime using tools such as Microsoft SMS, IBM Tivoli, or any deployment tool that allows silent installations that use a bootstrapper
- Silently install the AIR application using the same tools used to deploy the runtime

For more information, see the Adobe AIR Administrator’s Guide (http://www.adobe.com/go/learn_air_admin_guide_en).

Digitally signing an AIR file

Digitally signing your AIR installation files with a certificate issued by a recognized certification authority (CA) provides significant assurance to your users that the application they are installing has not been accidentally or maliciously altered and identifies you as the signer (publisher). AIR displays the publisher name during installation when the AIR application has been signed with a certificate that is trusted, or which chains to a certificate that is trusted on the installation computer. Otherwise the publisher name is displayed as “Unknown.”

Important: A malicious entity could forge an AIR file with your identity if it somehow obtains your signing keystore file or discovers your private key.

Information about code-signing certificates

The security assurances, limitations, and legal obligations involving the use of code-signing certificates are outlined in the Certificate Practice Statements (CPS) and subscriber agreements published by the issuing certification authority. For more information about the agreements for the certification authorities that currently issue AIR code signing certificates, refer to:

- ChosenSecurity (http://www.chosensecurity.com/products/tc_publisher_id_adobe_air.htm)
- ChosenSecurity CPS (http://www.chosensecurity.com/resource_center/repository.htm)
- GlobalSign (http://www.globalsign.com/developer/code-signing-certificate/index.htm)
- GlobalSign CPS (http://www.globalsign.com/repository/index.htm)
- Thawte CPS (http://www.thawte.com/cps/index.html)
- VeriSign CPS (http://www.verisign.com/repository/CPS/)
- VeriSign Subscriber’s Agreement (https://www.verisign.com/repository/subscriber/SUBAGR.html)

About AIR code signing

When an AIR file is signed, a digital signature is included in the installation file. The signature includes a digest of the package, which is used to verify that the AIR file has not been altered since it was signed, and it includes information about the signing certificate, which is used to verify the publisher identity.
AIR uses the public key infrastructure (PKI) supported through the operating system’s certificate store to establish whether a certificate can be trusted. The computer on which an AIR application is installed must either directly trust the certificate used to sign the AIR application, or it must trust a chain of certificates linking the certificate to a trusted certification authority in order for the publisher information to be verified.

If an AIR file is signed with a certificate that does not chain to one of the trusted root certificates (and normally this includes all self-signed certificates), then the publisher information cannot be verified. While AIR can determine that the AIR package has not been altered since it was signed, there is no way to know who actually created and signed the file.

**Note:** A user can choose to trust a self-signed certificate and then any AIR applications signed with the certificate displays the value of the common name field in the certificate as the publisher name. AIR does not provide any means for a user to designate a certificate as trusted. The certificate (not including the private key) must be provided to the user separately and the user must use one of the mechanisms provided by the operating system or an appropriate tool to import the certificate into the proper location in system certificate store.

### About AIR publisher identifiers

**Important:** As of AIR 1.5.3 the publisher ID is deprecated and no longer computed based on the code signing certificate. New applications do not need and should not use a publisher ID. When updating existing applications, you must specify the original publisher ID in the application descriptor file.

Prior to AIR 1.5.3, the AIR application installer generated a publisher ID during the installation of an AIR file. This was an identifier that is unique to the certificate used to sign the AIR file. If you reused the same certificate for multiple AIR applications, they received the same publisher ID. Signing an application update with a different certificate and sometimes even a renewed instance of the original certificate changed the publisher ID.

In AIR 1.5.3 and later, a publisher ID is not assigned by AIR. An application published with AIR 1.5.3 can specify a publisher ID string in the application descriptor. You should only specify a publisher ID when publishing updates for applications originally published for versions of AIR prior to 1.5.3. If you do not specifying the original ID in the application descriptor then the new AIR package is not treated as an update of the existing application.

To determine the original publisher ID, find the `publisherid` file in the META-INF/AIR subdirectory where the original application is installed. The string within this file is the publisher ID. Your application descriptor must specify the AIR 1.5.3 runtime (or later) in the namespace declaration of the application descriptor file in order to specify the publisher ID manually.

The publisher ID, when present, is used for the following purposes:

- As part of the encryption key for the encrypted local store
- As part of the path for the application storage directory
- As part of the connection string for local connections
- As part of the identity string used to invoke an application with the AIR in-browser API
- As part of the OSID (used when creating custom install/uninstall programs)

When a publisher ID changes, the behavior of any AIR features relying on the ID also changes. For example, data in the existing encrypted local store can no longer be accessed and any Flash or AIR instances that create a local connection to the application must use the new ID in the connection string. The publisher ID cannot change in AIR 1.5.3, or later. If you use a different publisher ID when publishing an AIR package, the installer treats the new package as a different application.
About Certificate formats

The AIR signing tools accept any keystores accessible through the Java Cryptography Architecture (JCA). This includes file-based keystores such as PKCS12-format files (which typically use a .pfx or .p12 file extension), Java .keystore files, PKCS11 hardware keystores, and the system keystores. The keystore formats that ADT can access depend on the version and configuration of the Java runtime used to run ADT. Accessing some types of keystore, such as PKCS11 hardware tokens, may require the installation and configuration of additional software drivers and JCA plug-ins.

To sign AIR files, you can use most existing code signing certificates or you can obtain a new one issued expressly for signing AIR applications. For example, any of the following types of certificate from VeriSign, Thawte, GlobalSign, or ChosenSecurity can be used:

- ChosenSecurity
  - TC Publisher ID for Adobe AIR
- GlobalSign
  - ObjectSign Code Signing Certificate
- Thawte:
  - AIR Developer Certificate
  - Apple Developer Certificate
  - JavaSoft Developer Certificate
  - Microsoft Authenticode Certificate
- VeriSign:
  - Adobe AIR Digital ID
  - Microsoft Authenticode Digital ID
  - Sun Java Signing Digital ID

Note: The certificate must be created for code signing. You cannot use an SSL or other type of certificate to sign AIR files.

Time stamps

When you sign an AIR file, the packaging tool queries the server of a timestamp authority to obtain an independently verifiable date and time of signing. The time stamp obtained is embedded in the AIR file. As long as the signing certificate is valid at the time of signing, the AIR file can be installed, even after the certificate has expired. On the other hand, if no time stamp is obtained, the AIR file ceases to be installable when the certificate expires or is revoked.

By default, the AIR packaging tools obtain a time stamp. However, to allow applications to be packaged when the time-stamp service is unavailable, you can turn time stamping off. Adobe recommends that all publicly distributed AIR files include a time stamp.

The default time-stamp authority used by the AIR packaging tools is Geotrust.

Obtaining a certificate

To obtain a certificate, you would normally visit the certification authority web site and complete the company’s procurement process. The tools used to produce the keystore file needed by the AIR tools depend on the type of certificate purchased, how the certificate is stored on the receiving computer, and, in some cases, the browser used to obtain the certificate. For example, to obtain and export an Adobe Developer certificate from Thawte you must use Mozilla Firefox. The certificate can then be exported as a .p12 or .pfx file directly from the Firefox user interface.
You can generate a self-signed certificate using the Air Development Tool (ADT) used to package AIR installation files. Some third-party tools can also be used.

For instructions on how to generate a self-signed certificate, as well as instructions on signing an AIR file, see “Packaging an AIR installation file using the AIR Developer Tool (ADT)” on page 25. You can also export and sign AIR files using Flex Builder, Dreamweaver, and the AIR update for Flash.

The following example describes how to obtain an AIR Developer Certificate from the Thawte Certification Authority and prepare it for use with ADT.

Example: Getting an AIR Developer Certificate from Thawte

Note: This example illustrates only one of the many ways to obtain and prepare a code signing certificate for use. Each certification authority has its own policies and procedures.

To purchase an AIR Developer Certificate, the Thawte web site requires you to use the Mozilla Firefox browser. The private key for the certificate is stored within the browser’s keystore. Ensure that the Firefox keystore is secured with a master password and that the computer itself is physically secure. (You can export and remove the certificate and private key from the browser keystore once the procurement process is complete.)

As part of the certificate enrollment process a private/public key pair is generated. The private key is automatically stored within the Firefox keystore. You must use the same computer and browser to both request and retrieve the certificate from Thawte’s web site.

1. Visit the Thawte web site and navigate to the Product page for Code Signing Certificates.
2. From the list of Code Signing Certificates, select the Adobe AIR Developer Certificate.
3. Complete the three step enrollment process. You need to provide organizational and contact information. Thawte then performs its identity verification process and may request additional information. After verification is complete, Thawte will send you e-mail with instructions on how to retrieve the certificate.

   Note: Additional information about the type of documentation required can be found here: https://www.thawte.com/ssl-digital-certificates/free-guides-whitepapers/pdf/enroll_codesign_eng.pdf.

4. Retrieve the issued certificate from the Thawte site. The certificate is automatically saved to the Firefox keystore.
5. Export a keystore file containing the private key and certificate from the Firefox keystore using the following steps:

   Note: When exporting the private key and certificate from Firefox, it is exported in a .p12 (pfx) format which ADT, Flex, Flash, and Dreamweaver can use.

   a. Open the Firefox Certificate Manager dialog:
   b. On Windows: open Tools -> Options -> Advanced -> Encryption -> View Certificates
   c. On Mac OS: open Firefox -> Preferences -> Advanced -> Encryption -> View Certificates
   d. On Linux: open Edit -> Preferences -> Advanced -> Encryption -> View Certificates
   e. Select the Adobe AIR Code Signing Certificate from the list of certificates and click the Backup button.
   f. Enter a file name and the location to which to export the keystore file and click Save.
   g. If you are using the Firefox master password, you are prompted to enter your password for the software security device in order to export the file. (This password is used only by Firefox.)
   h. On the Choose a Certificate Backup Password dialog box, create a password for the keystore file.

   Important: This password protects the keystore file and is required when the file is used for signing AIR applications. A secure password should be chosen.
i Click OK. You should receive a successful backup password message. The keystore file containing the private key and certificate is saved with a .p12 file extension (in PKCS12 format)

6 Use the exported keystore file with ADT, Flex Builder, Flash, or Dreamweaver. The password created for the file is required whenever an AIR application is signed.

**Important:** The private key and certificate are still stored within the Firefox keystore. While this permits you to export an additional copy of the certificate file, it also provides another point of access that must be protected to maintain the security of your certificate and private key.

### Changing certificates

In some circumstances, you must change the certificate you use to sign updates for your AIR application. Such circumstances include:

- Renewing the original signing certificate.
- Upgrading from a self-signed certificate to a certificate issued by a certification authority
- Changing from a self-signed certificate that is about to expire to another
- Changing from one commercial certificate to another, for example, when your corporate identity changes

For AIR to recognize an AIR file as an update, you must either sign both the original and update AIR files with the same certificate or apply a certificate migration signature to the update. A migration signature is a second signature applied to the update AIR package using the original certificate. The migration signature uses the original certificate to establish that the signer is the original publisher of the application.

After an AIR file with a migration signature is installed, the new certificate becomes the primary certificate. Subsequent updates do not require a migration signature. However, you should apply migration signatures for as long as possible to accommodate users who skip updates.

**Important:** The certificate must be changed before the original certificate expires. If you do not create an update signed with a migration signature before your certificate expires, users will have to uninstall their existing version of your application before installing a new version. As of AIR 1.5.3, an expired certificate can be used to apply a migration signature within a 180 day grace period after the certificate has expired. (You cannot use the expired certificate to apply the main application signature.)

To change certificates:

1. Create an update to your application
2. Package and sign the update AIR file with the **new** certificate
3. Sign the AIR file again with the **original** certificate (using the ADT `-migrate` command)

An AIR file with a migration signature is, in other respects, a normal AIR file. If the application is installed on a system without the original version, AIR installs the new version in the usual manner.

**Note:** Prior to AIR 1.5.3, signing an AIR application with a renewed certificate did not always require a migration signature. Starting with AIR 1.5.3, a migration signature is always required for renewed certificates.

The procedure for applying a migration signature is described in “Signing an AIR file to change the application certificate” on page 35.

### Application identity changes

Prior to AIR 1.5.3, the identity of an AIR application changed when an update signed with a migration signature was installed. Changing the identity of an application has the several repercussions, including:

- The new application version cannot access data in the existing encrypted local store.
• The location of the application storage directory changes. Data in the old location is not copied to the new directory. (But the new application can locate the original directory based on the old publisher ID).
• The application can no longer open local connections using the old publisher ID.
• The identity string used to access an application from a web page changes.
• The OSID of the application changes. (The OSID is used when writing custom install/uninstall programs.)
When publishing an update with AIR 1.5.3, the application identity cannot change. The original application and publisher IDs must be specified in the application descriptor of the update AIR file. Otherwise, the new package is not recognized as an update.

Note: When publishing a new AIR application with AIR 1.5.3 or later, you should not specify a publisher ID.

Expired certificates
As of AIR 1.5.3, a certificate that has expired within the last 180 days can still be used to apply a migration signature to an application update. To take advantage of this grace period, and the update application descriptor must specify 1.5.3 in the namespace attribute. After the grace period, the certificate can no longer be used. Users updating to a new version of your application will have to uninstall their existing version. Note that there is no grace period in AIR versions before 1.5.3.

Terminology
This section provides a glossary of some of the key terminology you should understand when making decisions about how to sign your application for public distribution.

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Certification Authority (CA)</td>
<td>An entity in a public-key infrastructure network that serves as a trusted third party and ultimately certifies the identity of the owner of a public key. A CA normally issues digital certificates, signed by its own private key, to attest that it has verified the identity of the certificate holder.</td>
</tr>
<tr>
<td>Certificate Practice Statement (CPS)</td>
<td>Sets forth the practices and policies of the certification authority in issuing and verifying certificates. The CPS is part of the contract between the CA and its subscribers and relying parties. It also outlines the policies for identity verification and the level of assurances offered by the certificates they provide.</td>
</tr>
<tr>
<td>Certificate Revocation List (CRL)</td>
<td>A list of issued certificates that have been revoked and should no longer be relied upon. AIR checks the CRL at the time an AIR application is signed, and, if no timestamp is present, again when the application is installed.</td>
</tr>
<tr>
<td>Certificate chain</td>
<td>A certificate chain is a sequence of certificates in which each certificate in the chain has been signed by the next certificate.</td>
</tr>
<tr>
<td>Digital Certificate</td>
<td>A digital document that contains information about the identity of the owner, the owner’s public key, and the identity of the certificate itself. A certificate issued by a certification authority is itself signed by a certificate belonging to the issuing CA.</td>
</tr>
<tr>
<td>Digital Signature</td>
<td>An encrypted message or digest that can only be decrypted with the public key half of a public-private key pair. In a PKI, a digital signature contains one or more digital certificates that are ultimately traceable to the certification authority. A digital signature can be used to validate that a message (or computer file) has not been altered since it was signed (within the limits of assurance provided by the cryptographic algorithm used), and, assuming one trusts the issuing certification authority, the identity of the signer.</td>
</tr>
<tr>
<td>Keystore</td>
<td>A database containing digital certificates and, in some cases, the related private keys.</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PKCS #11</td>
<td>The Cryptographic Token Interface Standard by RSA Laboratories. A hardware token based keystore.</td>
</tr>
<tr>
<td>Private Key</td>
<td>The private half of a two-part, public-private key asymmetric cryptography system. The private key must be kept secret and should never be transmitted over a network. Digitally signed messages are encrypted with the private key by the signer.</td>
</tr>
<tr>
<td>Public Key</td>
<td>The public half of a two-part, public-private key asymmetric cryptography system. The public key is openly available and is used to decrypt messages encrypted with the private key.</td>
</tr>
<tr>
<td>Public Key Infrastructure (PKI)</td>
<td>A system of trust in which certification authorities attest to the identity of the owners of public keys. Clients of the network rely on the digital certificates issued by a trusted CA to verify the identity of the signer of a digital message (or file).</td>
</tr>
<tr>
<td>Time stamp</td>
<td>A digitally signed datum containing the date and time an event occurred. ADT can include a time stamp from an RFC 3161 compliant time server in an AIR package. When present, AIR uses the time stamp to establish the validity of a certificate at the time of signing. This allows an AIR application to be installed after its signing certificate has expired.</td>
</tr>
<tr>
<td>Time stamp authority</td>
<td>An authority that issues time stamps. To be recognized by AIR, the time stamp must conform to RFC 3161 and the time stamp signature must chain to a trusted root certificate on the installation machine.</td>
</tr>
</tbody>
</table>
Chapter 37: Updating AIR applications

Users can install or update an AIR application by double-clicking an AIR file on their computer or from the browser (using the seamless install feature). The Adobe® AIR® installer application manages the installation, alerting the user if they are updating an already existing application. (See “Distributing, Installing, and Running AIR applications” on page 360.)

