Adobe AIR Programming Unleashed
Thanks for grabbing a copy of Adobe® AIR™ Programming Unleashed!

Adobe® AIR™ technology is dramatically changing the landscape of web development. Even prior to its 1.0 release, the excitement around this product even in beta was astounding. With each new build, more and more features were being baked into the Adobe AIR platform—pushing the reach of Web technologies further into the desktop world.

If you are a developer who has been locked inside the browser world along with the rest of us, this technology will breathe new life into both you and your projects.

The goals of this book are remarkably simple:

- Make broad strokes through the fundamentals of the Adobe AIR platform to help you get up and running as quickly as possible
- Explain concepts in plain English in an easy-to-read format
- Offer approachable standalone code samples you can download, compile, and execute to see features in action

Personally, I’ve always had trouble understanding concepts presented in software books on the first pass. Although the authors might be the supreme authorities on a subject, it’s conceivable that they sometimes forget what’s easy for them is not easy for someone just getting started.

I’ve done my best to keep the writing on the straight and narrow with regard to simplicity. I sincerely hope it serves you well.
Who Should Read This Book?

This book is for any web developers looking to leverage what they already know and apply those skills in desktop software.

The Adobe AIR platform supports applications developed with HTML, AJAX, Adobe® Flex™, Adobe® Flash®, PDF, or virtually any combination thereof. I should note, however, that this title leans more toward Adobe AIR application development with Adobe Flex serving as the primary citizen.

If you’re also new to Adobe Flex, don’t worry. The examples presented within the chapters are approachable for newcomers.

Software Requirements

Adobe Flex Builder 3 has everything you need to build applications for the Adobe AIR platform. It is a commercial product available in standard and professional versions. However, if you are an educator or student, you can obtain your copy free by visiting this Adobe website:

www.flexregistration.com

Standalone software development kits (SDK) are available for both Adobe Flex and Adobe AIR. Both are entirely free. Combined with your favorite IDE, you can build Adobe AIR applications at no cost beyond your own time. In addition, the Flex SDK is now open source! Nightly builds are available to the public. For information on downloads or submitting a patch or to simply peruse the bug database, visit

http://opensource.adobe.com/wiki/display/flexsdk/Flex+SDK

Adobe AIR

Adobe AIR is comprised of an SDK and a runtime component installed on the user’s machine. It’s similar to Adobe Flash, but, rather than operate within the browser context, the Adobe AIR platform offers a suite of native desktop functionality to applications. Another significant difference is that Adobe AIR applications are installed like native applications and offer direct access from the user’s desktop.

Windows Requirements

- Intel Pentium 1GHz or faster processor
- Microsoft Windows 2000 with Service Pack 4; Windows XP with Service Pack 2; or Windows Vista Home Premium, Business, Ultimate, or Enterprise
- 512MB of RAM
Mac OS X Requirements

- PowerPC G4 1GHz or faster processor or Intel Core Duo 1.83GHz or faster processor
- Mac OS X v10.4.9 or later, 10.5.1 (Intel processor required for H.264 video)
- 512MB of RAM

For Adobe AIR applications leveraging the full-screen video playback features of the integrated Adobe Flash player, the following configurations are recommended:

Windows

- Intel Pentium 2GHz or faster processor
- Windows 2000 with Service Pack 4; Windows XP with Service Pack 2; or Windows Vista Home Premium, Business, Ultimate, or Enterprise
- 512MB of RAM; 32MB of VRAM

Mac OS X

- PowerPC G4 1.8GHz or faster processor or Intel Core Duo 1.33GHz or faster processor
- Mac OS X v.10.4.9 or later or 10.5.1 (Intel or PowerPC; Intel processor required for H.264 video)
- 512MB of RAM; 32MB of VRAM

Adobe Flex

Adobe Flex Builder 3 is an Integrated Development Environment (IDE) based on Eclipse in which you can code, build, test, and optimize Adobe Flex applications. It also comes with built-in Adobe AIR support, including debug support that allows developers to quickly launch and test applications without having to package and deploy. Adobe Flex Builder offers a single environment no matter what the nature of your project.

Adobe Flex Builder 3 can be downloaded via the Adobe website:

www.adobe.com/products/flex/features/flex_builder/

For information on upgrades and an Adobe Flex feature comparison chart, visit

www.adobe.com/products/flex/upgrade/

Development of Adobe Flex Builder 3 for Linux is underway at the time of this writing. For more information, visit

http://labs.adobe.com/technologies/flex/flexbuilder_linux/

Adobe Flex Builder 3 for Windows (Standard and Professional) Requirements

- Intel Pentium 4 processor
- Microsoft Windows XP with Service Pack 2 or Windows Vista Home Premium
1GB of RAM (2GB recommended)
500MB of available hard-disk space (additional 500MB required for plug-in configuration)
Java Virtual Machine: Sun JRE 1.4.2, Sun JRE 1.5 (included), IBM JRE 1.5, or Sun JRE 1.6
Eclipse 3.2.2–3.4 for plug-in configuration (Eclipse 3.3–3.4 recommended for Windows Vista)
Adobe Flash Player 10 software (see following note)
BEA Workshop 10.1
IBM Rational Software Architect 7.0.0.3 (Eclipse 3.3 plug-in configuration only)

Adobe Flex Builder 3 for Mac OS (Standard and Professional)

- PowerPC G4 1.25GHz or Intel processor
- Mac OS X v10.4.7–10.4.10 or 10.5
- 1GB of RAM (2GB of RAM recommended)
- 500MB of available hard-disk space
- Java Virtual Machine: JRE 1.5 or JRE 1.6 from Apple
- Eclipse 3.2.2–3.4 (for plug-in configuration)
- Adobe Flash Player 10 software

---

NOTE

When installing Adobe Flex Builder 3, the latest version of the Adobe Flash Player 10 is also installed. You can verify the version of the player by visiting Adobe’s website: http://kb.adobe.com/selfservice/viewContent.do?externalId=tn_15507.

Adobe Flex 3 SDK

Although Adobe Flex Builder 3 offers a seamless environment for Adobe Flex and Adobe AIR development, they are not mandatory. The Adobe Flex SDK on its own contains everything needed to build Adobe Flex applications from a command line.
In other cases, even if you’re developing applications in Adobe Flex Builder 3, you still need to download the SDK if you’re planning on using a build process (for example, Apache ANT). The requirements for Adobe Flex 3 SDK are as follows:

- Windows 2000, Windows XP, or Windows Server 2003, Java 1.4 (Sun, IBM, or BEA) or 1.5 (Sun)
- Mac OS X v10.4.x, Java 1.5 (as shipped from Apple) on PowerPC or Intel processor
- Red Hat Enterprise Linux 3 or 4, SUSE 10, Java 1.4 (Sun, IBM, or BEA) or 1.5 (Sun)
- Solaris 9, 10, Java 1.4 or 1.5 (Sun) Compilers only
- 512MB of RAM (1GB recommended)
- 200MB of available hard-disk space

**Code Samples for This Book**

Every concept introduced in this book is backed up with a complete code sample. Each of these is available as a standalone Adobe AIR project that can be built and run inside of Adobe Flex Builder.

For your added convenience, all project files have been made available on Google Code. Simply install the Subversion Eclipse plug-in directly into Adobe Flex Builder, point to the code repository, and sync! See Appendix C, “Downloading Source Code for *Adobe AIR Programming Unleashed*,” for instructions on checking out the code files.