However, you can also have an installed application update itself to a new version, using the Updater class. (An installed application may detect that a new version is available to be downloaded and installed.) 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.

Both the application ID and the publisher ID of an update AIR file must match the application to be updated. In AIR 1.5.3 or later, the publisher ID is optional and can be specified in the application descriptor file. In earlier versions of AIR, the publisher ID was derived from the signing certificate, which meant that both the update and the application to be updated had to be signed with the same certificate. When updating an application published before AIR 1.5.3 with an update created with AIR 1.5.3 or later, add the original publisher ID to the application descriptor file of the update. New applications do not need and should not specify a publisher ID.

As of AIR 1.1 and later, you can migrate an application to use a new code-signing certificate. Migrating an application to use a new signature involves signing the update AIR file with both the new and the original certificates. Certificate migration is a one-way process. After the migration, only AIR files signed with the new certificate (or with both certificates) will be recognized as updates to an existing installation.

Managing updates of applications can be complicated. AIR 1.5 includes the new update framework for AdobeAIR applications. This framework provides APIs to assist developers in providing good update capabilities in AIR applications.

You can use certificate migration any time you need to change from using one certificate to another — including changing to a renewed version of an existing certificate. If you do not migrate the certificate, existing users must remove their current version of your application before installing the new version. For more information see “Changing certificates” on page 373.

**Note:** In earlier versions of AIR, you could sometimes sign an application with a renewed certificate without using a migration signature. As of AIR 1.5.3, a migration signature is always required to change certificates — even for renewals.

It is a good practice to include an update mechanism in your application. If you create a new version the application, the update mechanism can prompt the user to install the new version.

**Note:** New versions of the Adobe AIR runtime may include updated versions of WebKit. An updated version of WebKit may result in unexpected changes in HTML content in a deployed AIR application. These changes may require you to update your application. An update mechanism can inform the user of the new version of the application. For more information, see “About the HTML environment” on page 67.

About updating applications

The Updater class (in the flash.desktop 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 = new air.Updater();
var airFile = air.File.desktopDirectory.resolvePath("Sample_App_v2.air");
var version = "2.01";
updater.update(airFile, version);

Before an application uses the Updater class, the user or the application must download the updated version of the AIR file to the computer. For more information, see “Downloading an AIR file to the user’s computer” on page 378.

**Results of the Updater.update() 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 and publisher ID specified in the AIR file matches the application ID and publisher ID for the application calling the `update()` method. (For information on the application ID and publisher ID, see “Setting AIR application properties” on page 116.) 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 reopens the existing (pre-install) version of the application.

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

If the updated version of the application requires an updated version of the runtime, the new runtime version is installed. To update the runtime, a user must have administrative privileges for the computer.

When testing an application using ADL, calling the `update()` method 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 application should also check the version string in the AIR file with version string in the installed application to prevent downgrade attacks.

The version string can be of any format. For example, it can be "2.01" or "version 2". The format of this string is left for you, the application developer, to decide. The runtime does not validate the version string; the application code should do validate this string before updating the application.

If an Adobe 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 second record in the archive.)

For details on the application descriptor file, see “Setting AIR application properties” on page 116.
Presenting a custom application update user interface

AIR includes a default update interface:

![Application Install](image)

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. If your application defines a custom update interface, specify a `customUpdateUI` element in the application descriptor file for the currently installed application:

```xml
<customUpdateUI>true</customUpdateUI>
```

When the application is installed and the user opens an AIR file with an application ID and a publisher ID that match the installed application, the runtime opens the application, rather than the default AIR application installer. For more information, see “Providing a custom user interface for application updates” on page 123.

The application can decide, when it is run (when the `NativeApplication.nativeApplication` object dispatches an `load` 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, the user or the application must first save an AIR file locally to the user’s computer.

**Note:** AIR 1.5 includes an update framework, which assists developers in providing good update capabilities in AIR applications. Using this framework may be much easier than using the `update()` method of the `Updater` class directly. For details, see “Using the update framework” on page 381.

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 application storage directory:
Updating AIR applications

var urlString = "http://example.com/air/updates/Sample_App_v2.air";
var urlReq = new air.URLRequest(urlString);
var urlStream = new air.URLStream();
var fileData = new air.ByteArray();
urlStream.addEventListener(air.Event.COMPLETE, loaded);
urlStream.load(urlReq);

function loaded(event) {
    urlStream.readBytes(fileData, 0, urlStream.bytesAvailable);
    writeAirFile();
}

function writeAirFile() {
    var file = air.File.desktopDirectory.resolvePath("My App v2.air");
    var fileStream = new air.FileStream();
    fileStream.open(file, air.FileMode.WRITE);
    fileStream.writeBytes(fileData, 0, fileData.length);
    fileStream.close();
    trace("The AIR file is written.");
}

For more information, see “Workflow for reading and writing files” on page 201.

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 a "getting started" or "welcome" message. Upon launching, the application checks to see if it is running for the first time, so that it can determine whether to display the message.

Note: AIR 1.5 includes an update framework, which assists developers in providing good update capabilities in AIR applications. This framework provides easy methods to check if a version of an application is running for the first time. For details, see “Using the update framework” on page 381.

One way to do this is to save a file to the application store directory upon initializing the application. 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 for the current user. 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.

The following example demonstrates the concept:
If your application saves data locally (such as, in the application storage directory), you may want to check for any previously saved data (from previous versions) upon first run.
Using the update framework

Managing updates of applications can be complicated. The update framework for AdobeAIR applications provides APIs to assist developers in providing good update capabilities in AIR applications. The functionality in the AIR update framework assists developers in the following:

- Periodically checking for updates based on an interval or at the request of the user
- Downloading AIR files (updates) from a web source
- Alerting the user on the first run of the newly installed version
- Confirming that the user wants to check for updates
- Displaying information on the new update version to the user
- Displaying download progress and error information to the user

The AIR update framework supplies a sample user interface that your application can use. It provides the user with basic information and options related to application updates. Your application can also define its own custom user interface for use with the update framework.

The AIR update framework lets you store information about the update version of an AIR application in simple XML configuration files. For most applications, setting up these configuration files and including some basic code provides good update functionality to the end user.

Even without using the update framework, Adobe AIR includes an Updater class that AIR applications can use to upgrade to new versions. This class lets an application upgrade to a version contained in an AIR file on the user’s computer. However, upgrade management can involve more than simply having the application upgrade based on a locally stored AIR file.

Files in the AIR update framework

The AIR update framework includes the following directories:

- doc—The documentation (which you are reading now) for the AIR update framework.
- frameworks—This directory includes SWC files, for Flex development, and SWF files, for HTML development. For more information, see “Including framework files in an HTML-based AIR application” on page 381.
- samples—This directory includes Flex- and HTML-based samples showing how to use the application update framework. Compile and test these files, as you would any AIR application.
- templates—This directory contains sample update descriptor files (simple and localized) and configuration files. (See Setting up your Setting up your Flex development environment and “Basic example: Using the ApplicationUpdaterUI version” on page 382 for more information on these files.)

Including framework files in an HTML-based AIR application

The frameworks/html directory of the update framework includes these SWF files:

- ApplicationUpdater.swf—Defines the basic functionality of the update library, without any user interface
- ApplicationUpdater_UI.swf—Defines the basic functionality of the update library, including a user interface that your application can use to display update options

JavaScript code in AIR applications can use classes defined in SWF files.
To use the update framework, include either the ApplicationUpdater.swf or ApplicationUpdater_UI.swf file in your application directory (or a subdirectory). Then, in the HTML file that will use the framework (in JavaScript code), include a script tag that loads the file:

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

Or use this script tag to load the ApplicationUpdater_UI.swf file:

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

The API defined in these two files is described in the remainder of this document.

**Basic example: Using the ApplicationUpdaterUI version**

The ApplicationUpdaterUI version of the update framework provides a basic interface that you can easily use in your application. The following is a basic example.

First, create an AIR application that calls the update framework:

1. If your application is an HTML-based AIR application, load the ApplicationUpdaterUI.js file:
   ```html
   <script src="ApplicationUpdater_UI.swf" type="application/x-shockwave-flash"/>
   ``

2. In your AIR application program logic, instantiate an ApplicationUpdaterUI object.
   - In ActionScript, use the following code:
     ```actionscript
     var appUpdater:ApplicationUpdaterUI = new ApplicationUpdaterUI();
     ```
   - In JavaScript, use the following code:
     ```javascript
     var appUpdater = new runtime.air.update.ApplicationUpdaterUI();
     ```
   You may want to add this code in an initialization function that executes when the application has loaded.

3. Create a text file named updateConfig.xml and add the following to it:
   ```xml
   <?xml version="1.0" encoding="utf-8"?>
   <configuration xmlns="http://ns.adobe.com/air/framework/update/configuration/1.0">
     <url>http://example.com/updates/update.xml</url>
     <delay>1</delay>
   </configuration>
   ```
   Edit the URL element of the updateConfig.xml file to match the eventual location of the update descriptor file on your web server (see the next procedure).
   - The delay is the number of days the application waits between checks for updates.

4. Add the updateConfig.xml file to the project directory of your AIR application.

5. Have the updater object reference the updateConfig.xml file, and call the object's `initialize()` method.
   - In ActionScript, use the following code:
     ```actionscript
     appUpdater.configurationFile = new File("app:/updateConfig.xml");
     appUpdater.initialize();
     ```
   - In JavaScript, use the following code:
     ```javascript
     appUpdater.configurationFile = new air.File("app:/updateConfig.xml");
     appUpdater.initialize();
     ```

6. Create a second version of the AIR application that has a different version than the first application. (The version is specified in the application descriptor file, in the `version` element.)
Next, add the update version of the AIR application to your web server:

1. Place the update version of the AIR file on your web server.
2. Create a text file named updateDescriptor.xml, and add the following contents to it:

   ```xml
   <?xml version="1.0" encoding="utf-8"?>
   <update xmlns="http://ns.adobe.com/air/framework/update/description/1.0">
   <version>1.1</version>
   <url>http://example.com/updates/sample_1.1.air</url>
   <description>This is the latest version of the Sample application.</description>
   </update>
   
   Edit the version, URL, and description of the updateDescriptor.xml file to match your update AIR file.

3. Add the updateDescriptor.xml file to the same web server directory that contains the update AIR file.

This is a basic example, but it provides update functionality that is sufficient for many applications. The remainder of this document describes how to use the update framework to best suit your needs.

For another example of using the update framework, see the following sample application at the Adobe AIR developer center: [Update Framework in a HTML-based Application](http://www.adobe.com/go/learn_air_qs_update_framework_html_en).

### Defining the update descriptor file and adding the AIR file to your web server

When you use the AIR update framework, you define basic information about the available update in an update descriptor file, stored on your web server. The update descriptor file is a simple XML file. The update framework included in the application checks this file to see if a new version has been uploaded.

The update descriptor file contains the following data:

- **version**—The new version of the air application. It must be the same string that is used in the new air application descriptor file as the version. If the version in the update descriptor file does not match the update AIR file’s version, the update framework will throw an exception.
- **url**—The location of the update AIR file. This is the file that contains the update version of the AIR application.
- **description**—Details regarding the new version. This information can be displayed to the user during the update process.

The `version` and `url` elements are mandatory. The `description` element is optional.

Here is a sample update descriptor file:

```xml
<?xml version="1.0" encoding="utf-8"?>
<update xmlns="http://ns.adobe.com/air/framework/update/description/1.0">
  <version>1.1a1</version>
  <url>http://example.com/updates/sample_1.1a1.air</url>
  <description>This is the latest version of the Sample application.</description>
</update>
```

If you want to define the `description` tag using multiple languages, use multiple `text` elements that define a `lang` attribute:
Place the update descriptor file, along with the update AIR file, on your web server.

The templates directory included with the update descriptor includes sample update descriptor files. These include both single-language and multi-language versions.

### Instantiating an updater object

After loading the AIR update framework in your code (see “Including framework files in an HTML-based AIR application” on page 381), you need to instantiate an updater object, as in the following example:

```javascript
var appUpdater = new runtime.air.update.ApplicationUpdater();
```

The previous code uses the ApplicationUpdater class (which provides no user interface). If you want to use the ApplicationUpdaterUI class (which provides a user interface), use the following:

```javascript
var appUpdater = new runtime.air.update.ApplicationUpdaterUI();
```

The remaining code samples in this document assume that you have instantiated an updater object named `appUpdater`.

### Configuring the update settings

Both ApplicationUpdater and ApplicationUpdaterUI can be configured via a configuration file delivered with the application or via ActionScript or JavaScript in the application.

#### Defining update settings in an XML configuration file

The update configuration file is an XML file. It can contain the following elements:

- **updateURL** — A String. Represents the location of the update descriptor on the remote server. Any valid URLRequest location is allowed. You must define the `updateURL` property, either via the configuration file or via script (see “Defining the update descriptor file and adding the AIR file to your web server” on page 383). You must define this property before using the updater (before calling the `initialize()` method of the updater object, described in “Initializing the update framework” on page 387).

- **delay** — A Number. Represents an interval of time given in days (values like 0.25 are allowed) for checking for updates. A value of 0 (which is the default value) specifies that the updater does not perform an automatic periodical check.

The configuration file for the ApplicationUpdaterUI can contain the following element in addition to the `updateURL` and `delay` elements:

- **defaultUI** — A list of dialog elements. Each dialog element has a `name` attribute that corresponds to dialog box in the user interface. Each dialog element has a `visible` attribute that defines whether the dialog box is visible. The default value is `true`. Possible values for the `name` attribute are the following:
  - **checkForUpdate** — Corresponding to the Check for Update, No Update, and Update Error dialog boxes
Updating AIR applications

When set to false, the corresponding dialog box does not appear as part of the update procedure.

Here is an example of the configuration file for the ApplicationUpdater framework:

```xml
<?xml version="1.0" encoding="utf-8"?>
<configuration xmlns="http://ns.adobe.com/air/framework/update/configuration/1.0">
  <url>http://example.com/updates/update.xml</url>
  <delay>1</delay>
</configuration>
```

Here is an example of the configuration file for the ApplicationUpdaterUI framework, which includes a definition for the defaultUI element:

```xml
<?xml version="1.0" encoding="utf-8"?>
<configuration xmlns="http://ns.adobe.com/air/framework/update/configuration/1.0">
  <url>http://example.com/updates/update.xml</url>
  <delay>1</delay>
  <defaultUI>
    <dialog name="checkForUpdate" visible="false" />
    <dialog name="downloadUpdate" visible="false" />
    <dialog name="downloadProgress" visible="false" />
  </defaultUI>
</configuration>
```

Point the configurationFile property to the location of that file: as in the following

- ActionScript:
  ```javascript
  appUpdater.configurationFile = new File("app:/cfg/updateConfig.xml");
  ```

- JavaScript:
  ```javascript
  appUpdater.configurationFile = new air.File("app:/cfg/updateConfig.xml");
  ```

The templates directory of the update framework includes a sample configuration file, config-template.xml.

### Defining update settings ActionScript or JavaScript code

These configuration parameters can also be set using code in the application, as in the following:

```javascript
appUpdater.updateURL = "http://example.com/updates/update.xml";
appUpdater.delay = 1;
```

The properties of the updater object are updateURL and delay. These properties define the same settings as the updateURL and delay elements in the configuration file: the URL of the update descriptor file and the interval for checking for updates. If you specify a configuration file and settings in code, any properties set using code take precedence over corresponding settings in the configuration file.