Optionally, all code will also be available as a Zip archive at the following location: www.informit.com/title/9780672329715.
Creating windows in Adobe® AIR™ applications is a significant departure from traditional webcentric Adobe® Flex™ development. For starters, Adobe AIR applications run on the user’s desktop. So the “windows” we’re referring to originate from the underlying native operating system, as with any other desktop software. Web developers no longer need to rely on Adobe Flex TitleWindow, JavaScript pop-ups, or browser windows propped up as a poor substitute for the real thing.

Implementing any kind of windowlike container in Adobe Flex today serves as a reminder of the limitations imposed on the user experience by the browser environment. At first glance, a TitleWindow resembles the idiom of a “windowed interface,” but users soon discover their artificial nature. They cannot be minimized to the taskbar or dragged to a secondary screen as with native windows.

**For Adobe Flex Beginners**

A TitleWindow is a layout container in the Adobe Flex framework (mx.containers.TitleWindow). It’s most often used as a pop-up container. Although it can be moved independent of the underlying Adobe Flex application, its movement is limited to the confines of the browser window.

Another option in achieving a multiwindow interface is to launch additional browser windows. There is no arguing the fact that this approach does deliver native windows, but this approach brings about a new set of challenges.

First, browser pop-up windows offer limited control over their appearance and behavior. Second, and more important,
there is a high cost in complexity when loading and communicating with content hosted in this context. In the case of Adobe Flex applications, we’re talking about a Shockwave Flash (SWF) file compiled from MXML, hosted in a single browser window. Any additional Flash or Hypertext Markup Language (HTML) content loaded in a browser pop-up does not exist as part of your Adobe Flex application. Any communication between the two needs to be brokered by other means—either by maintaining a LocalConnection or by writing a whack of JavaScript code!

Windows in Adobe AIR

Coding my first Window examples in Adobe AIR gave me a warm and fuzzy feeling. Sure, they look and behave like native windows, but the real benefit resides in the application framework itself. All windows of an Adobe AIR application exist in the same context.

For example, picture a main application window designed as a drawing canvas with a second, smaller window off to the side as a floating tool palette. For the drawing canvas to “hear” and react to button click events in the tool palette, such as the user selecting a new drawing tool, an event listener can be added on the tool palette directly from the main canvas.

This is made possible in Adobe AIR by having all windows tied to our application available as an Array in an application scope.

```javascript
var arrayOfOpenWindows:Array = NativeApplication.openedWindows;
```

In this chapter, we look at different methods of window creation and where they’re applicable in an Adobe AIR application. In addition, we look at moving beyond the default system chrome and investigate what’s involved in creating custom window chrome.

Let’s start with three window classes available to us in Adobe AIR:

- **flash.display.NativeWindow**—The lowest common denominator in terms of windows in Adobe AIR. Content such as SWFs, images, and HTML can be added to them, whereas other window types wrap this base functionality and offer extended behavior.

- **mx.core.WindowedApplication**—An application container used to house Adobe Flex applications and deliver desktop functionality. This type can only serve as the root window of an application and is configured via the application.xml file.

- **mx.core.Window**—Also a container for housing Adobe Flex content but can be instantiated any number of times. Adobe Flex developers will rely on this type most of the time.

Creating Windows Using NativeWindow

NativeWindow can be used to host an array of content such as HTML, Adobe® Flash® SWF files, or images. It is not, however, intended for use with Adobe Flex components directly. Instead, please refer to “Creating Windows Using mx.core.Window” later in this chapter.

A special type of NativeWindow, HTMLLoader.createRootContent(), exists specifically for hosting HTML content. It includes the necessary machinery for loading HTML as well as support for scrolling content.
For now let’s start with the basics. Here’s how to go about creating and configuring a NativeWindow:

- Create and configure NativeWindowInitOptions.
- Create an instance of NativeWindow, passing in NativeWindowInitOptions.
- Open the Window onscreen.

Listing 5.1 outlines these steps in ActionScript code. If you have downloaded the source code for this book, then you will find the correlating project in your FlexBuilder called "Chapter05-01".

LISTING 5.1 Creating a NativeWindow

```xml
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication
 xmlns:mx="http://www.adobe.com/2006/mxml"
 layout="vertical" verticalAlign="middle" horizontalAlign="center">

<mx:Script>
<![CDATA[
 private function openWindow():void
 {
  var windowOptions:NativeWindowInitOptions = new NativeWindowInitOptions();
  windowOptions.systemChrome = NativeWindowSystemChrome.STANDARD;
  windowOptions.type = NativeWindowType.NORMAL;

  var newWindow:NativeWindow = new NativeWindow( windowOptions );
  newWindow.activate();
 }
 ]]>}
</mx:Script>

<mx:Button label="Create Window" click="openWindow()" />

</mx:WindowedApplication>
```

Setting NativeWindowInitOptions

NativeWindow initialization options, NativeWindowInitOptions, describe the look and behavior of your window. Once set, these parameters are passed into the constructor when instantiating the NativeWindow instance. These options are not mandatory because they all have default values. For instance, not passing in NativeWindowInitOptions gives you a
standard-looking window for your operating system with standard window controls. As we progress through this chapter, we explore how we can change this default behavior—but keep in mind that after the window is created, these options cannot be changed! Table 5.1 outlines the configurable options.

Let’s explore what each of these NativeWindowInitOptions are and how they affect the characteristics of a new native window. First up is the systemChrome. The chrome is what frames the content of a native window.

Table 5.1 Properties of NativeWindowInitOptions

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>systemChrome</td>
<td>Specifies the type of system chrome used by the window</td>
</tr>
<tr>
<td>type</td>
<td>Specifies the type of the window to be created</td>
</tr>
<tr>
<td>maximizable</td>
<td>Specifies whether the window can be maximized</td>
</tr>
<tr>
<td>minimizable</td>
<td>Specifies whether the window can be minimized</td>
</tr>
<tr>
<td>resizable</td>
<td>Specifies whether the window can be resized</td>
</tr>
<tr>
<td>transparent</td>
<td>Specifies whether the window supports transparency and alpha blending against the desktop</td>
</tr>
</tbody>
</table>

NativeWindowInitOptions.systemChrome

The frame that encompasses a window is referred to as the chrome. The chrome typically offers controls to manipulate the window, such as minimize, drag, resize, and close.

There are three options for systemChrome, as shown in the following sections.

NativeWindowSystemChrome.STANDARD This option creates a standard-looking native window as per the operating system the Adobe AIR application is running on (see Figure 5.1). Also, the transparent property of the window must be set to false (which is the default value). The following snippet demonstrates how to set the systemChrome to standard, which is also the default value if none is specified.

```javascript
var windowOptions:NativeWindowInitOptions = new NativeWindowInitOptions();
windowOptions.systemChrome = NativeWindowSystemChrome.STANDARD;
```

Figure 5.1 Standard system chrome on Mac OS X and Windows XP.
NOTE

The standard chrome is managed by the operating system, and your application has no direct access to the controls themselves. You can, however, react to the events that are dispatched as a result of the user interacting with these controls. (See “Understanding Window Events” later in this chapter.)

**NativeWindowSystemChrome.NONE**  This option specifies that the window should not display any system chrome whatsoever. Creating a `NativeWindow` with no chrome generates a rectangle onscreen with no controls. This is the starting point for implementing custom chrome discussed later in this chapter. The following demonstrates how to specify no system chrome:

```javascript
var windowOptions:NativeWindowInitOptions = new NativeWindowInitOptions();
windowOptions.systemChrome = NativeWindowSystemChrome.NONE;
```

**NativeWindowInitOptions.type**
Each window offers unique traits suited for different roles in an application. There are three `NativeWindowTypes` to choose from:

- NORMAL
- UTILITY
- LIGHTWEIGHT

**NativeWindowType.NORMAL**  This is the default window type. If nothing is specified for this parameter in your `NativeWindowInitOptions`, Figure 5.2 shows what is displayed.

![Windows XPMac OS X](image)

**FIGURE 5.2**  Default window type on Mac OS X and Windows XP

NORMAL windows have typical controls such as minimize, maximize, and close. Their physical characteristics match that of any standard window on each respective operating system.
NativeWindowType.UTILITY In Figure 5.3, you see the same system chrome but differences in both physical and behavioral aspects of NativeWindow.

![Figure 5.3](image)

**FIGURE 5.3** UTILITY windows have a slimmer title bar, and they don’t show up in the Windows taskbar or the Mac OS X Dock (note the lack of a Minimize icon).

Often used as containers for supporting content or tool palettes, these windows do not serve as the primary focus of an application. Their content may change as events happen in the main application window, such as displaying properties of an object that has received focus.

**NOTE**

There are applications that utilize this window type as its primary user interface. These are typically smaller, more specialized applications such as instant messaging or media players. There isn’t a need to crowd the user’s Dock or taskbar with an application running in the background most of the time.

NativeWindowType.LIGHTWEIGHT LIGHTWEIGHT NativeWindows have no chrome whatsoever. In fact, you’ll get a runtime error unless you specifically set the systemChrome property to NONE. Creating a window in this fashion gives you a white box that can’t be moved or even closed directly. Figure 5.4 demonstrates a native window with no chrome and uses a bitmap image as the window’s background.

![Figure 5.4](image)

**FIGURE 5.4** An Adobe AIR application implemented with custom window chrome.
Creating Windows Using NativeWindow

Uses for LIGHTWEIGHT NativeWindows range from custom system chrome implementations to toast messages (dialogs that temporarily slide up onscreen like toast out of a toaster) to drawer dialogs common on Mac OS X.

NativeWindowInitOptions.transparent
This property refers to the transparency of the window background window. A transparent window has no default background. Any area not occupied by a display object is invisible; for example, whatever lies beneath your application window shows through.

You can also change the alpha property of your display objects to allow underlying desktop content to show through.

**CAUTION**
Display objects with an alpha setting of less than .06 (approximately) prevent the window from capturing mouse events in that area. It will appear as though you have clicked the object behind the window.

**NOTE**
You cannot create transparent windows in combination with any system chrome.

NativeWindowInitOptions.maximizable
When this property is set to false, the window cannot be maximized. For a window with system chrome, this affects the appearance of the window Maximize button, such as making it appear disabled.

**NOTE**
On Mac OS X, you’ll have to set both the maximizable and resizable options to false to prevent the window from being zoomed or resized.

NativeWindowInitOptions.minimizable
When this property is set to false, the window cannot be minimized. As with a window with system chrome, this affects the appearance of the window Minimize button.

NativeWindowInitOptions.resizable
When this property is set to false, the window cannot be resized.

**NOTE**
As with the NativeWindowInitOptions.maximizable property, on Mac OS X, you’ll have to set both the maximizable and resizable options to false to prevent the window from being zoomed or resized.
Creating an Instance of the Window

Now we need to create a new NativeWindow instance. Remember that the properties defined in NativeWindowInitOptions cannot be changed after we instantiate the window. The default window size is determined by the operating system, but you can change it by setting the window bounds. (We’ll look at this later in the chapter.)

```javascript
var newWindow:NativeWindow = new NativeWindow( windowOptions );
```

The variable `windowOptions` refers to the `NativeWindowInitOptions` we constructed in the previous section.

Putting the Window Onscreen

If we were to stop at the previous step, the user would not see anything appear onscreen. After instantiating our `NativeWindow`, we need to specifically put it on the screen. There are two ways this can be accomplished:

```javascript
NativeWindow.activate()
```

or

```javascript
NativeWindow.visible = true
```

Using `NativeWindow.activate()`

Invoking the `activate()` method on the `NativeWindow` instance does the following:

- Makes the window visible
- Brings the window to the front
- Gives the window keyboard and mouse focus

The following snippet instantiates a new `NativeWindow`, passing in window options, followed by the `activate()` method.

```javascript
var newWindow:NativeWindow = new NativeWindow( windowOptions );
newWindow.activate();
```

Using `NativeWindow.visible`

This property specifies whether the window is visible on the desktop. It affects only visibility and does not give the window focus or bring it to the front.

For example, you might want to open a supporting `UTILITY` type window for an application where focus must remain on the primary window. Rather than activating your window, simply set its `visible` property to `true`, and it appears onscreen without the primary window flashing in and out of focus.

By default, `visible` is set to `false`. To make the window visible, do the following:

```javascript
var newWindow:NativeWindow = new NativeWindow( windowOptions );
newWindow.visible = true
```
Creating Windows Using `mx.core.Window`

The Adobe Flex `mx.core.Window` class essentially wraps `NativeWindow` and facilitates the addition of Adobe Flex content. As an Adobe Flex developer, you will find yourself using this class to create windows in most cases.

The steps to creating a `Window` differ slightly from `NativeWindow`:

- Create an instance of `Window`.
- Set `Window` properties (optional—there are defaults).
- Open the `Window` on the Screen.

**NOTE**

Rather than include full class path on each mention of `mx.core.Window`, we use “Window” instead—capitalizing the “W.”

If we’re just referring to the generic term “window,” it is not capitalized.

Let’s take a look at a simplistic example of instantiating a `Window` instance and opening it onscreen. (See Listing 5.2)

**LISTING 5.2 Simple Example of Using `mx.core.Window`**

```xml
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication
    xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical"
    verticalAlign="middle" horizontalAlign="center">
    <mx:Script>
        <![CDATA[
            import mx.core.Window;
        ]]>
    </mx:Script>
</mx:WindowedApplication>
```
private function openWindow():void
{
    var myWindow:Window = new Window();
    myWindow.systemChrome = NativeWindowSystemChrome.STANDARD;
    myWindow.type = NativeWindowType.NORMAL;
    myWindow.open( true );
}

</mx:Script>
<mx:Button label="Create Window" click="openWindow()" />
</mx:WindowedApplication>

Creating an Instance of Window
Using the Adobe Flex Window class, we create an instance:

var myWindow:Window = new Window();

Notice there is no NativeWindowInitOptions object passed into the constructor of Window. You can now set those same properties directly on the Window instance itself, as you will see demonstrated in the following section.

NOTE
Although a number of window properties can now be set after the Window instance has been created, certain properties still follow the rule of having to be applied before a window is opened onscreen, for example, systemChrome, type, and so on. After they're set, they cannot be changed.

Setting Window Properties
Using mx.core.Window differs from NativeWindow in that we can set all parameters after it has been instantiated. The one exception is the nativeWindow property of Window; this is not accessible until we open it onscreen.

To create a window using mx.core.Window, do the following:

var myWindow:Window = new Window();
myWindow.systemChrome = NativeWindowSystemChrome.STANDARD;
myWindow.type = NativeWindowType.NORMAL;

NOTE
You can still use the same static variables from the NativeWindow classes because they are essentially just resolving to strings.
As with `NativeWindow`, you have the same options to choose from with regard to both the chrome of the window instance and the window type. There are some differences in the results of these options which we'll take a closer look at now.