You must define the updateURL property, either via the configuration file or via script (see “Defining the update descriptor file and adding the AIR file to your web server” on page 383) before using the updater (before calling the `initialize()` method of the updater object, described in “Initializing the update framework” on page 387).
The ApplicationUpdaterUI framework defines these additional properties of the updater object:

- `isCheckForUpdateVisible`—Corresponding to the Check for Update, No Update, and Update Error dialog boxes
- `isDownloadUpdateVisible`—Corresponding to the Download Update dialog box
- `isDownloadProgressVisible`—Corresponding to Download Progress and Download Error dialog boxes
- `isInstallUpdateVisible`—Corresponding to Install Update dialog box
- `isFileUpdateVisible`—Corresponding to File Update, File No Update, and File Error dialog boxes
- `isUnexpectedErrorVisible`—Corresponding to Unexpected Error dialog box

Each property corresponds to one or more dialog box in the ApplicationUpdaterUI user interface. Each property is a Boolean value, with a default value of `true`. When set to `false` the corresponding dialog boxes do not appear as part of the update procedure.

These dialog box properties override settings in the update configuration file.

**The update process**

The AIR update framework completes the update process in the following steps:

1. The updater initialization checks to see if an update check has been performed within the defined delay interval (see “Configuring the update settings” on page 384). If an update check is due, the update process continues.
2. The updater downloads and interprets the update descriptor file.
3. The updater downloads the update AIR file.
4. The updater installs the updated version of the application.

The updater object dispatches events at the completion of each of these steps. In the ApplicationUpdater version, you can cancel the events that indicate successful completion of a step in the process. If you cancel one of these events, the next step in the process is canceled. In the ApplicationUpdaterUI version, the updater presents a dialog box allowing the user to cancel or proceed at each step in the process.

If you cancel the event, you can call methods of the updater object to resume the process.

As the ApplicationUpdater version of the updater progresses through the update process, it records its current state, in a `currentState` property. This property is set to a string with the following possible values:

- "UNINITIALIZED"—The updater has not been initialized.
- "INITIALIZING"—The updater is initializing.
- "READY"—The updater has been initialized
- "BEFORE_CHECKING"—The updater has not yet checked for the update descriptor file.
- "CHECKING"—The updater is checking for an update descriptor file.
- "AVAILABLE"—The updater descriptor file is available.
- "DOWNLOADING"—The updater is downloading the AIR file.
- "DOWNLOADED"—The updater has downloaded the AIR file.
- "INSTALLING"—The updater is installing the AIR file.
- "PENDING_INSTALLING"—The updater has initialized and there are pending updates.

Some methods of the updater object only execute if the updater is in a certain state.
Initializing the update framework

After setting the configuration properties (see “Basic example: Using the ApplicationUpdaterUI version” on page 382), call the `initialize()` method to initialize the update:

```javascript
appUpdater.initialize();
```

This method does the following:

- It initializes the update framework, silently installing synchronously any pending updates. It is required to call this method during application startup because it may restart the application when it is called.
- It checks if there is a postponed update and installs it;
- If there is an error during the update process, it clears the update file and version information from the application storage area.
- If the delay has expired, it starts the update process. Otherwise it restarts the timer.

Calling this method can result in the updater object dispatching the following events:

```javascript
UpdateEvent.INITIALIZED — Dispatched when the initialization is complete.
ErrorEvent.ERROR — Dispatched when there is an error during initialization.
```

Upon dispatching the `UpdateEvent.INITIALIZED` event, the update process is completed.

When you call the `initialize()` method, the updater starts the update process, and completes all steps, based on the timer delay setting. However, you can also start the update process at any time by calling the `checkNow()` method of the updater object:

```javascript
appUpdater.checkNow();
```

This method does nothing if the update process is already running. Otherwise, it starts the update process.

The updater object can dispatch the following event as a result of the calling the `checkNow()` method:

```javascript
UpdateEvent.CHECK_FOR_UPDATE event just before it attempts to download the update descriptor file.
```

If you cancel the `checkForUpdate` event, you can call the `checkForUpdate()` method of the updater object. (See the next section.) If you do not cancel the event, the update process proceeds to check for the update descriptor file.

Managing the update process in the ApplicationUpdaterUI version

In the ApplicationUpdaterUI version, the user can cancel the process via Cancel buttons in the dialog boxes of the user interface. Also, you can programmatically cancel the update process by calling the `cancelUpdate()` method of the ApplicationUpdaterUI object.

You can set properties of the ApplicationUpdaterUI object or define elements in the update configuration file to specify which dialog box confirmations the updater displays. For details, see “Configuring the update settings” on page 384.

Managing the update process in the ApplicationUpdater version

You can call the `preventDefault()` method of event objects dispatched by the ApplicationUpdater object to cancel steps of the update process (see “The update process” on page 386). Canceling the default behavior gives your application a chance to display a message to the user asking if they want to proceed.

The following sections describe how to continue the update process when a step of the process has been canceled.
**Downloading and interpreting the update descriptor file**

The ApplicationUpdater object dispatches the `checkForUpdate` event before the update process begins, just before the updater tries to download the update descriptor file. If you cancel the default behavior of the `checkForUpdate` event, the updater does not download the update descriptor file. You can call the `checkForUpdate()` method to resume the update process:

```javascript
appUpdater.checkForUpdate();
```

Calling the `checkForUpdate()` method causes the updater to asynchronously download and interpret the update descriptor file. As a result of calling the `checkForUpdate()` method, the updater object can dispatch the following events:

- **StatusUpdateEvent.UPDATE_STATUS**—The updater has downloaded and interpreted the update descriptor file successfully. This event has these properties:
  - `available`—A Boolean value. Set to `true` if there is a different version available than that of the current application; `false` otherwise (the version is the same).
  - `version`—A String. The version from the application descriptor file of the update file
  - `details`—An Array. If there are no localized versions of the description, this array returns an empty string (“”) as the first element and the description as the second element.

If there are multiple versions of the description (in the update descriptor file), the array contains multiple sub-arrays. Each array has two elements: the first is a language code (such as “en”), and the second is the corresponding description (a String) for that language. See “Defining the update descriptor file and adding the AIR file to your web server” on page 383.

- **StatusUpdateErrorEvent.UPDATE_ERROR**—There was an error, and the updater could not download or interpret the update descriptor file.

**Downloading the update AIR file**

The ApplicationUpdater object dispatches the `updateStatus` event after the updater successfully downloads and interprets the update descriptor file. The default behavior is to start downloading the update file if it is available. If you cancel the default behavior, you can call the `downloadUpdate()` method to resume the update process:

```javascript
appUpdater.downloadUpdate();
```

Calling this method causes the updater to asynchronously download update version of the AIR file.

The `downloadUpdate()` method can dispatch the following events:

- **UpdateEvent.DOWNLOAD_START**—The connection to the server was established. When using ApplicationUpdaterUI library, this event displays a dialog box with a progress bar to track the download progress.
- **ProgressEvent.PROGRESS**—Dispatched periodically as file download progresses.
- **DownloadErrorEvent.DOWNLOAD_ERROR**—Dispatched if there is an error while connecting or downloading the update file. It is also dispatched for invalid HTTP statuses (such as “404 - File not found”). This event has an `errorID` property, an integer defining additional error information. An additional `subErrorID` property may contain more error information.
- **UpdateEvent.DOWNLOAD_COMPLETE**—The updater has downloaded and interpreted the update descriptor file successfully. If you do not cancel this event, the ApplicationUpdater version proceeds to install the update version. In the ApplicationUpdaterUI version, the user is presented with a dialog box that gives them the option to proceed.
Updating the application
The ApplicationUpdater object dispatches the `downloadComplete` event when the download of the update file is complete. If you cancel the default behavior, you can call the `installUpdate()` method to resume the update process:

```javascript
appUpdater.installUpdate(file);
```

Calling this method causes the updater install an update version of the AIR file. The method includes one parameter, `file`, which is a File object referencing the AIR file to use as the update.

The ApplicationUpdater object can dispatch the `beforeInstall` event as a result of calling the `installUpdate()` method:

- `UpdateEvent.BEFORE_INSTALL`—Dispatched just before installing the update. Sometimes, it is useful to prevent the installation of the update at this time, so that the user can complete current work before the update proceeds. Calling the `preventDefault()` method of the Event object postpones the installation until the next restart and no additional update process can be started. (These include updates that would result by calling the `checkNow()` method or because of the periodical check.)

Installing from an arbitrary AIR file
You can call the `installFromAIRFile()` method to install the update version to install from an AIR file on the user's computer:

```javascript
appUpdater.installFromAIRFile();
```

This method causes the updater to install an update version the application from the AIR file.

The `installFromAIRFile()` method can dispatch the following events:

- `StatusFileUpdateEvent.FILE_UPDATE_STATUS`—Dispatched after the ApplicationUpdater successfully validated the file sent using the `installFromAIRFile()` method. This event has the following properties:
  - `available`—Set to `true` if there is a different version available than one of the current application; `false` otherwise (the versions are the same).
  - `version`—The string representing the new available version.
  - `path`—Represents the native path of the update file.

You can cancel this event if the available property of the StatusFileUpdateEvent object is set to `true`. Canceling the event cancels the update from proceeding. Call the `installUpdate()` method to continue the canceled update.

- `StatusFileUpdateErrorEvent.FILE_UPDATE_ERROR`—There was an error, and the updater could not install the AIR application.

Canceling the update process
You can call the `cancelUpdate()` method to cancel the update process:

```javascript
appUpdater.cancelUpdate();
```

This method cancels any pending downloads, deleting any incomplete downloaded files, and restarts the periodical check timer.

The method does nothing if the updater object is initializing.
Localizing the ApplicationUpdaterUI interface

The ApplicationUpdaterUI class provides a default user interface for the update process. This includes dialog boxes that let the user start the process, cancel the process, and perform other related actions.

The description element of the update descriptor file lets you define the description of the application in multiple languages. Use multiple text elements that define lang attributes, as in the following:

```xml
<?xml version="1.0" encoding="utf-8"?>
<update xmlns="http://ns.adobe.com/air/framework/update/description/1.0">
  <version>1.1a1</version>
  <url>http://example.com/updates/sample_1.1a1.air</url>
  <description>
    <text xml:lang="en">English description</text>
    <text xml:lang="fr">French description</text>
    <text xml:lang="ro">Romanian description</text>
  </description>
</update>
```

The update framework uses the description that best fits the end user’s localization chain. For more information, see Defining the update descriptor file and adding the AIR file to your web server.

Flex developers can directly add a new language to the "ApplicationUpdaterDialogs" bundle.

JavaScript developers can call the addResources() method of the updater object. This method dynamically adds a new resource bundle for a language. The resource bundle defines localized strings for a language. These strings are used in various dialog box text fields.

JavaScript developers can use the localeChain property of the ApplicationUpdaterUI class to define the locale chain used by the user interface. Typically, only JavaScript (HTML) developers use this property. Flex developers can use the ResourceManager to manage the locale chain.

For example, the following JavaScript code defines resource bundles for Romanian and Hungarian:

```javascript
appUpdater.addResources("ro_RO",
    {titleCheck: "Titlu", msgCheck: "Mesaj", btnCheck: "Buton"});
appUpdater.addResources("hu", {titleCheck: "Cím", msgCheck: "Üzenet"});
var languages = ['ro', 'hu'];
languages = languages.concat(air.Capabilities.languages);
var sortedLanguages = air.Localizer.sortLanguagesByPreference(languages,
    air.Capabilities.language,
    "en-US");
sortedLanguages.push("en-US");
appUpdater.localeChain = sortedLanguages;
```

For details, see the description of the addResources() method of the ApplicationUpdaterUI class in the language reference.
Chapter 38: Viewing Source Code

Just as a user can view source code for an HTML page in a web browser, users can view the source code of an HTML-based AIR application. The Adobe® AIR® SDK includes an AIRSourceViewer.js JavaScript file that you can use in your application to easily reveal source code to end users.

Loading, configuring, and opening the Source Viewer

The Source Viewer code is included in a JavaScript file, AIRSourceViewer.js, that is included in the frameworks directory of the AIR SDK. To use the Source Viewer in your application, copy the AIRSourceViewer.js to your application project directory and load the file via a script tag in the main HTML file in your application:

```html
<script type="text/javascript" src="AIRSourceViewer.js"></script>
```

The AIRSourceViewer.js file defines a class, SourceViewer, which you can access from JavaScript code by calling `air.SourceViewer`.

The SourceViewer class defines three methods: `getDefault()`, `setup()`, and `viewSource()`.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getDefault()</code></td>
<td>A static method. Returns a SourceViewer instance, which you can use to call the other methods.</td>
</tr>
<tr>
<td><code>setup()</code></td>
<td>Applies configuration settings to the Source Viewer. For details, see “Configuring the Source Viewer” on page 391</td>
</tr>
<tr>
<td><code>viewSource()</code></td>
<td>Opens a new window in which the user can browse and open source files of the host application.</td>
</tr>
</tbody>
</table>

**Note:** Code using the Source Viewer must be in the application security sandbox (in a file in the application directory).

For example, the following JavaScript code instantiates a Source Viewer object and opens the Source Viewer window listing all source files:

```javascript
var viewer = air.SourceViewer.getDefault();
viewer.viewSource();
```

Configuring the Source Viewer

The `config()` method applies given settings to the Source Viewer. This method takes one parameter: `configObject`. The `configObject` object has properties that define configuration settings for the Source Viewer. The properties are `default`, `exclude`, `initialPosition`, `modal`, `typesToRemove`, and `typesToAdd`.

**default**

A string specifying the relative path to the initial file to be displayed in the Source Viewer.

For example, the following JavaScript code opens the Source Viewer window with the index.html file as the initial file shown:

```javascript
var viewer = air.SourceViewer.getDefault();
var configObj = {};
configObj.default = "index.html";
viewer.viewSource(configObj);
```
**exclude**
An array of strings specifying files or directories to be excluded from the Source Viewer listing. The paths are relative to the application directory. Wildcard characters are not supported.

For example, the following JavaScript code opens the Source Viewer window listing all source files except for the AIRSourceViewer.js file, and files in the Images and Sounds subdirectories:

```javascript
var viewer = air.SourceViewer.getDefault();
var configObj = {};
configObj.exclude = ["AIRSourceViewer.js", "Images", "Sounds"];
viewer.viewSource(configObj);
```

**initialPosition**
An array that includes two numbers, specifying the initial x and y coordinates of the Source Viewer window.

For example, the following JavaScript code opens the Source Viewer window at the screen coordinates [40, 60] (X = 40, Y = 60):

```javascript
var viewer = air.SourceViewer.getDefault();
var configObj = {};
configObj.initialPosition = [40, 60];
viewer.viewSource(configObj);
```

**modal**
A Boolean value, specifying whether the Source Viewer should be a modal (true) or non-modal (false) window. By default, the Source Viewer window is modal.

For example, the following JavaScript code opens the Source Viewer window such that the user can interact with both the Source Viewer window and any application windows:

```javascript
var viewer = air.SourceViewer.getDefault();
var configObj = {};
configObj.modal = false;
viewer.viewSource(configObj);
```

**typesToAdd**
An array of strings specifying the file types to include in the Source Viewer listing, in addition to the default types included.

By default, the Source Viewer lists the following file types:

- Text files—TXT, XML, MXML, HTM, HTML, JS, AS, CSS, INI, BAT, PROPERTIES, CONFIG
- Image files—JPG, JPEG, PNG, GIF

If no value is specified, all default types are included (except for those specified in the typesToExclude property).

For example, the following JavaScript code opens the Source Viewer window include VCF and VCARD files:

```javascript
var viewer = air.SourceViewer.getDefault();
var configObj = {};
configObj.typesToAdd = ["text.vcf", "text.vcard"];
viewer.viewSource(configObj);
```

For each file type you list, you must specify “text” (for text file types) or “image” (for image file types).
**typesToExclude**
An array of strings specifying the file types to exclude from the Source Viewer.