**Chrome Options for `mx.core.Window`**

Creating a `Window` with standard window chrome yields the same result as with `NativeWindow`. After all, `mx.core.Window` is essentially a `NativeWindow` primed to host Adobe Flex content. The only visual difference visually is the gray background, which represents the Adobe Flex content area (see Figure 5.5).

![Figure 5.5](image)

**Figure 5.5** `mx.core.Window` of type `NORMAL` with standard system chrome.

![Figure 5.6](image)

**Figure 5.6** `mx.core.Window` of type `UTILITY` with standard system chrome.

![Figure 5.7](image)

**Figure 5.7** `mx.core.Window` of type `NORMAL` with `NONE` system chrome. By default Adobe Flex displays its own chrome.

![Figure 5.8](image)

**Figure 5.8** `mx.core.Window` of type `UTILITY` with `NONE` system chrome.
### CHAPTER 5  Working with Windows

**FIGURE 5.9**  mx.core.Window of type LIGHTWEIGHT with NONE system chrome.

**NOTE**

Although the options for window types are the same as NativeWindow, a difference lies in how you deal with windows with systemChrome set to NONE. When systemChrome is set to NONE, Adobe Flex displays its own system chrome. You can disable this by setting the showFlexChrome property to false on your Window instance.

At times, you will still need to access the underlying NativeWindow properties. For example, moving a window from one location onscreen to another requires setting the x and y coordinates of NativeWindow (see Listing 5.3). You won’t find those properties on the parent mx.core.Window class.

**LISTING 5.3** Referencing nativeWindow Properties When Using mx.core.Window

```javascript
var myWindow:Window = new Window();
myWindow.systemChrome = NativeWindowSystemChrome.STANDARD;
myWindow.type = NativeWindowType.NORMAL;
myWindow.open( true );
myWindow.nativeWindow.x = 100;
myWindow.nativeWindow.y = 100;
```

**Opening a Window Onscreen**

Finally, to open a Window onscreen, use the open() method. Although the Window defaults to “active,” you have the option to change this via a Boolean passed in with the method call as follows:

```javascript
newWindow.open( true );
```

Passing false into the open method will cause the Window to open but not make it active. In other words, give the window focus.

**Getting a Window Reference**

Before you can work with a particular window, you first need to get a reference of that Window instance. The following sections describe the various ways to obtain a Window reference.
Window Constructor
You can use the window constructor for a new NativeWindow to get a reference, like this:
```
var myWindow:NativeWindow = new NativeWindow();
```

Current Window Stage
You can get a reference directly from the current window stage, as follows:
```
stage.nativeWindow
```

Display Object on the Stage
Any display object on the stage can also give you a reference, as follows:
```
aDisplayObject.stage.nativeWindow
```

As an example, suppose you have an mx.containers.Panel in some window. To get the reference to the parent NativeWindow instance, you can do this:
```
myPanel.stage.nativeWindow
```

Referencing the Active Window
A desktop window that currently holds user focus is referred to as the “active” window. You can reference this window via NativeApplication, as follows:
```
var myWindow:NativeWindow = NativeApplication.nativeApplication.activeWindow;
```

NOTE
If the active window on the desktop is not associated with your application, `activeWindow` returns a null value.

Referencing All Opened Windows
All open windows can be referenced via the nativeApplication object. These can be cycled through like any Array. Each element will be a NativeWindow instance.
```
var myWindows:Array = NativeApplication.nativeApplication.openedWindows;
```

Window Operations
In this section we look into controlling a Window's dimensions, positioning and behaviors.
**Resizing a Window**

You can invoke a resize action on a window by calling the following method:

```
NativeWindow.startResize();
```

NOTE

The resize functionality only exists in `NativeWindow`. In your `Window` instance of type `mx.core.Window` or `mx.core.WindowedApplication`, you need to call the `startResize()` method on the `nativeWindow` property of your window. (`Window` and `WindowedApplication` are essentially just an Adobe Flex wrapper on `NativeWindow`.)

The next code example (as shown in Listing 5.4) demonstrates an `mx.core.Window` being created with a button that initiates the resize of that same window from the lower-right corner. (Click and hold the Start Resize button and drag your mouse to resize the window.)

**LISTING 5.4 Initiating Window Resize**

```xml
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication
 xmlns:mx="http://www.adobe.com/2006/mxml"
 layout="vertical"
 verticalAlign="middle"
 horizontalAlign="center">

<mx:Script>
 <![CDATA[
 import mx.controls.Button;
 import mx.core.Window;
 private var myWindow:Window;

 private function openWindow():void
 {
  var dragButton:Button = new Button();
  dragButton.label = "Click, hold and drag mouse";
  dragButton.addEventListener( MouseEvent.MOUSE_DOWN, resizeWindow );
  myWindow = new Window();
  myWindow.width = 300;
  myWindow.systemChrome = NativeWindowSystemChrome.STANDARD;
  myWindow.type = NativeWindowType.NORMAL;
  myWindow.setStyle( "horizontalAlign", "center" );
  myWindow.setStyle( "verticalAlign", "middle" );
  myWindow.addChild( dragButton );
  myWindow.open( true );
 }]
 ]]>

</mx:Script>
</mx:WindowedApplication>
```
Listing 5.4 is an oversimplified example for sake of clarity. A more realistic use case would involve having graphic elements within a custom window chrome initiate this resize behavior. (See “Creating Custom Window Chrome” later in this chapter.)

**Moving a Window**

To move a window, call the `startMove()` method on the `NativeWindow` instance. If you’re using `mx.core.Window`, reference the underlying `NativeWindow` via the `nativeWindow` property:

```ActionScript
var myWindow:Window = new Window();
myWindow.open();
myWindow.nativeWindow.startMove();
```

**Maximizing, Minimizing, and Restoring a Window**

Maximizing causes a window to expand to the bounds of the current screen. To maximize a window, use

```
NativeWindow.maximize();
```

To minimize a window, use

```
NativeWindow.minimize();
```

To restore a window, use

```
NativeWindow.restore();
```

Restoring a window simply means that the window will return to the size that it was before it was either minimized or maximized.

**Closing a Window**

To close a window, use

```
NativeWindow.close()
```

Closing a window empties the contents of the window, but if any other objects have references to that content, the content objects are not destroyed. You can check the `closed`
property of a window to test whether a window has been closed. If the window being closed is the last one, and the `NativeApplication.autoExit` property is set to `true` (the default setting), the application quits.

**Understanding Window Events**

An event-based programming model is used to interact with `NativeWindows`, so let's take a look at what happens when an event takes place before we get into any specific operations.

For some `NativeWindow` operations, there are two associated events. The first dispatched event notifies you that something is *about to happen*, allowing you the opportunity to interject with a callback function. The second event tells you that something *has already happened*.

You’ll have to register a listener with that particular window instance to handle these events. The listener catches any of the events and allows you to execute logic using a callback function. In other words, “when object xyz dispatches a certain event, execute this particular function I’ve defined.”

Suppose a user clicks the Close button of a window. An event is dispatched to notify listeners that a window is about to close, giving our application a chance to react. We might want to prompt the users to save their work if they haven’t done so already. If the users choose to save, we’d first invoke the necessary functionality to save, and after that’s done, trigger the window to close. If our users don’t want to save their work, our callback function logic simply does nothing, and the window closes. Now, a second event is dispatched signaling that the window has finished closing.