By default, the Source Viewer lists the following file types:

- Text files—TXT, XML, MXML, HTM, HTML, JS, AS, CSS, INI, BAT, PROPERTIES, CONFIG
- Image files—JPG, JPEG, PNG, GIF

For example, the following JavaScript code opens the Source Viewer window without listing GIF or XML files:

```javascript
var viewer = air.SourceViewer.getDefault();
var configObj = {};
configObj.typesToExclude = ["image.gif", "text.xml"]; 
viewer.viewSource(configObj);
```

For each file type you list, you must specify "text" (for text file types) or "image" (for image file types).

**Opening the Source Viewer**
You should include a user interface element, such as a link, button, or menu command, that calls the Source Viewer code when the user selects it. For example, the following simple application opens the Source Viewer when the user clicks a link:

```html
<html>
<head>
    <title>Source Viewer Sample</title>
    <script type="text/javascript" src="AIRSourceViewer.js"></script>
    <script type="text/javascript">
        function showSources(){
            var viewer = air.SourceViewer.getDefault();
            viewer.viewSource();
        }
    </script>
</head>
<body>
    <p>Click to view the source files.</p>
    <input type="button" onclick="showSources()" value="View Source" />
</body>
</html>
```
Source Viewer user interface

When the application calls the `viewSource()` method of a `SourceViewer` object, the AIR application opens a Source Viewer window. The window includes a list of source files and directories (on the left) and a display area showing the source code for the selected file (on the right):

Directories are listed in brackets. The user can click a brace to expand or collapse the listing of a directory.

The Source Viewer can display the source for text files with recognized extensions (such as HTML, HTML, JS, TXT, XML, and others) or for image files with recognized image extensions (JPG, JPEG, PNG, and GIF). If the user selects a file that does not have a recognized file extension, an error message is displayed ("Cannot retrieve text content from this filetype").

Any source files that are excluded via the `setup()` method are not listed (see "Loading, configuring, and opening the Source Viewer" on page 391).
Chapter 39: Localizing AIR applications

Adobe® AIR® includes support for multiple languages (starting with version 1.1).

Introduction to localization

Localization is the process of including assets to support multiple locales. A locale is the combination of a language and a country code. For example, en_US refers to the English language as spoken in the United States, and fr_FR refers to the French language as spoken in France. To localize an application for these locales, you would provide two sets of assets: one for the en_US locale and one for the fr_FR locale.

Locales can share languages. For example, en_US and en_GB (Great Britain) are different locales. In this case, both locales use the English language, but the country code indicates that they are different locales, and might therefore use different assets. For example, an application in the en_US locale might spell the word “color”, whereas the word would be “colour” in the en_GB locale. Also, units of currency would be represented in dollars or pounds, depending on the locale, and the format of dates and times might also be different.

You can also provide a set of assets for a language without specifying a country code. For example, you can provide en assets for the English language and provide additional assets for the en_US locale, specific to U.S. English.

The AIR SDK provides an HTML Localization Framework (contained in an AIRLocalizer.js file). This framework includes APIs that assist in working with multiple locales. For details see “Localizing HTML content” on page 396.

Localization goes beyond just translating strings used in your application. It can also include any type of asset such as audio files, images, and videos.

Localizing the application name and description in the application installer

You can specify multiple languages for the name and description elements in the application descriptor file. For example, the following specifies the application name in three languages (English, French, and German):

```xml
<name>
   <text xml:lang="en">Sample 1.0</text>
   <text xml:lang="fr">Échantillon 1.0</text>
   <text xml:lang="de">Stichprobe 1.0</text>
</name>
```

The xml:lang attribute for each text element specifies a language code, as defined in RFC4646 (http://www.ietf.org/rfc/rfc4646.txt).

The name element defines the application name that the AIR application installer displays. The AIR application installer uses the localized value that best matches the user interface languages defined by the operating system settings.

You can similarly specify multiple language versions of the description element in the application descriptor file. This element defines description text that the AIR application installer displays.
These settings only apply to the languages available in the AIR application installer. They do not define the locales available for the running, installed application. AIR applications can provide user interfaces that support multiple languages, including and in addition to those available to the AIR application installer.

For more information, see “Defining properties in the application descriptor file” on page 117.

Choosing a locale

To determine which locale your application uses, you can use one of the following methods:

- User prompt — You can start the application in some default locale, and then ask the user to choose their preferred locale.
- `Capabilities.languages` — The `Capabilities.languages` property lists an array of languages available on the user’s preferred languages, as set through the operating system. The strings contain language tags (and script and region information, where applicable) defined by RFC4646 (http://www.ietf.org/rfc/rfc4646.txt). The strings use hyphens as a delimiter (for example, "en-US" or "ja-JP"). The first entry in the returned array has the same primary language ID as the language property. For example, if `languages[0]` is set to "en-US", then the language property is set to "en". However, if the language property is set to "xu" (specifying an unknown language), the first element in the `languages` array will be different.
- `Capabilities.language` — The `Capabilities.language` property provides the user interface language code of the operating system. However, this property is limited to 20 known languages. And on English systems, this property returns only the language code, not the country code. For these reasons, it is better to use the first element in the `Capabilities.languages` array.

Localizing HTML content

The AIR 1.1 SDK includes an HTML localization framework. The AIRLocalizer.js JavaScript file defines the framework. The frameworks directory of the AIR SDK contains the AIRLocalizer.js file. This file includes an `air.Localizer` class, which provides functionality to assist in creating applications that support multiple localized versions.

Loading the AIR HTML localization framework code

To use the localization framework, copy the AIRLocalizer.js file to your project. Then include it in the main HTML file of the application, using a script tag:

```html
<script src="AIRLocalizer.js" type="text/javascript" charset="utf-8"></script>
```

Subsequent JavaScript can call the `air.Localizer.localizer` object:

```javascript
<script>
   var localizer = air.Localizer.localizer;
</script>
```

The `air.Localizer.localizer` object is a singleton object that defines methods and properties for using and managing localized resources. The Localizer class includes the following methods:
Localizing AIR applications

The Localizer class includes the following static properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>localizer</td>
<td>Returns a reference to the singleton Localizer object for the application.</td>
</tr>
<tr>
<td>ultimateFallbackLocale</td>
<td>The locale used when the application supports no user preference. See “Defining the locale chain” on page 401.</td>
</tr>
</tbody>
</table>

Defining resource bundles

The HTML localization framework reads localized versions of strings from localization files. A localization file is a collection of key-based values, serialized in a text file. A localization file is sometimes referred to as a bundle.

Create a subdirectory of your application project directory, named locale. (You can also use a different name, see “Customizing AIR HTML Localizer settings” on page 400.) This directory will include the localization files. This directory is known as the bundles directory.

For each locale that your application supports, create a subdirectory of the bundles directory. Name each subdirectory to match the locale code. For example, name the French directory “fr” and name the English directory “en.” You can use an underscore (_) character to define a locale that has a language and country code. For example, name the U.S. English directory “en_us.” (Alternately, you can use a hyphen instead of an underscore, as in “en-us.” The HTML localization framework supports both.)

You can add any number of resource files to a locale subdirectory. Generally, you create a localization file for each language (and place the file in the directory for that language). The HTML localization framework includes a getFile() method that allows you to read the contents of a file (see “Getting resources for a specific locale” on page 402.

Files that have the .properties file extension are known as localization properties files. You can use them to define key-value pairs for a locale. A properties file defines one string value on each line. For example, the following defines a string value “Hello in English.” for a key named greeting:

greeting=Hello in English.
A properties file containing the following text defines six key-value pairs:

```
title=Sample Application
greeting=Hello in English.
extMessage=Thank you for using the application.
color1=Red
color2=Green
color3=Blue
```

This example shows an English version of the properties file, to be stored in the en directory.

A French version of this properties file is placed in the fr directory:

```
title=Application Example
greeting=Bonjour en français.
extMessage=Merci d’avoir utilisé cette application.
color1=Rouge
color2=Vert
color3=Bleu
```

You may define multiple resource files for different kinds of information. For example, a legal.properties file may contain boilerplate legal text (such as copyright information). You may want to reuse that resource in multiple applications. Similarly, you might define separate files that define localized content for different parts of the user interface.

Use UTF-8 encoding for these files, to support multiple languages.

### Managing locale chains

When your application loads the AIRLocalizer.js file, it examines the locales defined in your application. These locales correspond to the subdirectories of the bundles directory (see “Defining resource bundles” on page 397). This list of available locales is known as the locale chain. The AIRLocalizer.js file automatically sorts the locale chain based on the preferred order defined by the operating system settings. (The Capabilities.languages property lists the operating system user interface languages, in preferred order.)

So, if an application defines resources for "en", "en_US" and "en_UK" locales, the AIR HTML Localizer framework sorts the locale chain appropriately. When an application starts on a system that reports "en" as the primary locale, the locale chain is sorted as ["en", "en_US", "en_UK"]. In this case, the application looks for resources in the "en" bundle first, then in the "en_US" bundle.

However, if the system reports "en-US" as the primary locale, then the sorting uses ["en_US", "en", "en_UK"]. In this case, the application looks for resources in the "en_US" bundle first, then in the "en" bundle.

By default, the application defines the first locale in the locale chain as the default locale to use. You may ask the user to select a locale upon first running the application. You may then choose to store the selection in a preferences file and use that locale in subsequent start-up of the application.

Your application can use resource strings in any locale in the locale chain. If a specific locale does not define a resource string, the application uses the next matching resource string for other locales defined in the locale chain.

You can customize the locale chain by calling the setLocaleChain() method of the Localizer object. See “Defining the locale chain” on page 401.
Updating the DOM elements with localized content

An element in the application can reference a key value in a localization properties file. For example, the title element in the following example specifies a local_innerHTML attribute. The localization framework uses this attribute to look up a localized value. By default, the framework looks for attribute names that start with "local_." The framework updates the attributes that have names that match the text following "local_." In this case, the framework sets the innerHTML attribute of the title element. The innerHTML attribute uses the value defined for the mainWindowTitle key in the default properties file (default.properties):

```
<title local_innerHTML="default.mainWindowTitle"/>
```

If the current locale defines no matching value, then the localizer framework searches the rest of the locale chain. It uses the next locale in the locale chain for which a value is defined.

In the following example, the text (innerHTML attribute) of the p element uses the value of the greeting key defined in the default properties file:

```
<p local_innerHTML="default.greeting" />
```

In the following example, the value attribute (and displayed text) of the input element uses the value of the btnBlue key defined in the default properties file:

```
<input type="button" local_value="default.btnBlue" />
```

To update the HTML DOM to use the strings defined in the current locale chain, call the update() method of the Localizer object. Calling the update() method causes the Localizer object to parse the DOM and apply manipulations where it finds localization ("local_...") attributes:

```
air.Localizer.localizer.update();
```

You can define values for both an attribute (such as "innerHTML") and its corresponding localization attribute (such as "local_innerHTML"). In this case, the localization framework only overwrites the attribute value if it finds a matching value in the localization chain. For example, the following element defines both value and local_value attributes:

```
<input type="text" value="Blue" local_value="default.btnBlue"/>
```

You can also update a specific DOM element only. See the next section, “Updating DOM elements to use the current locale” on page 399.

By default, the AIR HTML Localizer uses "local_" as the prefix for attributes defining localization settings for an element. For example, by default a local_innerHTML attribute defines the bundle and resource name used for the innerHTML value of an element. Also, by default a local_value attribute defines the bundle and resource name used for the value attribute of an element. You can configure the Localizer to use an attribute prefix other than "local_". See “Customizing AIR HTML Localizer settings” on page 400.

Updating DOM elements to use the current locale

When the Localizer object updates the HTML DOM, it causes marked elements to use attribute values based on strings defined in the current locale chain. To have the HTML localizer update the HTML DOM, call the update() method of the Localizer object:

```
air.Localizer.localizer.update();
```

To update only a specified DOM element, pass it as a parameter to the update() method. The update() method has only one parameter, parentNode, which is optional. When specified, the parentNode parameter defines the DOM element to localize. Calling the update() method and specifying a parentNode parameter sets localized values for all child elements that specify localization attributes.
For example, consider the following div element:

```html
<div id="colorsDiv">
  <h1 local_innerHTML="default.lblColors" ></h1>
  <p><input type="button" local_value="default.btnBlue" /></p>
  <p><input type="button" local_value="default.btnRed" /></p>
  <p><input type="button" local_value="default.btnGreen" /></p>
</div>
```

To update this element to use localized strings defined in the current locale chain, use the following JavaScript code:

```javascript
var divElement = window.document.getElementById("colorsDiv");
air.Localizer.localizer.update(divElement);
```

If a key value is not found in the locale chain, the localization framework sets the attribute value to the value of the "local_" attribute. For example, in the previous example, suppose the localization framework cannot find a value for the lblColors key (in any of the default.properties files in the locale chain). In this case, it uses "default.lblColors" as the innerHTML value. Using this value indicates (to the developer) missing resources.

The `update()` method dispatches a `resourceNotFound` event when it cannot find a resource in the locale chain. The `air.Localizer RESOURCE NOT FOUND` constant defines the string "resourceNotFound". The event has three properties: `bundleName`, `resourceName`, and `locale`. The `bundleName` property is the name of the bundle in which the resource is not found. The `resourceName` property is the name of the bundle in which the resource is not found. The `locale` property is the name of the locale in which the resource is not found.

The `update()` method dispatches a `bundleNotFound` event when it cannot find the specified bundle. The `air.Localizer BUNDLE NOT FOUND` constant defines the string "bundleNotFound". The event has two properties: `bundleName` and `locale`. The `bundleName` property is the name of the bundle in which the resource is not found. The `locale` property is the name of the locale in which the resource is not found.

The `update()` method operates asynchronously (and dispatches `resourceNotFound` and `bundleNotFound` events asynchronously). The following code sets event listeners for the `resourceNotFound` and `bundleNotFound` events:

```javascript
air.Localizer.localizer.addEventListener(air.Localizer RESOURCE NOT FOUND, rnfHandler);
air.Localizer.localizer.addEventListener(air.Localizer BUNDLE NOT FOUND, rnfHandler);
air.Localizer.localizer.update();
function rnfHandler(event)
{
  alert(event.bundleName + "::" + event.resourceName + ":." + event.locale);
}
function bnfHandler(event)
{
  alert(event.bundleName + ":." + event.locale);
}
```

**Customizing AIR HTML Localizer settings**

The `setBundlesDirectory()` method of the Localizer object lets you customize the bundles directory path. The `setLocalAttributePrefix()` method of the Localizer object lets you customize the bundles directory path and customize the attribute value used by the Localizer.

The default bundles directory is defined as the locale subdirectory of the application directory. You can specify another directory by calling the `setBundlesDirectory()` method of the Localizer object. This method takes one parameter, `path`, which is the path to the desired bundles directory, as a string. The value of the `path` parameter can be any of the following:

- A String defining a path relative to the application directory, such as "locales"
Localizing AIR applications

A String defining a valid URL that uses the app, app-storage, or file URL schemes, such as "app://languages" (do not use the http URL scheme)

A File object

For information on URLs and directory paths, see "Paths of File objects" on page 187.

For example, the following code sets the bundles directory to a languages subdirectory of the application storage directory (not the application directory):

```javascript
air.Localizer.localizer.setBundlesDirectory("languages");
```

Pass a valid path as the path parameter. Otherwise, the method throws a BundlePathNotFoundError exception. This error has "BundlePathNotFoundError" as its name property, and its message property specifies the invalid path.

By default, the AIR HTML Localizer uses "local_" as the prefix for attributes defining localization settings for an element. For example, the local_innerHTML attribute defines the bundle and resource name used for the innerHTML value of the following input element:

```html
<p local_innerHTML="default.greeting" />
```

The setLocalAttributePrefix() method of the Localizer object lets you use an attribute prefix other than "local_". This static method takes one parameter, which is the string you want to use as the attribute prefix. For example, the following code sets the localization framework to use "loc_" as the attribute prefix:

```javascript
air.Localizer.localizer.setLocalAttributePrefix("loc_");
```

You can customize the attribute prefix the localization framework uses. You may want to customize the prefix if the default value ("local_") conflicts with the name of another attribute used by your code. Be sure to use valid characters for HTML attributes when calling this method. (For example, the value cannot contain a blank space character.)

For more information on using localization attributes in HTML elements, see "Updating the DOM elements with localized content" on page 399.

The bundles directory and attribute prefix settings do not persist between different application sessions. If you use a custom bundles directory or attribute prefix setting, be sure to set it each time the application initiates.

### Defining the locale chain

By default, when you load the AIRLocalizer.js code, it sets the default locale chain. The locales available in the bundles directory and the operating system language settings define this locale chain. (For details, see “Managing locale chains” on page 398.)

You can modify the locale chain by calling the static setLocaleChain() method of the Localizer object. For example, you may want to call this method if the user indicates a preference for a specific language. The setLocaleChain() method takes one parameter, chain, which is an array of locales, such as ["fr_FR", "fr", "fr_CA"]). The order of the locales in the array sets the order in which the framework looks for resources (in subsequent operations). If a resource is not found for the first locale in the chain, it continues looking in the other locale’s resources. If the chain argument is missing, is not an array, or is an empty array, the function fails and throws an IllegalArgumentsError exception.

The static getLocaleChain() method of the Localizer object returns an Array listing the locales in the current locale chain.

The following code reads the current locale chain and adds two French locales to the head of the chain:

```javascript
var currentChain = air.Localizer.localizer.getLocaleChain();
newLocales = ["fr_FR", "fr"];
air.Localizer.localizer.setLocaleChain(newLocales.concat(currentChain));
```
The `setLocaleChain()` method dispatches a "change" event when it updates the locale chain. The `air.Localizer.LOCALE_CHANGE` constant defines the string "change". The event has one property, `localeChain`, an array of locale codes in the new locale chain. The following code sets an event listener for this event:

```javascript
var currentChain = air.Localizer.localizer.getLocaleChain();
newLocales = ["fr_FR", "fr"];
localizer.addEventListener(air.Localizer.LOCALE_CHANGE, changeHandler);
air.Localizer.localizer.setLocaleChain(newLocales.concat(currentChain));
function changeHandler(event)
{
    alert(event.localeChain);
}
```

The static `air.Localizer.ultimateFallbackLocale` property represents the locale used when the application supports no user preference. The default value is "en". You can set it to another locale, as shown in the following code:

```javascript
air.Localizer.ultimateFallbackLocale = "fr";
```

### Getting resources for a specific locale

The `getString()` method of the `Localizer` object returns the string defined for a resource in a specific locale. You do not need to specify a locale value when calling the method. In this case the method looks at the entire locale chain and returns the string in the first locale that provides the given resource name. The method has the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bundleName</td>
<td>The bundle that contains the resource. This is the filename of the properties file without the .properties extension. (For example, if this parameter is set as &quot;alerts&quot;, the Localizer code looks in localization files named alerts.properties.)</td>
</tr>
<tr>
<td>resourceName</td>
<td>The resource name.</td>
</tr>
<tr>
<td>templateArgs</td>
<td>Optional. An array of strings to replace numbered tags in the replacement string. For example, consider a call to the function where the templateArgs parameter is [&quot;Raúl&quot;, &quot;4&quot;] and the matching resource string is &quot;Hello, {0}. You have {1} new messages.&quot;. In this case, the function returns &quot;Hello, Raúl. You have 4 new messages.&quot;. To ignore this setting, pass a null value.</td>
</tr>
<tr>
<td>locale</td>
<td>Optional. The locale code (such as &quot;en&quot;, &quot;en_us&quot;, or &quot;fr&quot;) to use. If a locale is provided and no matching value is found, the method does not continue searching for values in other locales in the locale chain. If no locale code is specified, the function returns the string in the first locale in the locale chain that provides a value for the given resource name.</td>
</tr>
</tbody>
</table>

The localization framework can update marked HTML DOM attributes. However, you can use localized strings in other ways. For example, you can use a string in some dynamically generated HTML or as a parameter value in a function call. For example, the following code calls the `alert()` function with the string defined in the `error114` resource in the default properties file of the `fr_FR` locale:

```javascript
alert(air.Localizer.localizer.getString("default", "error114", null, "fr_FR"));
```

The `getString()` method dispatches a `resourceNotFound` event when it cannot find the resource in the specified bundle. The `air.Localizer.RESOURCE_NOT_FOUND` constant defines the string "resourceNotFound". The event has three properties: `bundleName`, `resourceName`, and `locale`. The `bundleName` property is the name of the bundle in which the resource is not found. The `resourceName` property is the name of the bundle in which the resource is not found. The `locale` property is the name of the locale in which the resource is not found.
The `getString()` method dispatches a `bundleNotFound` event when it cannot find the specified bundle. The `air.Localizer.BUNDLE_NOT_FOUND` constant defines the string "bundleNotFound". The event has two properties: `bundleName` and `locale`. The `bundleName` property is the name of the bundle in which the resource is not found. The `locale` property is the name of the locale in which the resource is not found.

The `getString()` method operates asynchronously (and dispatches the `resourceNotFound` and the `bundleNotFound` events asynchronously). The following code sets event listeners for the `resourceNotFound` and `bundleNotFound` events:

```javascript
air.Localizer.localizer.addEventListener(air.Localizer.RESOURCE_NOT_FOUND, rnfHandler);
air.Localizer.localizer.addEventListener(air.Localizer.BUNDLE_NOT_FOUND, bnfHandler);
var str = air.Localizer.localizer.getString("default", "error114", null, "fr_FR");
function rnfHandler(event)
{
    alert(event.bundleName + " : " + event.resourceName + " :." + event.locale);
}
function bnfHandler(event)
{
    alert(event.bundleName + " :." + event.locale);
}
```

The `getResourceBundle()` method of the `Localizer` object returns a specified bundle for a given locale. The return value of the method is an object with properties matching the keys in the bundle. (If the application cannot find the specified bundle, the method returns `null`.)

The method takes two parameters—`locale` and `bundleName`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>locale</td>
<td>The locale (such as &quot;fr&quot;).</td>
</tr>
<tr>
<td>bundleName</td>
<td>The bundle name.</td>
</tr>
</tbody>
</table>

For example, the following code calls the `document.write()` method to load the default bundle for the fr locale. It then calls the `document.write()` method to write values of the `str1` and `str2` keys in that bundle:

```javascript
var aboutWin = window.open();
var bundle = localizer.getResourceBundle("fr", "default");
aboutWin.document.write(bundle.str1);
aboutWin.document.write("<br/>");
aboutWin.document.write(bundle.str2);
aboutWin.document.write("<br/>");
```

The `getResourceBundle()` method dispatches a `bundleNotFound` event when it cannot find the specified bundle. The `air.Localizer.BUNDLE_NOT_FOUND` constant defines the string "bundleNotFound". The event has two properties: `bundleName` and `locale`. The `bundleName` property is the name of the bundle in which the resource is not found. The `locale` property is the name of the locale in which the resource is not found.

The `getFile()` method of the `Localizer` object returns the contents of a bundle, as a string, for a given locale. The bundle file is read as a UTF-8 file. The method includes the following parameters:
Localizing AIR applications

For example, the following code calls the `document.write()` method using the contents of the about.html file of the fr locale:

```javascript
var aboutWin = window.open();
var aboutHtml = localizer.getFile("about.html", null, "fr");
aboutWin.document.close();
aboutWin.document.write(aboutHtml);
```

The `getFile()` method dispatches a `fileNotFound` event when it cannot find a resource in the locale chain. The `air.Localizer.FILE_NOT_FOUND` constant defines the string "resourceNotFound". The `getFile()` method operates asynchronously (and dispatches the `fileNotFound` event asynchronously). The event has two properties: `fileName` and `locale`. The `fileName` property is the name of the file not found. The `locale` property is the name of the locale in which the resource is not found. The following code sets an event listener for this event:

```javascript
air.Localizer.localizer.addEventListener(air.Localizer.FILE_NOT_FOUND, fnfHandler);
air.Localizer.localizer.getFile("missing.html", null, "fr");
function fnfHandler(event)
{
    alert(event.fileName + ": " + event.locale);
}
```

Localizing dates, times, and currencies

The way applications present dates, times, and currencies varies greatly for each locale. For example, the U.S. standard for representing dates is month/day/year, whereas the European standard for representing dates is day/month/year.

You can write code to format dates, times, and currencies. For example, the following code converts a Date object into month/day/year format or day/month/year format. If the `locale` variable (representing the locale) is set to "en_US", the function returns month/day/year format. The example converts a Date object into day/month/year format for all other locales:
function convertDate(date)
{
    if (locale == "en_US")
    {
        return (date.getMonth() + 1) + "/" + date.getDate() + "/" + date.getFullYear();
    }
    else
    {
        return date.getDate() + "/" + (date.getMonth() + 1) + "/" + date.getFullYear();
    }
}

Some Ajax frameworks provide support for localizing dates and numbers.
Chapter 40: XML signature validation

Use the classes in the XMLSignatureValidator API to validate digital signatures conforming to a subset of the W3C recommendation for XML-Signature Syntax and Processing (http://www.w3.org/TR/xmldsig-core/). XML signatures can be used to help verify the integrity and signer identity of data or information.

XML signatures can be used to validate messages or resources downloaded by your application. For example, if your application provides services on a subscription basis, you could encapsulate the subscription terms in a signed XML document. If someone attempted to alter the subscription document, validation would fail.

You could also use an XML signature to help validate downloaded resources used by your application by including a signed manifest containing digests of those resources. Your application could verify that the resources have not been altered by comparing the digest in the signed file with a digest computed from the loaded bytes. This is particularly valuable when the downloaded resource is a SWF file or other active content that you want to run in the application security sandbox.

Additional online information about XML signature validation

You can find more information about the XMLSignatureValidator API from these sources:

Quick Starts (Adobe AIR Developer Connection)
- Creating and validating XML signatures

Language Reference
- XMLSignatureValidator
- IURIDereferencer
- ReferencesValidationSetting
- RevocationCheckSettings
- SignerTrustSettings
- SignatureStatus

Adobe Developer Connection Articles and Samples
- Adobe AIR Developer Connection for HTML and Ajax (search for `XML signatures`)

Basics of XML signature validation

Adobe® AIR® provides the XMLSignatureValidator class and IURIDereferencer interface for validating XML signatures. The XML syntax accepted by the XMLSignatureValidator class is a subset of the W3C recommendation for XML Signature Syntax and Processing. (Because only a subset of the recommendation is supported, not all legal signatures can be validated.) AIR does not provide an API for creating XML signatures.
XML signature validation classes

The XML signature validation API includes the following classes:

<table>
<thead>
<tr>
<th>Package</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>flash.security</td>
<td>• XMLSignatureValidator</td>
</tr>
<tr>
<td></td>
<td>• IURIDereferencer (interface)</td>
</tr>
<tr>
<td></td>
<td>XMLSignatureValidator string constants are defined in the following classes:</td>
</tr>
<tr>
<td></td>
<td>• ReferencesValidationSetting</td>
</tr>
<tr>
<td></td>
<td>• RevocationCheckSettings</td>
</tr>
<tr>
<td></td>
<td>• SignatureStatus</td>
</tr>
<tr>
<td></td>
<td>• SignerTrustSettings</td>
</tr>
<tr>
<td>flash.events</td>
<td>• Event</td>
</tr>
<tr>
<td></td>
<td>• ErrorEvent</td>
</tr>
</tbody>
</table>

Using the XML signature validation classes

To use the XMLSignatureValidator class to validate an XML signature, you must:

- Create an XMLSignatureValidator object
- Provide an implementation of the IURIDereferencer interface. The XMLSignatureValidator object calls the IURIDereferencer dereference() method, passing in the URI for each reference in a signature. The dereference() method must resolve the URI and return the referenced data (which could be in the same document as the signature, or could be in an external resource).
- Set the certificate trust, revocation checking, and reference validation settings of the XMLSignatureValidator object as appropriate for your application.
- Add event listeners for the complete and error events.
- Call the verify() method, passing in the signature to validate.
- Handle the complete and error events and interpret the results.

The following example implements a validate() function that verifies the validity of an XML signature. The XMLSignatureValidator properties are set such that the signing certificate must be in the system trust store, or chain to a certificate in the trust store. The example also assumes that a suitable IURIDereferencer class named XMLDereferencer exists.
private function validate(xmlSignature:XML):void
{
    var verifier:XMLSignatureValidator = new XMLSignatureValidator();
    verifier.addEventListener(Event.COMPLETE, verificationComplete);
    verifier.addEventListener(ErrorEvent.ERROR, verificationError);
    try
    {
        verifier.uriDereferencer = new XMLDereferencer();
        verifier.referencesValidationSetting = ReferencesValidationSetting.VALID_IDENTITY;
        verifier.revocationCheckSetting = RevocationCheckSettings.BEST_EFFORT;
        verifier.useSystemTrustStore = true;

        //Verify the signature
        verifier.verify(xmlSignature);
    }
    catch (e:Error)
    {
        trace("Verification error.\n" + e);
    }
}

//Trace verification results
private function verificationComplete(event:Event):void
{
    var signature:XMLSignatureValidator = event.target as XMLSignatureValidator;
    trace("Signature status: " + signature.validityStatus + "\n");
    trace(" Digest status: " + signature.digestStatus + "\n");
    trace(" Identity status: " + signature.identityStatus + "\n");
    trace(" Reference status: " + signature.referencesStatus + "\n");
}

private function verificationError(event:ErrorEvent):void
{
    trace("Verification error.\n" + event.text);
}

The XML signature validation process

When you call the XMLSignatureValidator verify() method, AIR performs the following steps:

- The runtime verifies the cryptographic integrity of the signature using the public key of the signing certificate.
- The runtime establishes the cryptographic integrity, identity, and trustworthiness of the certificate based on the current settings of the XMLSignatureValidator object.

The trust placed in the signing certificate is key to the integrity of the validation process. Signature validation is conducted using a well-defined cryptographic process, but the trustworthiness of the signing certificate is a judgment that cannot be made algorithmically.

In general, you have three ways to decide whether a certificate is trustworthy:

- By relying on certification authorities and the operating system trust store.
- By obtaining, directly from the signer, a copy of the certificate, another certificate that serves as a trust anchor for the certificate, or sufficient information to reliably identify the certificate, such as the public key.
- Asking the end user of your application if they trust the certificate. Such a query is invalid with self-signed certificates since the identifying information in the certificate is inherently unreliable.

- The runtime verifies the cryptographic integrity of the signed data.

The signed data is verified with the help of your IURIDereferencer implementation. For each reference in the signature document, the IURIDereferencer implementation dereference() method is called. The data returned by the dereference() method is used to compute the reference digest. This digest value is compared to the digest recorded in the signature document. If the digests match, then the data has not been altered since it was signed.

One important consideration when relying on the results of validating an XML signature is that only what is signed is secure. For example, consider a signed manifest listing the files in a package. When the XMLSignatureValidator verifies the signature, it only checks whether the manifest itself is unaltered. The data in the files is not signed, so the signature will still validate when files referenced in the manifest are changed or deleted.

**Note:** To verify files in such a manifest, you can compute the digest of the file data (using the same hashing algorithm used in the manifest) and compare the result to the digest stored in the signed manifest. In some cases, you should also check for the presence of additional files.

### Interpreting validation results

The validation results are reported by the status properties of the XMLSignatureValidator object. These properties can be read after the validator object dispatches the complete event. The four status properties include: validityStatus, digestStatus, identityStatus, and referencesStatus.

#### The validityStatus property

The validityStatus property reports the overall validity of the signature. The validityStatus depends on the state of the other three status properties and can have one of the following values:

- **valid** — If digestStatus, identityStatus, and referencesStatus are all valid.
- **invalid** — If one or more of the individual status properties is invalid.
- **unknown** — If one or more of the individual status properties is unknown and no individual status is invalid.

#### The digestStatus property

The digestStatus property reports the results of the cryptographic verification of the message digest. The digestStatus property can have one of the following values:

- **valid** — If the signature document itself is unaltered since signing.
- **invalid** — If the signature document has been altered or is malformed.
- **unknown** — If the verify() method has not completed without error.