Listing 5.5 shows an example in which we add event listeners for both `Event.CLOSING` and `Event.CLOSE` on an instance of `mx.core.Window`.

**LISTING 5.5 Exploring Window CLOSE and CLOSING Events**

```xml
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication
   xmlns:mx="http://www.adobe.com/2006/mxml"
   layout="vertical"
   verticalAlign="middle"
   horizontalAlign="center">

<mx:Script>
  <![CDATA[
    import mx.core.Window;

    private function openWindow():void
    {
      var myWindow:Window = new Window();
```
myWindow.systemChrome = NativeWindowSystemChrome.STANDARD;
myWindow.type = NativeWindowType.NORMAL;
myWindow.open( true );

myWindow.nativeWindow.addEventListener( Event.CLOSE, onWindowClose );
myWindow.nativeWindow.addEventListener( Event.CLOSING, onWindowClosing );
}

private function onWindowClosing( event:Event ):void
{
    trace( "Window is about to close" );
}

private function onWindowClose( event:Event ):void
{
    trace( "Window has closed" );
}
]]>
</mx:Script>

<mx:Button label="Create Window" click="openWindow()" />
</mx:WindowedApplication>

**NOTE**

Event.CLOSE will not fire from mx.core.Window. You must listen to its parent
NativeWindow to be notified of the event. This is because the Adobe Flex context is
destroyed after the CLOSING event fires and is unavailable to dispatch the final CLOSE
event.

**Canceling a Window Event**

Often you’ll need to intercept an event and invoke conditional logic to determine whether
you want that event to continue, such as in the example cited earlier in Listing 5.5.
In that example we’re simply tracing a message to the output console, but in the real
world, you may want to prompt users that their work isn’t currently saved and ask if they
want to do so.

Listing 5.6 outlines how to interrupt the closing sequence by catching the CLOSING event
and calling preventDefault() on the event object. This stops the event in its tracks. In
this example we’re only doing this if isWorkSaved is false, indicating the user has
attempted to close the application without saving his or her work.

Our Alert dialog makes a callback to onAlertClose, upon which time we act on the users’
decision to save their work. When that has been done, we can simply call the close() method on our Window. We’re also calling exit() because this is our main application
Window we’re closing. If we didn’t call `exit()`, the Window would close but the application process would still be running, so it’s important to keep that in mind!

Here’s how we could add to our example in code Listing 5.5:

LISTING 5.6 Cancelling a Window CLOSING event

```xml
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication
    xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical"
    verticalAlign="middle"
    horizontalAlign="center"
    initialize="init()">

    <mx:Script>
        <![CDATA[
        import mx.events.CloseEvent;
        import mx.controls.Alert;

        private var isWorkSaved:Boolean = false;

        private function init():void
        {
            nativeWindow.addEventListener( Event.CLOSING, onWindowClosing );
        }

        private function onWindowClosing( event:Event ):void
        {
            if( !isWorkSaved )
            {
                event.preventDefault();
                Alert.show( "Would you like to save your work?", "Warning!", Alert.YES | Alert.NO, this, onAlertClose );
            }
        }

        private function onAlertClose( event:CloseEvent ):void
        {
            if( event.detail == 1 )
            {
                // Save users work here
                isWorkSaved = true;
                trace( "Work has been saved" );
            }
        }

        nativeWindow.close();
        ]]>}
    </mx:Script>
</mx:WindowedApplication>
```
exit();

NOTE

If you are using a custom window chrome, then it will be up to you to programmatically dispatch the CLOSING and CLOSE events.

Creating Custom Window Chrome

Adobe AIR projects generated from the New Flex Project Wizard in Adobe Flex Builder output a default MXML file with a root tag called WindowedApplication. As outlined earlier in this chapter, this gives your application a standard native window as expected.

What if a project calls for a truly customized window chrome, such as a fully branded look and feel that includes custom icons for window controls and a nonrectangular shape?

No sweat—this can be accomplished by the following steps:

1. Set the window chrome to none and the transparency to true in the application's descriptor file (see Listing 5.6).
2. On WindowedApplication set the showFlexChrome to false.
3. Create a Canvas with an embedded background image (optional).

In Listing 5.6 we've changed the chrome and transparency properties in the application descriptor, which prevents Adobe AIR from opening a visible default Window when the application is launched. This, in combination with setting the showFlexChrome to false in our application code (see Listing 5.7) delivers the desired effect.

LISTING 5.7 Modifying Window Properties in the Application Descriptor File

<!-- Settings for the application's initial window. Required. -->
<initialWindow>
    <!-- The main SWF or HTML file of the application. Required. -->
    <!-- Note: In Flex Builder, the SWF reference is set automatically. -->
    <content><![CDATA[This value will be overwritten by Flex Builder in the output app.xml]]></content>

    <!-- The title of the main window. Optional. -->
    <!-- <title>Custom Chrome</title> -->

    <!-- The type of system chrome to use (either "standard" or "none"). -->
    <!-- Optional. Default standard. -->
Embedding an image as your application’s background is completely optional. At this point you literally have a blank slate to work with inside your Adobe Flex application. You can use a circular or square background image or perhaps draw your application background yourself via the ActionScript drawing APIs, it’s up to you. See Figure 6.10.

FIGURE 5.10  Example of using a bitmap image as the custom chrome for a Window.
In Listing 5.8 we’ve opted to simply embed a bitmap image and used that as the background. In addition we’ve included a drop shadow filter on the Canvas that gives a floating perspective to the application.

LISTING 5.8  Creating a Window with Custom Chrome in Adobe Flex

```xml
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication
    xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical"
    horizontalScrollPolicy="off"
    verticalScrollPolicy="off"
    showFlexChrome="false"
    creationComplete="init()"/>

<mx:Style source="styles.css" />

<mx:Script>
    <![CDATA[
        import mx.controls.Label;

        private function init():void
        {
            myCanvas.addEventListener( MouseEvent.MOUSE_DOWN, moveWindow );
            var dropShadow:DropShadowFilter = new DropShadowFilter();
            var glow:GlowFilter = new GlowFilter(0x000000,1,5,5,3);
            var filters:Array = new Array( dropShadow, glow );
            myCanvas.filters = filters;
        }

        private function moveWindow( event:MouseEvent ):{
            stage.nativeWindow.startMove();
        }

        private function onMinimize():void
        {
            stage.nativeWindow.minimize();
        }

        private function onClose():void
        {
            stage.nativeWindow.close();
        }
    ]]>
</mx:Script>
```
There is a little added work going this route because you have to create your own mechanisms for standard window controls such as minimize, maximize, and so on. In Listing 5.7 we’ve added listeners on our window control images and explicitly, via the event handlers, initiated the desired window behavior.

**Summary**

We’ve explored how to create windows onscreen using both the *flash.desktop.NativeWindow* and *mx.core.Window* classes. In essence *mx.core.Window* is just a wrapper for the *NativeWindow* class, making it ready to host elements of an Adobe Flex application.

As for the look and feel of your application windows, the sky is the limit. If the standard operating system chrome won’t do the trick, then you can build your own customized chrome from scratch.
Index

A

Action Message Format (AMF), 228, 325
ActionScript, 3
    keyEquivalent properties, 174
ActionScript classes, 102
activate() method, windows, 74
active windows, referencing, 79
AC_FL_RunContent() function, 293
AC_FL_RunContent() JavaScript function, 293
addAsync() method, parameters, 385
addCommand() method, 348
addImage() function, 242
adding
    event listeners for InvokeEvent, 38
    management destinations to LCDS, 433
    test cases to test suites, FlexUnit, 379-380
AddItemCmd class, 347-348
AddItemView.mxml file, 353-355
ADL (Adobe AIR Debug Launcher), 54-56, 273, 362
Adobe AIR
    APIs, 12
    architecture of, 12-13
    compiling, 363
compiler background information, 363
writing build targets, 364
creating Hello World application, 29-36
exporting, 365-367
installing, 15-17
overview, 11
running, 365
updating, 309-314
via remote servers, 314-316
Adobe AIR Debug Launcher (ADL), 54-56
Adobe AIR Developer Tool, 367
Adobe AIR HTML Introspector, debugging, 57-60
Adobe AIR menus
   currentTarget properties, 175
   keyEquivalent properties, 173-175
Adobe AIR messaging applications, creating with BlazeDS, 229-238
Adobe Flash, 10, 439Ω
Adobe Flash Debugger (FDB), 56
Adobe Flex AIR components, 251
   FileSystemComboBox, 251-252
   FileSystemDataGrid, 257-258
   FileSystemList, 253-255
   FileSystemTree, 255-256
Adobe Flex applications, tips for, 279
Adobe Flex Builder, 365
Adobe Flex Builder 3
   installing, 17-20
   perspectives, 22
Adobe Flex Builder debugger, 52-53
Adobe Flex SDKs, installing, 26-27
Adobe Flex software development kit (SDK), documentation, 341
Adobe LiveCycle Data Services (LCDS), 191
ADT (Adobe AIR Developer Tool), 362, 367
   parameter descriptions for exporting, 366
   self-signed certificates for testing, creating, 367-368
Agile software development process, 373
.air, 366
AIR Debug Launcher, 273
.air distributable archive, 372
.air files, 315
AIRBadge() function, 297
AIRChat, example, 231-238
AIRIntrospector.js, 59
airversion parameter (FlashVars), 294
AMF (Action Message Format), 228, 325
Apache Ant, 359
   downloading and configuring, 360
   reasons for success, 359
   running FlexUnit from, 386
      compatibility modifications, 386-387
      compiling unit tests, 387-388
      running unit tests, 388-389
      unified reporting with JUnit, 389-390
Apache Flex Ant tasks, downloading and configuring, 360
APIs of Adobe AIR, 12
application descriptor files, 43-45
application frameworks, 322
application invoke events, 37-39
Application Menu, 161-163
application sandboxes, 264-266
application settings, reading, 43-45
application shutdown process, 41-43
application startup process, 36-37
application invoke events, 37-39
launching from browsers, 40
launching from the command line, 40
launching on login, 40
application/vnd.adobe.air-install-package mime type, 284
applicationComplete event, 279
applications
digital signatures, 303-306
Certificate Wizard, 304-306
overview, 299-301
signature.xml file, 302-303
installation
one-off installs, 285-291
overview, 283-285
seamless install feature, 291-299
shopping cart application (Cairngorm)
AddItemCmd class, 347-348
AddItemView.mxml file, 353-355
CancelAddCmd class, 346
CartController class, 348
CartItem class, 337-338
CartModel class, 338-339
CartView.mxml file, 350-353
ecart.mxml file349-350
Java server-side component, 335
LoadProductsCmd class, 344-345
LoadProductsEvent class, 343-344
overview, 335
PrepAddCartCmd class, 345-346
Product class, 336-337
ProductDelegate class, 341-342
ServiceLocator class, 340-341
applicationStorageDirectory, 103
appname parameter (FlashVars), 294
appurl parameter (FlashVars), 294
architecture of Adobe AIR, 12-13
architecture frameworks, 323
Cairngorm, 325
design patterns, 323-324
interacting with services, 325
managing states, 324
managing user gestures, 324
reducing tight coupling, 325
transferring data, 325
architectures, Cairngorm, 335
Are You Sure You Want to Install This Application to Your Computer? Dialog, 286
asynchronous database connections, establishing, 193
asynchronous database operations
versus synchronous database, 202, 204-206
versus synchronous database operations, 202
asynchronous file operations versus synchronous file operations, 107-108
asynchronous mode, executing multiple statements against a database, 205

How can we make this index more useful? Email us at indexes@samspublishing.com
asynchronous testing, FlexUnit, 382-385
automated builds, continuous integration, 396
automated deployment, continuous integration, 397

B

badge.fla file, 292
badge.swf file, 292
bindable variables, creating, 339
BITMAP_FORMAT data type, 137-143
BlazeDS
Adobe AIR messaging applications, creating, 229-238
installing, 228-229
messaging, 227-228
BlazeDS technology, 341
blogs, 454
Bonjour protocol, 422
bouncing Mac OS X Dock icon, 178
branches, version control system (continuous integration), 396
browseForOpen, 105
browser windows, 68
BrowserInvokeEvent, 41
browsers, launching from, 40
BUILD_FOLDER, 362
build targets, writing, 364
build tools, 359, 452
build.xml
creating, 361-362
final project, 368-370
compiling, 370
exporting, 370
running, 370
builds, automated builds (continuous integration), 396
business delegates, Cairngorm, 333-334
buttoncolor parameter (FlashVars), 294
ByteArray, 206