#### The identityStatus property

The identityStatus property reports the status of the signing certificate. The value of this property depends on several factors including:

- the cryptographic integrity of the certificate
- whether the certificate is expired or revoked
- whether the certificate is trusted on the current machine
- the state of the XMLSignatureValidator object (such as whether additional certificates have been added for building the trust chain, whether those certificates are trusted, and the values of the useSystemTrustStore and revocationCheckSettings properties)
The `identityStatus` property can have the following values:

- **valid** — To be considered valid, the signing certificate must meet the following conditions:
  - The signing certificate must be unaltered.
  - The signing certificate must not be expired or revoked—except when a valid timestamp is present in the signature. If the signature is timestamped, the certificate will be considered valid as long as it was valid at the time the document was signed. (The certificate used by the timestamp service to sign the timestamp must chain to a trusted root certificate on the user’s computer.)
  - The signing certificate is trusted. A certificate is trusted if the certificate is in the system trust store or chains to another certificate in the system trust store and you set the `useSystemTrustStore` property to true. You can also designate a certificate as trusted using the `addCertificate()` method of the `XMLSignatureValidator` object.
  - The certificate is, in fact, the signing certificate.
- **invalid** — The certificate is expired or revoked—and no timestamp proving validity at the time of signing is present—or the certificate has been altered.
- **unknown** — If the certificate is not invalid, but is not trusted either. Self-signed certificates, for example, will be reported as unknown (unless explicitly trusted). The `identityStatus` is also reported as unknown if the `verify()` method has not completed without error or if the identity has not been checked because the signature digest is invalid.

**The referencesStatus property**

The `referencesStatus` property reports the cryptographic integrity of the references in the `SignedData` element of the signature.

- **valid** — If the computed digest of every reference in the signature matches the corresponding digest recorded in the XML signature. A valid status indicates that the signed data has not been altered.
- **invalid** — If any computed digest does not match the corresponding digest in the signature.
- **unknown** — If the reference digests have not been checked. The references are not checked if the overall signature digest is invalid or the signing certificate is invalid. If the `identityStatus` is unknown, then the references are only checked when the `referencesValidationSetting` is `validOrUnknown`.

**About XML signatures**

An XML signature is a digital signature represented in XML syntax. The data in an XML signature can be used to validate that the signed information has not been altered since signing. In addition, when a signing certificate has been issued by a trusted certification authority, the identity of the signer can be verified through the public key infrastructure.

An XML signature can be applied to any type of digital data (in binary or XML format). XML signatures are typically used for such purposes as:

- checking whether external or downloaded resources have been modified
- verifying that messages come from a known source
- validating application license or subscription privileges
**Supported XML signature syntax**

AIR supports the following elements from the W3C recommendation for XML Signature Syntax and Processing:

- All core signature syntax elements (section 4 of the W3C recommendation document)—except the KeyInfo element is not fully supported
- The KeyInfo element must only contain an X509Data element
- An X509Data element must only contain an X509Certificate element
- The SHA256 digest method
- The RSA-SHA1 (PKCS1) signing algorithm
- The "Canonical XML without comments" canonicalization method and transform
- The enveloped signature transform
- timestamps

The following document illustrates a typical XML signature (most of the cryptographic data has been removed to simplify the example):

```xml
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    <CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
    <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
    <Reference URI="URI_to_signed_data">
        <Transforms>
            <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
        </Transforms>
        <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
        <DigestValue>u0o...Y=</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>Ked...w==</SignatureValue>
  <KeyInfo>
    <X509Data>
      <X509Certificate>i7d...w==</X509Certificate>
    </X509Data>
  </KeyInfo>
</Signature>
```

The key elements of a signature are:

- **SignedInfo** — Contains references to the signed data and the computed digest values at the time of signing. The signed data itself may be included in the same document as the XML signature or may be external.
- **SignatureValue** — Contains a digest of the SignedInfo element encrypted with the signer’s private key.
- **KeyInfo** — Contains the signing certificate, as well as any additional certificates needed to establish the chain of trust. Note that although the KeyInfo element is technically optional, AIR cannot validate the signature if it is not included.

There are three general types of XML signatures:

- **Enveloped** — the signature is inserted inside the XML data that it is signing.
- **Enveloping** — the signed XML data is contained within an Object element within the Signature element.
Detached — the signed data is external to the XML signature. The signed data might be in an external file. Alternately, it might be in the same XML document as the signature, just not a parent or child element of the Signature element.

XML signatures use URIs to reference the signed data. The signing and the validating applications must use the same conventions for resolving these URIs. When using the XMLSignatureValidator class, you must provide an implementation of the IURIDereferencer interface. This implementation is responsible for resolving the URI and returning the signed data as a ByteArray object. The returned ByteArray object is digested using the same algorithm that produced the digest in the signature.

Certificates and trust
A certificate consists of a public key, identifying information, and possibly one or more certificates belonging to the issuing certification authority.

There are two ways to establish trust in a certificate. You can establish trust by obtaining a copy of the certificate directly from the signer, for example on physical media, or through a secure digital transmission such as an SSL transaction. You can also rely on a certification authority to determine whether the signing certificate is trustworthy.

To rely on a certification authority, the signing certificate must be issued by an authority that is trusted on the computer upon which the signature is validated. Most operating system manufacturers place the root certificates of a number of certification authorities into the operating system trust store. Users can also add and remove certificates from the store.

Even if a certificate is issued by a trusted certification authority, you must still decide whether the certificate belongs to someone you trust. In many use cases, this decision is passed along to the end-user. For example, when an AIR application is installed, the AIR installer displays the identifying information from the publisher’s certificate when asking the user to verify whether they want to install the application. In other cases, you might have to compare the public key or other certificate information to a list of acceptable keys. (This list must be secured, perhaps by its own signature, or by storing in the AIR encrypted local store, so that the list itself cannot be tampered with.)

Note: While you can elect to trust the signing certificate without independent verification—such as when a signature is “self-signed”—you do not thereby gain much assurance of anything by verifying the signature. Without knowing who created a signature, the assurance that the signature has not been tampered with, is of little, if any, value. The signature could be a validly signed forgery.

Certificate expiration and revocation
All certificates expire. Certificates can also be revoked by the issuing certification authority if, for example, the private key related to the certificate is compromised or stolen. If a signature is signed with an expired or revoked certificate, then the signature will be reported as invalid unless a timestamp has been included as part of the signature. If a timestamp is present, then the XMLSignatureValidator class will validate the signature as long as the certificate was valid at the time of signing.

A timestamp is a signed digital message from a timestamp service that certifies that the data was signed at a particular time and date. Timestamps are issued by timestamp authorities and signed by the timestamp authority’s own certificate. The timestamp authority certificate embedded in the timestamp must be trusted on the current machine for the timestamp to be considered valid. The XMLSignatureValidator does not provide an API for designating a different certificate to use in validating the timestamp.
Implementing the IURIDereferencer interface

To validate an XML signature, you must provide an implementation of the IURIDereferencer interface. The implementation is responsible for resolving the URIs within the Reference elements of an XML signature document and returning the data so that the digest can be computed. The computed digest is compared with the digest in the signature to determine if the referenced data has been altered since the signature was created.

Note: HTML-based AIR applications must import a SWF library containing an ActionScript implementation in order to validate XML signatures. The IURIDereferencer interface cannot be implemented in JavaScript.

The IURIDereferencer interface has a single method, `dereference(uri:String)`, that must be implemented. The XMLSignatureValidator object calls this method for each reference in the signature. The method must return the data in a ByteArray object.

In most cases, you will also need to add properties or methods that allow your dereferencer object to locate the referenced data. For example, if the signed data is located in the same document as the signature, you could add a member variable that provides a reference to the XML document. The `dereference()` method can then use this variable, along with the URI, to locate the referenced data. Likewise, if the signed data is located in a directory of the local file system, the `dereference()` method might need a property providing the path to that directory in order to resolve the referenced files.

The XMLSignatureValidator relies entirely on the dereferencer for interpreting URI strings. The standard rules for dereferencing URIs are given in the section 4.3.3 of the W3C Recommendation for XML Signature Syntax and Processing.

Dereferencing URIs in enveloped signatures

When an enveloped XML signature is generated, the signature elements are inserted into the signed data. For example, if you signed the following message using an enveloped signature structure:

```xml
<message>
  <data>...</data>
</message>
```

The resulting signed document will look like this:
<message>
  <data>...</data>
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      <CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/RFC-xmldsig-c14n-20010315"/>
      <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
      <Reference URI="">
        <Transforms>
          <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
        </Transforms>
        <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
        <DigestValue>yv6...Z0Y=</DigestValue>
      </Reference>
    </SignedInfo>
    <SignatureValue>cCY...LQ==</SignatureValue>
    <KeyInfo>
      <X509Data>
        <X509Certificate>MII...4e</X509Certificate>
      </X509Data>
    </KeyInfo>
  </Signature>
</message>

Notice that the signature contains a single Reference element with an empty string as its URI. An empty string in this context refers to the root of the document.

Also notice that the transform algorithm specifies that an enveloped signature transform has been applied. When an enveloped signature transform has been applied, the XMLSignatureValidator automatically removes the signature from the document before computing the digest. This means that the dereferencer does not need to remove the Signature element when returning the data.

The following example illustrates a dereferencer for enveloped signatures:

package
{
    import flash.events.ErrorEvent;
    import flash.events.EventDispatcher;
    import flash.security.IURIDereferencer;
    import flash.utils.ByteArray;
    import flash.utils.IDataInput;

    public class EnvelopedDereferencer extends EventDispatcher implements IURIDereferencer
    {
        private var signedMessage:XML;

        public function EnvelopedDereferencer( signedMessage:XML )
        {
            this.signedMessage = signedMessage;
        }

        public function dereference( uri:String ):IDataInput
        {
            try
            {
                return null;
            }
            catch (e:ErrorEvent)
            {
                throw e;
            }
        }
    }
}
This dereferencer class uses a constructor function with a parameter, `signedMessage`, to make the enveloped signature document available to the `dereference()` method. Since the reference in an enveloped signature always refers to the root of the signed data, the `dereference()` method writes the document into a byte array and returns it.

### Dereferring Uris in enveloping and detached signatures

When the signed data is located in the same document as the signature itself, the URIs in the references typically use XPath or XPointer syntax to address the elements that are signed. The W3C Recommendation for XML Signature Syntax and Processing only recommends this syntax, so you should base your implementation on the signatures you expect to encounter (and add sufficient error checking to gracefully handle unsupported syntax).

The signature of an AIR application is an example of an enveloping signature. The files in the application are listed in a Manifest element. The Manifest element is addressed in the Reference URI attribute using the string, 

“#PackageContents”, which refers to the Id of the Manifest element:
A dereferencer for validating this signature must take the URI string containing, "#PackageContents" from the Reference element, and return the Manifest element in a ByteArray object. The "#" symbol refers to the value of an element Id attribute.

The following example implements a dereferencer for validating AIR application signatures. The implementation is kept simple by relying on the known structure of an AIR signature. A general-purpose dereferencer could be significantly more complex.
package
{
    import flash.events.ErrorEvent;
    import flash.security.IURIDereferencer;
    import flash.utils.ByteArray;
    import flash.utils.IDataInput;

    public class AIRSignatureDereferencer implements IURIDereferencer {
        private const XML_SIG_NS:Namespace =
            new Namespace( "http://www.w3.org/2000/09/xmldsig#" );
        private var airSignature:XML;

        public function AIRSignatureDereferencer( airSignature:XML ) {
            this.airSignature = airSignature;
        }

        public function dereference( uri:String ):IDataInput {
            var data:ByteArray = null;
            try {
                if( uri != "#PackageContents" )
                {
                    throw( new Error("Unsupported signature type."));
                }
                var manifest:XMLList =
                    airSignature.XML_SIG_NS::Object.XML_SIG_NS::Manifest;
                data = new ByteArray();
                data.writeUTFBytes( manifest.toXMLString());
                data.position = 0;
            }
            catch (e:Error)
            {
                data = null;
                throw new Error("Reference not resolvable: " + uri + ", " + e.message);
            }
            finally
            {
                return data;
            }
        }
    }
}

When you verify this type of signature, only the data in the Manifest element is validated. The actual files in the package are not checked at all. To check the package files for tampering, you must read the files, compute the SHA256 digest and compare the result to digest recorded in the manifest. The XMLSignatureValidator does not automatically check such secondary references.

Note: This example is provided only to illustrate the signature validation process. There is little use in an AIR application validating its own signature. If the application has already been tampered with, the tampering agent could simply remove the validation check.
Computing digest values for external resources
AIR does not include built-in functions for computing SHA256 digests, but the Flex SDK does include a SHA256 utility class. The SDK also includes the Base64 encoder utility class that is helpful for comparing the computed digest to the digest stored in a signature.

The following example function reads and validates the files in an AIR package manifest:

```actionscript
import mx.utils.Base64Encoder;
import mx.utils.SHA256;

private function verifyManifest( sigFile:File, manifest:XML ):Boolean {
    var result:Boolean = true;
    var message:String = ''; 
    var nameSpace:Namespace = manifest.namespace();

    if( manifest.nameSpace::Reference.length() <= 0 )
    {
        result = false;
        message = "Nothing to validate.";
    }
    for each (var reference:XML in manifest.nameSpace::Reference) {
        var file:File = sigFile.parent.parent.resolvePath( reference.@URI );
        var stream:FileStream = new FileStream();
        stream.open(file, FileMode.READ);
        var fileData:ByteArray = new ByteArray();
        stream.readBytes( fileData, 0, stream.bytesAvailable );

        var digestHex:String = SHA256.computeDigest( fileData );
        //Convert hexadecimal string to byte array
        var digest:ByteArray = new ByteArray();
        for( var c:int = 0; c < digestHex.length; c += 2 )
        {
            var byteChar:String = digestHex.charAt(c) + digestHex.charAt(c+1);
            digest.writeByte( parseInt( byteChar, 16 ));
        }
        digest.position = 0;

        var base64Encoder:Base64Encoder = new Base64Encoder();
        base64Encoder.insertNewLines = false;
        base64Encoder.encodeBytes( digest, 0, digest.bytesAvailable );
        var digestBase64:String = base64Encoder.toString();
        if( digestBase64 == reference.nameSpace::DigestValue )
        {
            result = result && true;
            message += "   " + reference.@URI + " verified.\n";
        } else {
            result = false;
            message += " ---- " + reference.@URI + " has been modified!\n";
        }
        base64Encoder.reset();
    }
    trace( message );
    return result;
}
```
The function loops through all the references in the Manifest element. For each reference, the SHA256 digest is computed, encoded in base64 format, and compared to the digest in the manifest. The URIs in an AIR package refer to paths relative to the application directory. The paths are resolved based on the location of the signature file, which is always in the META-INF subdirectory within the application directory. Note that the Flex SHA256 class returns the digest as a string of hexadecimal characters. This string must be converted into a ByteArray containing the bytes represented by the hexadecimal string.

To use the mx.utils.SHA256 and Base64Encoder classes in an HTML-based AIR application, you can compile a library SWF containing the classes using the Flex SDK and add the SWF to your HTML page using a `<script>` tag.

**Dereferencing URLs in detached signatures referencing external data**

When a URI refers to an external resource, the data must be accessed and loaded into a ByteArray object. If the URI contains an absolute URL, then it is simply a matter of reading a file or requesting a URL. If, as is probably the more common case, the URI contains a relative path, then your IURIDereferencer implementation must include a way to resolve the paths to the signed files.

The following example uses a File object initialized when the dereferencer instance is constructed as the base for resolving signed files.

```javascript
package
{
    import flash.events.ErrorEvent;
    import flash.events.EventDispatcher;
    import flash.filesystem.File;
    import flash.filesystem.FileMode;
    import flash.filesystem.FileStream;
    import flash.security.IURIDereferencer;
    import flash.utils.ByteArray;
    import flash.utils.IDataInput;

    public class RelativeFileDereferencer
    {
        private var base:File;

        public function RelativeFileDereferencer( base:File )
        {
            this.base = base;
        }

        public function dereference( uri:String ):IDataInput
        {
            // Implementation goes here
        }
    }
}
```
The dereference() function simply locates the file addressed by the reference URI, loads the file contents into a byte array, and returns the ByteArray object.

**Note:** Before validating remote external references, consider whether your application could be vulnerable to a “phone home” or similar type of attack by a maliciously constructed signature document.
Index

Symbols
: (colon) character, in SQL statement parameter names 254
? (question mark) character, in unnamed SQL parameters 254
@ (at) character, in SQL statement parameter names 254
& (ampersand) 347

Numerics
1024-RSA 36
2048-RSA 36

A
AC_FL_RunContent() function (in default_badge.html) 362
AC_RuntimeActiveContent.js 362
accelerator keys for menu commands 160
acompc compiler 60
Acrobat 69, 293
Action Message Format (AMF) 231, 234
activate() method (NativeWindow class) 139, 145
activating windows 139, 145
active event 150
active window 144, 145
activeWindow property (NativeApplication class) 144
activity (user), detecting 338
activity event 313
addChild() method (Stage class) 141
addChildAt() method (Stage class) 142
Adobe Acrobat Developer Center 294
Adobe AIR
installing 100
introduction 6
uninstalling 2
upating 100
Adobe ColdFusion 350
Adobe documentation 9
Adobe Dreamweaver 51
Adobe Media Player 316
Adobe Press books 9
Adobe Reader 69, 293
Adobe support website 9
AES-CBC 128-bit encryption 290
AIR applications
browser invocation 124
copyright information 121
detecting installation of 366
distributing 360
exiting 328
file type associations 124, 330, 337
icons 123
installation path 119
installing 100, 360
invoking 328
launching 328
quitting 328
running 360, 367
settings 116, 118, 336
uninstalling 103
updating 100, 123, 376
versions 119, 338, 377
AIR Debug Launcher (ADL) exit and error codes 25
AIR developer certificates 371
AIR Developer Tool (ADT)
creating self-signed certificates 36
packaging an AIR file 25
signing options 32
AIR Developer Tool (ADT)
AIRI files 34
AIR files
packaging 25
signing 369
AIR HTML/JavaScript Application Introspector 40
air property (AIRAliases.js file) 58, 68
AIR runtime
detecting 338, 365
patch levels 117, 338
uninstalling 2
updating 100
AIRAliases.js file 58, 68
AIRI files
creating with the AIR Developer Tool (ADT) 34
AIRIntrospector.js file 40
AIRLocalizer.js file 396
AIRSourceViewer.js file 391
Ajax
security 110
support in the application sandbox 110
allowBrowserInvocation element (application descriptor file) 124, 329, 332
allowCrossDomainXHR attribute (frame and iframe elements) 71, 77
allowDomain() method (LocalConnection class) 358
allowInsecureDomain() method (LocalConnection class) 358
alwaysInFront property (NativeWindow class) 145
ampersand (8) 347
app URL scheme 59, 64, 70, 113, 114, 140, 193, 294
appearance of windows 135
AppInstallDisabled (Windows registry setting) 103
Apple developer certificates 371
application descriptor file 116
reading 336
application directory 188
application IDs 119
application menus 157, 166
creating 161
application sandbox 40, 53, 54, 57, 64, 69, 70, 104, 338
application storage directory 59, 102, 188, 193
application/x-www-form-urlencoded 346
applicationDescriptor property (NativeApplication class) 336
applications
See AIR applications
applicationStorageDirectory property (File class) 188
ApplicationUpdater_U1.swf file 381
ApplicationUpdater.swf file 381
app-storage URL scheme 102, 113, 114, 193, 294
app-support URL scheme 64
arguments property
BrowserInvokeEvent class 333
InvokeEvent class 329
asfunction protocol 105
asynchronous programming databases 246, 250, 269 file-system 186 XMLHttpRequests 57 at (@) character, in SQL statement parameter names 254 attach() method (SQLConnection class) 265 audio Sesound autoExit property NativeApplication class 334 AUTOINCREMENT columns (SQL) 264 auto-launch (launching an AIR application at log-in) 331

B background of windows 136 big-endian byte order 233, 351 binary data See byte arrays bitmap images, setting for icons 181 bitmaps drag-and-drop support 213, 215 bitmaps property (Icon class) 181 bounce method() (Icon class) 182 browseForDirectory() method (File class) 191 browseForOpen() method (File class) 192 browseForSave() method (File class) 192 browser invocation feature 124, 332 browserInvoke event 333, 368 BrowserInvokeEvent class 332 browsing to select a directory 191 to select a file 192 bufferTime property (SoundMixer class) 302 byte arrays byte order 233 position in 232 size of 232 byte order 233, 351 ByteArray class bytesAvailable property 232 compress() method 234 length property 232 position property 232 readBytes() method 231 readFloat() method 231 readInt() method 231 readObject() method 231 readUTFBytes() method 231 uncompress() method 234 writeBytes() method 231 writeFloat() method 231 writeInt() method 231 writeObject() method 231 writeUTFBytes() method 231 bytesAvailable property (ByteArray class) 232 bytesLoaded property (ProgressEvent class) 301 bytesTotal property (ProgressEvent class) 301

C cancelable property (Event class) 84 Canvas object 72, 79 Capabilities language property 396 languages property 396 certificate authorities (CAs) 369 certificate practice statement (CPS) 374 certificate revocation list (CRL) 374 certificates ADT command line options 32 authorities (CAs) 115 chains 374 changing 35, 373 code signing 115 expiration of 371 formats of 371 migration 35, 373 signing AIR files 369 checked menu items 161 childSandboxBridge property Window object 107 ChosenSecurity certificates 369, 371 clearData() method ClipboardData object 73 DataTransfer object 74, 215 clearing directories 198 client property (LocalConnection class) 357 clientX property (HTML drag events) 215 clientY property (HTML drag events) 215 Clipboard 73 data formats 228, 229 security 223 System 222 Clipboard class generalClipboard property 222 setData() method 229 setDataHandler() method 229 clipboard event 74 clipboardData property (clipboard events) 74 clipboardData property (HTML copy-and-paste events) 223, 224 ClipboardFormats class 228 ClipboardTransferModes class 229 close event 150 close() method NativeWindow class 146 close() method (Sound class) 307 close() method (window object) 133 closing applications 334 closing event 86, 146, 150, 334 closing windows 134, 146, 334 code signing 115, 369 colon (:) character, in SQL statement parameter names 254 columns (database) 244 Command key 160 command-line arguments, capturing 329 commands, menu See menu items communication between applications 357 complete event 62, 82, 300 compress() method (ByteArray class) 234 compressing data 234 CompressionAlgorithm class 234 computeSpectrum() method (SoundMixer class) 309 connect() method (XMLSocket class) 351 connecting to a database 250 content element (application descriptor file) 122 contenteditable attribute (HTML) 217 contentType property (URLRequest class) 346 context menus 157 HTML 163 ContextMenu class 160 contextmenu event 163 ContextMenuIte class 160 Control key 160 cookies 73 copy and paste default menu items (Mac OS) 227 deferred rendering 229
data validation, application invocation 333
databases
    about 243
    asynchronous mode 246
    changing data 265
    columns 244
    connecting 250
    creating 247
    data typing 255, 268
    deleting data 265
    errors 265
    fields 244
    files 243
    in-memory 247
    multiple, working with 265
    performance 255
    primary keys 263, 264
    retrieving data 256
    row identifiers 264
    rows 244
    security 255
    structure 244
    synchronous mode 246
    tables 244, 248
    uses for 243
dataFormat property (URLLoader class) 349
DataTransfer object
    types property 217
DataTransfer object (HTML drag and drop) 74, 215, 216, 217
Date objects, converting between ActionScript and JavaScript 62
deactivate event 150
debugging 40
    using ADL 24
decode() method (URLVariables class) 347
default_badge.html 362
deferred rendering (copy and paste) 229
deflate compression 234
DELETE statement (SQL) 265
deleteDirectory() method (File class) 198
deleteDirectoryAsync() method (File class) 198
deleteFile() method (File class) 200
deleteFileAsync() method (File class) 200
deleting directories 198, 200
deleting files 200
description element (application descriptor file) 121
descriptor-sample.xml file 116
designKey property (Document object) 75, 217
desktop directory 188
desktop windows
    See windows
desktopDirectory property (File class) 188
dictionary class 58
digital rights management 316
digital signatures 25, 32, 369
dimensions, windows 123
directories 188, 196
    application invocation 329
    copying 197
    creating 196
    deleting 198, 200
    enumerating 197
    moving 197
    referencing 188
directory chooser dialog boxes 191
displayEvent() method (NativeWindow class) 134
display order, windows 145
display() method (NativeMenu class) 164
displaying event 159, 166
displays
    See screens
displayState property (Stage class) 151
displayStateChangeEvent 134, 151
displayStateChanging event 134, 151
distributing AIR applications 360
dock icons 182
    bouncing 182
    menus 161
    support 181
    window minimizing and 145
dock menus 158
Document object
    createElement() method 57
designMode property 75, 217
    stylesheets property 62
    write() method 75, 110
    writeln() method 57, 110
documentation, related 9
documentRoot attribute (frame and iframe elements) 64, 69, 77, 107
documentRoot attributes (frame and iframe elements) 107
documents directory 188
documentRoot property (File class) 188
domininitialize event 78
downgrade attacks and security 115
FLV videos, encryption of 316
FMRMS (Flash Media Rights Management Server) 316
frame elements 69, 71, 77
frames 107
full-screen windows 151
Function constructors (in JavaScript) 71
functions (JavaScript) 56
definitions 109
literals 109

generalClipboard property (Clipboard class) 222
getApplicationVersion() method (air.swf file) 366
getData() method
ClipboardData object 73
DataTransfer object 74, 217
HTML copy-and-paste event 224
getData() method (of a dataTransfer property of an HTML drag event) 215
getDefaultApplication() method (NativeApplication class) 337
getDirectoryListing() method (File class) 197
getDirectoryListingAsync() method (File class) 197
getResult() method (SQLStatement class) 263
getScreensForRectangle() method(Screen class) 154
getStatus() method (air.swf file) 365
GlobalSign certificates 369, 371
GZIP format 234

H
hasEventListener() method 88
height element (application descriptor file) 123
Hello World sample application 52
hiding windows 145
home directory 188
hostContainer property (PDF) 295
HTML
AIR extensions to 77
copy and paste 224
debugging 40
drag-and-drop support 213, 215
embedded objects 69
events 82
overlaying SWF content 141
plug-ins 69
printing 70
sandboxes 70
scrolling 82
security 63, 69, 106
windows 140
HTML DOM and native windows 133
htmlBoundsChanged event 82
htmlDOMInitialize event 82
HTMLLoader class
copy and paste 223
createRootWindow() method 140, 141
events 82
JavaScript access to 69
loadString() method 111
paintsDefaultBackgroundColor property 136, 142
pdfCapability property 293
placeLoadStringContentInApplicationSa
ndbox property 111
htmlLoader property (Window object) 69, 76, 139
htmlLoader property (window object) 140
HTMLPDFCapability class 293
HTTP tunneling 351

I
Icon class
bitmaps property 181
bounce() method 182
icon element (application descriptor file) 123
icon property (NativeApplication class) 181
icons
animating 181
application 123
dock 181, 182
images 181
removing 181
system tray 181
task bar 145
taskbar 181
id element (application descriptor file) 119
id element (NativeApplication class) 336
id3 event 300, 308
id3 property (Sound class) 308
ID3Info class 298
IDataInput and IDataOutput interfaces 351
idle time (user) 338
idleThreshold property (NativeApplication class) 338
iframe elements 69, 71, 77, 107
img tags (in TextField object contents) 105
Info.plist files (Mac OS) 121
initialWindow element (application descriptor file) 122, 133
in-memory databases 247
innerHTML property 57, 75, 109
INSERT statement (SQL) 268
installApplication() method (air.swf file) 367
installFolder element (application descriptor file) 121
installing AIR applications 360
INTERRUPT PRIMARY KEY columns (SQL) 264
introspector (AIR HTML/JavaScript Application Introspector) 40
invoke event 329
InvokeEvent class 125, 329
arguments property 329
currentDirectory property 329
invoking AIR applications 328
ioError event 300
isBuffering property (Sound class) 302
isDirectory property (File class) 198
isHTTPS property (BrowserInvokeEvent class) 333
isSetAsDefaultApplication() method (NativeApplication class) 337
IURIDereferencer interface 413

J
Java Cryptography Architecture (JCA) 32
Java socket server 352
JavaScript
accessing AIR APIs 57
AIR runtime and 68
AIR support for 70
AIRAliases.js file 58, 68
avoiding security errors 54
debugging 24, 40
error events 82
errors 53, 87
events, handling 86
PDF 294
programming 51
security 63
JavaScript security 108
javascript URL scheme 56, 77, 109
JavaSoft developer certificates 371
JSON 71

K
key equivalents copy and paste 227
key equivalents for menu commands 160
Keyboard class 160
KeyChain (association of encrypted data with users) 290
keyEquivalent property (NativeMenuItem class) 160
keyEquivalentModifiers property (NativeMenuItem class) 160
keystores 32, 36

L
label property (NativeMenuItem class) 227
languages, supported in the AIR application installer 120
lastInsertRowID property (SQLResult class) 263
lastUserInput property (NativeApplication class) 338
launching AIR applications 328
length property (ByteArray class) 232
lightweight windows 135
lineEnding property (File class) 196
listRootDirectories() method (File class) 188
little-endian byte order 233, 351
load event 34, 69, 71, 139
load events 57
load() method (FileReference class) 209
load() method (Sound class) 301
load() method (URLLoader class) 346
Loader class 141
loadString() method (HTMLLoader class) 111
local databases See databases
LocalConnection class 361, 368  
  allowInsecureDomain() method 358
  client property 357
  connectionName parameter 358
  send() method 357
locales, choosing for an application 396
localization 395
local-trusted sandbox 70, 104
local-with-filesystem sandbox 70, 104
local-with-networking sandbox 104
locationChange event 82
login, launching an AIR application upon 331