C

Cairngorm, 325-326
categories, 327
design patterns
business delegates, 333-334
FrontController, 330-332
ModelLocator, 328-330
ServiceLocator, 332-333
value objects, 327-328
microarchitecture, 326-327
Cairngorm Contact Manager application, 415
eMBEDDED_DATABASES, 416
OFFLINE_SCENARIOS, 417
Cairngorm microarchitecture, shopping cart application
AddItemCmd class, 347-348
AddItemView.mxml file, 353-355
CancelAddCmd class, 346
CartController class, 348
CartItem class, 337-338
CartModel class, 338-339
CartView.mxml file, 350-353
ecart.mxml file, 349-350
LoadProductsCmd class, 344-345
LoadProductsEvent class, 343-344
overview, 335
PrepAddCartCmd class, 345-346
Product class, 336-337
ProductDelegate class, 341-342
ServiceLocator class, 340-341
Cairngorm Microarchitecture Framework, contact manager with integrated Yahoo! maps application, 413
callRemoting() function, 216
CancelAddCmd class, 346
Canceling window events, 83-85
CartController class, 348
CartItem class, 337-338
CartModel class, 338-339
CartView.mxml file, 350-353
Certificate Wizard, 304-306
certificates
  security, 452
  self-signed certificates for testing creating applications, 367-368
  exporting, 367-368
CFC (ColdFusion Component), 430
  fill method inside assembler CFC, 432
  listener for event gateway, 431
checkout, version control system (continuous integration), 395
chrome, creating custom window chrome, 85-88
classes
  ActionScript classes, 102
  AddItemCmd, 347-348
  CancelAddCmd, 346
  CartController, 348
 CartItem, 337-338
  CartModel, 338-339
  LoadProductsCmd, 344-345
  LoadProductsEvent, 343-344
  NativeApplication class, toast messages, 179-184
  NativeMenu, 160-161
    Application Menu, 161-163
    Context Menu, 166
    Dock and System Tray Menu, 167-168
    Flex Menu, 170-171
    Pop-Up Menu, 169-170
    Window Menu, 163-165
  NativeWindow class
    bouncing Mac OS X Dock icon, 178
    statusBar notifications, 184-185
    system tray icon ToolTips, 187-188
    TaskBar highlighting, 185-187
  PrepAddCartCmd, 345-346
  Product, 336-337
  ProductDelegate, 341-342
  ServiceLocator, 340-341
How can we make this index more useful? Email us at indexes@samspublishing.com
Timer, 316
Updater, 309-311
Clipboard, deferring renderings, 154-156
Clipboard classes, 129-130
  BITMAP_FORMAT data type, 137-143
  Clipboard class, 130
  ClipboardTransferMode class, 130
  FILE_LIST_FORMAT data type, 147-151
  HTML_FORMAT data type, 142-147
  TEXT_FORMAT data type, 131-137
  URL_FORMAT data type, 151-154
ClipboardTransferMode class, 130
closing windows, 81
CLOSING event, 42
Coenraets, Christophe, 209
ColdFusion, 427
  LCDS, 433-436
  Remoting, 214-222
  ServiceCapture, 57
    watch folder process, 429-432
ColdFusion Component, 430
command line, launching from, 40
commits
daily commits, continuous integration, 396
version control system, continuous integration, 395
compiling
Adobe AIR applications, 363
  compiler background information, 363
  writing build targets, 364
build.xml, final project, 370
  unit tests, 387-388
Concurrent Versioning System (CVS), 395
CONFIG_FOLDER, 362
configuration file for watch folder event gateway, 430
configuring
  Apache Ant, 360
  CruiseControl, 399-402
connecting data services from Adobe Flex, 435
connections
  asynchronous database connections, establishing, 193
  LocalConnection objects, 238-249
  synchronous database connections, establishing, 192
contact manager with integrated Yahoo! maps application
  overview, 412-413
  synchronization, 415
    embedded database, 415-417
    offline scenarios, 417-419
    third-party components, 413
      Cairngorm Microarchitecture framework, 413
      PromptingTextInput, 414
      Yahoo! maps, 414-415
contact value objects, 201
ContactModel, 329
Context Menu, 166
continuous integration, 393-394, 453
  automated builds, 396
  automated deployment, 397
CruiseControl, 397
daily commits, 396
unit testing, 396-397
version control system, 394-395
branches, 396
checkout, 395
commits, 395
mainline, 395
merging, 396
coupling, reducing design patterns, 325
createTempDirectory() method, 113
creationComplete event, 279
CruiseControl, 397-398
configuring, 399-402
Dashboard, 403-405
downloading, 398
rebuilding, 406-407
starting, 402-403
currentTarget properties, Adobe AIR menus, 175
CVS (Concurrent Versioning System), 395

D
daily commits, continuous integration, 396
Dashboard, CruiseControl, 403-405
data
inserting into databases, 196-199
reading
in encrypted local store, 275
from tables, 200
removing from encrypted local store, 275
storing in encrypted local store, 275
transferring with design patterns, 325
Data Management Services, 341
Data Protection API, 274
data services, connecting from Adobe Flex, 435
data types, Clipboard classes
BITMAP_FORMAT, 137-143
FILE_LIST_FORMAT, 147-151
HTML_FORMAT, 142-147
TEXT_FORMAT, 131-137
URL_FORMAT, 151-154
database query results, 200-202
database tables, 194
creating, 191-192
example of creating an initial table structure, 194-196
databases
asynchronous database connections, establishing, 193
encrypted databases, 206-208
inserting data, 196-199
query results, 200-202
re-encrypting databases with new keys, 208
reasons for using local SQL databases, 191
retrieving primary keys of inserted rows, 199-200
SQLite, 190
synchronous database connections, establishing, 192

How can we make this index more useful? Email us at indexes@samspublishing.com
synchronous versus asynchronous, 202
writing asynchronous database, 204-206
writing synchronous database, 202-204

Davidson, James Duncan, 359

debugging
with Adobe AIR Debug Launcher (ADL), 54-56
with Adobe AIR HTML Introspector, 57-60
with Adobe Flash Debugger (FDB), 56
with Adobe Flex Builder debugger, 52-53
content loaded into HTML controls, 61
with third-party tools, 56-57
default_badge.html file, 293-297
deferring Clipboard renderings, 154-156
deleting directories, 114
deployment, automated deployment, 397
Descriptor, toast messages, 182
design patterns
architecture frameworks, 323-324
interacting with services, 325
managing states, 324
managing user gestures, 324
reducing tight coupling, 325
transferring data, 325
Cairngorm
business delegates, 333-334
FrontController, 330-332
ModelLocator, 328-330
ServiceLocator, 332-333
value objects, 327-328
detecting user presence, 47-48
development resources, 453
dialogs, Are You Sure You Want to Install This Application to Your Computer?, 286
digital signatures, 34, 303-306
Certificate Wizard, 304-306
overview, 299-301
signature.xml file, 302-303
directories, 113
as static constants, 103
creating new, 113
deleting, 114
flash.filesystem.File, 113
retrieving directory listings, 114
temporary directories, creating, 113
directory paths, 102
display objects, z-axis, 445-448
distributable archives, .air, 372
distributing Adobe AIR applications
digital signatures, 303-306
Certificate Wizard, 304-306
overview, 299-301
signature.xml file, 302-303
one-off installs, 285-291
overview, 283-285
seamless install feature, 292-299
badge.fla files, 292
badge.swf files, 292
default_badge.html file, 293-297
FlashVars parameters, 293
getApplicationVersion() method, 298
getStatus() method, 297-298
installApplication() method, 298-299
launchApplication() method, 299
overview, 291
doAdd() function, 355
doCancel() function, 355
Dock and System Menu, 167-168
Dock icon (Mac OS X), bouncing, 178
doCopy() function, 137
documentation, Adobe Flex software development kit (SDK), 341
doPaste() function, 136-137
downloading
   Apache Ant, 360
   Apache Flex Ant tasks, 360
   CruiseControl, 398
   FlexUnit, 374
   FlexUnit Ant tasks, 374-375
   zip files, 462
DPAPI (Data Protection API), 274
drag-and-drop, 115-116
   drag-in gestures, 121-127
   drag-out gestures, 116-121
drag-in gestures, 121-127
drag-out gestures, 116-121

E
ecart.mxml file, 349-350
Eclipse, perspectives, 22
event gateway, CFC listener for, 431
embedded databases, Cairngorm Contact Manager application, 415-417
encrypted databases, 206-208
encrypted local store, 273-275
   reading data, 275
   removing data, 275
   storing data, 275
event gateway, event gateway, CFC listener for, 431
error(), Adobe AIR HTML Introspector, 60
establishing, file associations, 45-46
event listeners, adding to InvokeEvent, 38
events, canceling window events, 82-83
execute() function, 345-346
executing multiple statements against a database in asynchronous mode, 205
EXITING event, 41
exporting
   Adobe AIR applications, 365-367
   build.xml, final project, 370
   extensions, .air, 366
F
fault() function, 345
FDB (Adobe Flash Debugger), 56
file associations, establishing, 45-46
File object, 102
   File.url, 102-104
file paths, 102
   File.nativePath, 104
   File.desktopDirectory, 103
   File.nativePath, 104
   File.url, 102-104
   FileMode, 109
files