M
Mac OS proxy icons 138
toolbar 138
main screen 154
mainScreen property (Screen class) 154
maximizable element (application descriptor file) 123
maximize() method (NativeWindow class) 147
maximizing windows 123, 134, 147
maxSize element (application descriptor file) 123
menu application 166
  events 166
  structure 159
menu bars 159
menu items 159
  accelerator keys 160
  checked 161
  copy and paste 227
  creating 163
  data, assigning to 161
  enabled 161
  key equivalents 160
  mnemonic characters 160
  selecting 165
  states 161
menuSelect events 160
menus 157
  application 161
  classes for working with 157
  creating 161
  custom 158
  default system 158
  dock 158
  dock item 161
  event flow 159, 164
  items 159
  key equivalents 160
  pop-up 161, 164
  separator lines 163
  structure 158
  submenus 159, 162
  system tray icon 161
  system tray icons 158
types of 157
window 161, 166
menuSelect events 160
messageHandler property (PDF) 295
method property (URLLoader class) 347
microphone accessing 312
detecting activity 313
routing to local speakers 313
Microphone class 298
Microsoft authenticode certificates 371
Microsoft authenticode digital IDs 371
Microsoft Windows title bar icons 138
migrating a signature 35, 373
MIME types
  HTML copy and paste 73, 228
  HTML drag and drop 215
  minimizable element (application descriptor file) 123
  minimize() method (NativeWindow class) 147
  minimizing windows 123, 134, 145, 147
  minimumPatchLevel attribute (application descriptor file) 117
  minSize element (application descriptor file) 123
  mnemonic characters menu items 160
  mnemonicIndex property
    NativeMenuItem class 160
  modificationDate property (File class) 198
  modifier keys, menu items 160
  monitors
    See screens
    mouseDown event 148
    move event 134, 150
    moveTo() method
      File class 199
      Window object 133
    moveToAsync() method (File class) 199
    moveToTrash() method (File class) 200
    moveToTrashAsync() method (File class) 200
    moving directories 197
    moving event 150
    moving files 199
    moving windows 134, 148, 149
    MP3 files 299, 308
    My Documents directory (Windows) 188
N
name element (application descriptor file) 120
name property (File class) 198
named parameters (in SQL statements) 254
native menus
See menus
native windows
See windows
NativeApplication class 76
activeWindow property 144
addEventListner() method 329
applicationDescriptor property 336
autoExit property 334
exit method 334
getDefaultApplication() method 337
icon property 181
id property 336
idleThreshold property 338
isSetAsDefaultApplication() method 337
lastUserInfo property 338
publisherID property 336, 370
removeAsDefaultApplication() method 337
runtimePatchLevel property 337
runtimeVersion property 337
setAsDefaultApplication() method 124
startAtLogin property 331
supportsDockIcon property 181
supportsMenu property 166
supportsSystemTrayIcon property 181
NativeApplication.setAsDefaultApplication() method 337
NativeBoundsEvent class 150
NativeMenu class 159, 164
NativeMenuItem class 159
data property 161
keyEquivalent property 160
keyEquivalentModifiers property 160
label property 227
mnemonicIndex property 160
submenu property 159
nativePath property (File class) 188, 198
NativeWindow class 132
activate method 145
activate method() 139
activate() method 145
addEventListner() method 150
alwaysInFront property 145
close() method. 146
constructor 139
dispatchEvent() method 134
events 150
instantiating 143
JavaScript access to 68
maximize() method 147
minimize() method 147
orderBehind() method 145
orderInBackOf() method 145
orderInFrontOf() method 145
orderToFront() method 145
restore() method 147
stage property 141
startMove() method 149
startResize() method 148
systemChrome property 135
systemMaxSize property 139
systemMinSize property 139
transparent property 135, 136
type property 135
visible property 139, 145
nativeWindow property
Stage class 144
Window object 68, 76
nativeWindow property (window object) 133, 139, 140
NativeWindowDisplayStateEvent class 151
NativeWindowInitOptions class 138, 139, 140
NetStream class
preloadEmbeddedMetadata() method 318
resetDRMVouchers() method 321
setDRMAuthenticationCredentials() method 317, 321
Netstream class
encrypted content, playing with 317
network byte order 351
networking
about 342
concepts and terms 342
non-application sandboxes 49, 53, 54, 64, 69, 70, 106, 220
normal windows 135
NSUserDefaults field (Mac OS) 121
O
object literals (in JavaScript) 109
objects, inspecting and debugging 41
OID column name (SQL) 264
onclick handler 56
onDocumentReady attribute 78
onload handler 108
onmouseover handler 56
open event 300
open() method
SQLConnection class 246
Window object 76, 111, 140
open() method (SQLConnection class) 247
openAsync() method (SQLConnection class) 246, 247, 250
opener property (window object) 140
order of windows 145
orderBehind() method (NativeWindow class) 145
orderInBackOf() method (NativeWindow class) 145
orderInFrontOf() method (NativeWindow class) 145
ordering windows 145
orderToFront() method (NativeWindow class) 145
orderToFront() method (NativeWindow class) 145
outerHTML properties 75
P
P12 files 371
packaging AIR files
AIR Developer Tool (ADT) 25
passwords
setting for encrypted media content 316
patch levels
AIR runtime 338
patch levels, AIR runtime 117
path delimiter (file system) 192
paths (file and directory) 192
paths, relative 193
PDF
  support for 69, 293
PDF content
  adding to AIR applications 293
JavaScript communication 294
  known limitations 296
loading 294
pdfCapability property (HTMLLoader class) 293
PFX files 371
placeLoadStringContentInApplicationSand box property (HTMLLoader class) 111
play() method (Sound class) 304
plug-ins (in HTML) 69
pop-up menus 157, 164
  creating 161
position of windows 123
position property (ByteArray class) 232
position property (SoundChannel class) 305
postMessage() method (PDF object) 295
  preloadEmbeddedMetadata() method (NetStream class) 318
preventDefault() method 83
primary keys
  databases 263
    menu items 160
print() method (Window object) 70
printing 70
private keys 32
privileges required to update the AIR runtime or an AIR application 100, 360, 367
Program Files directory (Windows) 360
programMenuFolder element (application descriptor file) 122
progress event 300
proxy icons
  Mac OS 138
publisher identifiers 336, 370
publisher name 369
publisherid file 336
publisherID property (NativeApplication class) 336, 370
reading files 201
readInt() method (ByteArray class) 231
readObject() method (ByteArray class) 231
readUTFBytes() method (ByteArray class) 231
ReferencesValidationSetting class 407
RegEx objects, converting between ActionScript and JavaScript 62
registering file types 337
relational databases
  See databases
relative paths (between files) 193
relativize() method (File class) 193
remote sandboxes 70, 104
removeAsDefaultApplication() method (NativeApplication class) 337
removeEventListener() method 87
requirements
  PDF rendering 293
resetDRMVouchers() method (NetStream class) 321
resizable element (application descriptor file) 123
resize event 134, 150
resizing event 150
resizing windows 123, 134, 148
resolvePath() method (File class) 188
Responder class 252, 263
restore() method (NativeWindow class) 147
restoring windows 134, 147
result event 252
RevocationCheckSettings class 407
rich internet applications (RIAs) 6
root volumes 188
_ROWID_ column name (SQL) 264
_ROWID_ column name (SQL) 264
rows (database) 244, 263
running AIR applications 360, 367
runtime property (Window object) 58, 68, 75, 139, 140
runtimePatchLevel property (NativeApplication class) 337
runtimeVersion property (NativeApplication class) 337
sandboxRoot property
  frame 107
iframe 107
sandboxType property
  BrowserInvokeEvent class 333
Security class 338
save() method (FileReference class) 209
scalable vector graphics (SVG) 70
scaleMode property
  Stage class 148
Screen class 153
  getScreenForRectangle() method 154
mainScreen property 154
screens property 154
screens 153
  enumerating 154
main 154
windows, moving between 154
screens property (Screen class) 154
screenX property (HTML drag events) 215
screenY property (HTML drag events) 215
script tags 40, 57, 58, 60, 71, 75, 301
src property of 109
scroll event 82
security
  Ajax frameworks 110
application sandbox 104
application storage directory 102
asfunction protocol 105
best practices 114
browser invocation feature 333
Clipboard 223
cross-domain cache 105
cross-scripting 111
CSS 106
database 255
downgrade attacks 115
dynamic code generation 109
encrypting data 290
eval() function 108
file system 112
frames 106, 107
HTML 53, 69, 106, 108
iframes 106, 107
img tags 105
installation (application and runtime) 100
JavaScript 63
JavaScript errors 53
loading content 140
non-application sandboxes 106

Q
question mark (?) character, in unnamed
  SQL parameters 254
quitting AIR applications 328

R
readBytes() method (ByteArray class) 231
readFloat() method (ByteArray class) 231
sandbox bridges 64, 107, 111
sandboxes 63, 69, 70, 104, 338
text fields 105
user credentials 114
user privileges for installation 100
window.open() 111
XMLHttpRequest 77
XMLHttpRequest objects 110
Security class
   sandboxType property 338
securityDomain property (BrowserInvokeEvent class) 333
select event 159, 165, 166
SELECT statement (SQL) 256, 268
self-signed certificates 36, 115, 369
send() method (LocalConnection class) 357
separator lines, menu 163
separator property (File class) 196
serialized objects
copy-and-paste support 222
server-side scripts 349
setAsDefaultApplication() method (NativeApplication class) 124, 337
setData() method
   ClipboardData object 73
   Clipboard method 229
DataTransfer object 74, 215, 217
setDataHandler() method (Clipboard class) 229
setDragImage() method (of a dataTransfer property of an HTML drag event) 215
setDRMAuthenticationCredentials() method (NetStream class) 317, 321
setInterval() function 56, 76, 109
setTimeout() function 56, 76, 109
Shift key 160
showing windows 145
signature validation 406
signatures
   migrating 35, 373
SignatureStatus class 407
SignerTrustSettings class 407
signing AIR files 25
size of windows 123
size property (File class) 198
size, windows 139
Socket class 350
socket connections 350
socket server 352
sound
   basics 298
classes 298
data 309
events 300
loading MP3 files 299
loading progress 301
sending to and from a server 315
Sound class 298, 304
SoundChannel class 298, 307
soundComplete event 305
SoundLoaderContext class 298
SoundMixer class 298, 302
sounds
   loading from SWF files 301
   playing 304
   streaming 302
SoundTransform class 298
soundTransform property (SoundChannel class) 307
source code, viewing 391
Source Viewer 391
spaceAvailable property (File class) 195
speakers and microphones 313
SQL
   about 244
   AUTOINCREMENT columns 264
   CREATE TABLE statement 248
data typing 255, 268
   DELETE statement 265
   INSERT statement 268
   INTEGER PRIMARY KEY columns 264
   named parameters (in statements) 254
   OID column name 264
   parameters in statements 253
   ROWID column name 264
   _ROWID_. column name 264
   SELECT statement 256, 268
   statements 252
   unnamed parameters (in statements) 254
   UPDATE statement 265
SQLConnection class
   attach() method 265
   open method 247
   open() method 246
   openAsync() method 246, 247, 250
   sqlConnection property (SQLStatement class) 252
SQLException class 252
SQLErrorEvent class 252
SQLite database support 242
See also databases
SQLMode class 252
SQLResultSet class 263
SQLStatement class 252
   execute method 253
   execute() method 256, 263
   getResult() method 263
   parameters object 253
   parameters property 253
   sqlConnection property 252
text property 252, 253, 256, 265
Stage class
   addChild() method 141
   addChildAt() method 142
displayState property 151
nativeWindow property 144
scaleMode property 139, 148
stage property
   NativeWindow class 141
StageDisplayState class 151
StageScaleMode class 139, 148
Start menu (Windows) 122
startAtLogin property (NativeApplication class) 331
startMove() method (NativeWindow class) 149
startResize() method (NativeWindow class) 148
start-up (system), launching an AIR application upon 331
   statements, SQL 252
status event 313
StatusEvent class 317
top() method (SoundChannel class) 305
strong binding of encrypted data 291
stylesheets property (Document object) 62
stylesheets, HTML
   manipulating in ActionScript 62
subErrorID property (DRMErrorEvent class) 324
submenu property
   NativeMenuButtonItem class 159
submenus 159, 162
Sun Java signing digital IDs 371
supportsDockIcon property (NativeApplication class) 181
supportsMenu property (NativeApplication class) 166
supportsSystemTrayIcon property (NativeApplication class) 181
SWF content
in HTML 69
overlaying above HTML 141
SWF files
communication with AIR applications 357
embedded sounds 301
loading via a script tag 60
synchronous programming
databases 246, 250, 269
file-system 186
XMLHttpRequests 57
system chrome 135
HTML windows 140
system log-in, launching an AIR application upon 331
system tray icons 158, 161
support 181
systemChrome property (NativeWindow class) 135
systemMaxSize property (NativeWindow class) 139
systemMinSize property (NativeWindow class) 139

T
tables (database) 244
creating 248
taskbar icons 145, 181
technical support 9
temporary directories 197
temporary files 201
text
drag-and-drop support 213, 215
text property (SQLStatement class) 252, 253, 256, 265
TextField class
copy and paste 223
img tags 105
Thawte certificates 369, 371
timestamps 371
title bar icons (Windows) 138
title element (application descriptor file) 122
toast-style windows 145
toolbar (Mac OS) 138
translating applications 395
transparent element (application descriptor file) 122
transparent property (NativeWindow class) 135, 136
transparent windows 122, 136
trash (deleting a file) 200
type property (Event class) 84
type property (File class) 198
type property (NativeWindow class) 135
types property
DataTransfer object 74
HTML copy-and-paste event 224
HTML drag event 215
types property (DataTransfer object) 217
U
uncaughtScriptException event 82
uncompress() method (ByteArray class) 234
uninstalling
AIR applications 103
AIR runtime 2
unknown publisher name (in AIR application installer) 369
unload events 75
unnamed parameters (in SQL statements) 254
UntrustedAppInstallDisabled (Windows registry settings) 103
update configuration file (update framework) 384
update descriptor file (update framework) 383
update framework 381
UPDATE statement (SQL) 265
update() method (Updater class) 376
UpdateDisabled (Windows registry settings) 103
Updater class
Update class 376
upgrading AIR applications 123, 376
URL encoding 347
url property
File class 188, 198
url property (File class) 188
URL schemes 193
URLLoader class
about 346
constructor 346
dataFormat property 349
load() method 346, 347
URLLoaderDataFormat class 349
URLRequest class 346, 347
contentType property 346
data property 347
method property 347
URLRequestMethod class 347
URLs 59
drag-and-drop support 213, 215
URLStream class 71
URLVariables class 346
decode() method 347
user activity, detecting 338
user credentials and security 114
user names
setting for encrypted media content 316
userDirectory property (File class) 188
userIdle event 338
userPresent event 338
utility windows 135
V
VeriSign certificates 371
Verisign certificates 369, 371
version element (application descriptor file) 119
versions, AIR application 338
video content encryption 316
visibility of windows 123
visible element (application descriptor file) 123
visible property
NativeWindow class 139, 145
vouchers, using with DRM-encrypted content 316
W
web browsers
detecting AIR runtime from 365
detecting if an AIR application is installed from 366
installing AIR applications from 367
launching AIR applications from 367
launching an AIR application from 332
WebKit 67, 70, 80
-webskit-border-horizontal-spacing CSS property 81
-webskit-border-vertical-spacing CSS property 81
-webskit-line-break CSS property 81
-webskit-margin-bottom-collapse CSS property 81
-webskit-margin-collapse CSS property 81
-webskit-margin-start CSS property 81
-webskit-margin-top-collapse CSS property 81
-webskit-nbsp-mode CSS property 81
-webskit-padding-start CSS property 81
-webskit-rtl-ordering CSS property 81
-webskit-text-fill-color CSS property 81
-webskit-text-security CSS property 81
-webkit-user-drag CSS property 81, 214, 216
-webkit-user-modify CSS property 81
-webkit-user-select CSS property 81, 214, 216

width element (application descriptor file) 123
Window class 132
window menus 157, 166
creating 161
Window object
childSandboxBridge property 107
close() method 133
htmlLoader object 69
htmlLoader property 76, 139, 140
moveTo() method 133
nativeWindow object 68
nativeWindow property 76, 133, 139, 140
open method 76
open() method 111, 140
opener property 140
parent property 140
parentSandboxBridge property 65, 76, 107
print() method 70
runtime property 58, 68, 75, 105, 110, 139, 140
WindowedApplication class 132
windows 132
activating 139
active 144, 145
appearance 135
background of 136
behavior 135
crime 135
classes for working with 133
closing 134, 146, 334
creating 138, 143
custom chrome 135
display order 145
event flow 134
events 150
hiding 145
initial 133
initializing 138
lightweight 135
managing 144
maximizing 123, 134, 147
maximum size 139
minimizing 123, 134, 145, 147
minimum size 139
moving 134, 148, 149, 154
non-rectangular 136
normal windows 135
order 145
position 123
properties 122
resizing 123, 134, 148
restoring 134, 147
showing 145
size 139
size of 123
stage scale modes 139
style 135
system chrome 135
transparency 122, 136
types 135
utility windows 135
visibility 123
Windows registry settings 103
write() method (Document object) 57, 75
writeBytes() method (ByteArray class) 231
writeFloat() method (ByteArray class) 231
writeInt() method (ByteArray class) 231
writeInt() method (Document object) 57, 75
writeObject() method (ByteArray class) 231
writeUTFBytes() method (ByteArray class) 231
writing files 201

X
x element (application descriptor file) 123
XML
class 58
socket server 352
XML namespace (application descriptor file) 117
XML signatures 406
XMLHttpRequest object 49, 57, 71, 77, 110
XMLList class 58
xmlns (application descriptor file) 117
XMLSignatureValidator class 406
XMLSocket class 351

Y
y element (application descriptor file) 123

Z
ZIP file format 237
ZLIB compression 234