AddItemView.mxml, 353-355
badge.fla, 292
badge.swf, 292
CartView.mxml, 350-353
creating and writing, 109-112
default_badge.html, 293-297
ecart.mxml, 349-350
FileStream, security sandboxes, 265
FileStream object, 106
FileStreamComboBox, 251-252
FileStreamDataGrid, 257-258
FileStreamList, 253-255
fileSystemTree, 102, 255-256
FILE_LIST_FORMAT data type, 147-151
Flash Remoting, 214-222
FlashVars parameters, 293
Front Controller pattern, 330
full-screen mode, 99-100
FlexMenu, 170-171
FlexUnit, 373, 452
async testing, 382-385
downloading, 374
running from Apoch Ant, 386
compatibility modifications, 386-387
compiling unit tests, 387-388
running unit tests, 388-389
unified reporting with JUnit, 389-390
task parameters, 388
test cases
adding to test suites, 379-380
creating, 375-377
creating visual test case runners, 380-382
implementing setup() and tear-down(), 378-379
naming conventions, 377
running, 379-380
FlexUnit Ant tasks, downloading, 374-375
frameworks, 322
application frameworks, Spring Framework, 322
architecture frameworks, 323
Cairngorm, 325
design patterns, 323-325
libraries, 322
FrontController, Cairngorm, 330-332
full-screen mode, 99-100
functions
AC_FL_RunContent(), 293
AC_FL_RunContent() JavaScript, 293
AIRBadge(), 297
doAdd(), 355
doCancel(), 355
execute(), 345-346
fault(), 345
getProducts(), 342
initApp(), 350
initializeDatabase(), 195
resetView(), 355
result(), 345

G
getAddressVersion() method, 297-298
getClipBoard() function, 133
getFlightDetails(), 216
getProducts() function, 342
getStatus() method, 297-298

H
hasFormat() method, 125
Hello World application, creating, 29-36
hierarchical nature of Adobe AIR menus, 171-173
HTML controls, debugging loaded content, 61

HTMLLoader, security sandboxes, 265
HTML_FORMAT data type, 142-147
HTTP protocol, Java Mini Web Server, 423
HTTPService, 222-225

I
ICommand interface, 331, 345
IDEs, 451
images, proxy images, 115
imageurl parameter (FlashVars), 294
IModelLocator interface, 338
info(), Adobe AIR HTML Introspector, 60, 214
initApp() function, 350
initializeDatabase(), 195
inserting data into databases, 196-199
installApplication() method, 297-299
installing
Adobe AIR, 15-17
digital signatures, 299-306
one-off installs, 285-291
overview, 283-285
seamless install feature, 291-299
Adobe Flex Builder, 317-20
Apache Ant, 360
Apache Flex Ant tasks, 360
BlazeDS, 228-229
SDKs (software development kits)
  Flex, 26-27
  Java, 23
Subversion, 457-462

How can we make this index more useful? Email us at indexes@samspublishing.com
instances
mx.core.Window, creating, 76
NativeWindow, creating, 74
instant messaging, 276
integration, continuous integration, 453
interacting with services, design patterns, 325
interface, IValueObject, 336
interfaces
  ICommand, 331, 345
  IModelLocator, 338
  IResponder, 345
InvokeEvent, 38
IResponder interface, 345
IValueObject interface, 336

J
Java
  adding to system paths, 24-26
  installing, 23
Java Development Kit, 23
Java Mini Web Server, 423
Java Runtime Environment (JRE), 23
JUnit, unified reporting, 389-390

K
keyEquivalent properties, Adobe AIR menus, 173-175
keys, re-encrypting databases with new keys, 208
launchApplication() method, 297-299
launching
  from browsers, application invoke events, 40
  from the command line, application invoke events, 40
  on login, application invoke events, 40
launching applications
  application startup process, 36-37
    application invoke events, 37-39
    launching from browsers, 40
    launching from the command line, 40
    launching on login, 40
LCDS (LiveCycle Data Services), 191, 428
  ColdFusion, video distribution systems, 433-436
libraries, 322, 454
loadAllProducts() method, 342
loadFiles() function, 147
LoadProductsCmd class, 344-345
LoadProductsEvent class, 343-344
local SQL databases, reasons for using, 191
local-trusted sandboxes, 264
local-with-filesystem sandboxes, 265
local-with-networkings sandboxes, 264
LocalConnection objects, 238-249
login credentials
  persisting, 277-279
  storing, 276-279
Mac OS X Dock icon, bouncing, 178
mailing lists, 452
mainline, version control system (continuous integration), 395
management destinations, adding to LCDS, 433
managing
  states, design patterns, 324
  user gestures, design patterns, 324
maximizing windows, 81
McLeod, Alistair, 326
memory profiling, 61-64
menus
  Application Menu, NativeMenu class, 161-163
  Context Menu, NativeMenu class, 166
  Dock and System Tray Menu, NativeMenu class, 167-168
  Flex Menu, NativeMenu class, 170-171
  hierarchical nature of, 171-173
  Pop-Up Menu, NativeMenu class, 169-170
  Window Menu, NativeMenu class, 163-165
merging version control system (continuous integration), 396
messagecolor parameter (FlashVars), 294
messages, toast messages (NativeApplication class), 179-184
messaging with BlazeDS, 227-228
  Adobe AIR messaging applications, creating, 229-238
  installing, 228-229
methods
  activate(), 74
  addCommand(), 348
  fill, 432
  getApplicationVersion(), 297-298
  getStatus(), 297-298
  installApplication(), 297-299
  launchApplication(), 297-299
  loadAllProducts(), 342
  selectDefaultRecord(), 199
  startMove(), 81
microarchitecture, Cairngorm, 326-327
minimizing windows, 81
model-view-controller, 325
ModelLocator, Cairngorm, 328-330
moving windows, 81
MVC (model-view-controller), 325
mx:HTTPService/, 222-225
mx:TraceTarget/, 54, 220
mx:WebService/, 225-227
mx.core.Window, 68, 75-76
  chrome option, 77
  instances, creating, 76
  opening Window onscreen, 78
  Window properties, setting, 76-78
mx.core.WindowsApplication, 68
MXML components, 363
MXML tags, mx:TraceTarget/, 54
naming conventions, test cases (FlexUnit), 377
NativeApplication class, toast messages, 179-184
nativeDragDrop handler gestures, 126
nativeDragEnter event, 125
NativeDragManager, 118, 121
NativeMenu class, 160-161
    Application Menu, 161-163
    Context Menu, 166
    Dock and System Tray Menu, 167-168
    Flex Menu, 170-171
    Pop-Up Menu, 169-170
    Window Menu, 163-165
nativePath, 105
NativeWindow
    creating windows, 68-69
        NativeWindowInitOptions, setting, 69-73
    instances, creating, 74
    putting windows onscreen, 74
        NativeWindow.activate(), 74
        NativeWindow.visible, 74-75
NativeWindow class
    bouncing Mac OS X Dock icon, 178
    statusBar notifications, 184-185
    system tray icon ToolTips, 187-188
    TaskBar highlighting, 185-187
NativeWindow.activate(), 74
NativeWindow.startResize(), 80
NativeWindow.visible, 74-75
NativeWindowInitOptions, 69-73
NativeWindowInitOptions.maximizable, 73
NativeWindowInitOptions.minimizable, 73
NativeWindowInitOptions.resizable, 73
NativeWindowInitOptions.systemChrome, 70
NativeWindowInitOptions.transparent, 73
NativeWindowInitOptions.type, 71
NativeWindowSystemChrome.NONE, 71
NativeWindowSystemChrome.STANDARD, 70
NativeWindowType.LIGHTWEIGHT, 72
NativeWindowType.NORMAL, 71
NativeWindowType.UTILITY, 72
offline scenarios, Cairngorm Contact Manager application, 417-419
one-off application installation, 285-291
onLoad() function, 279
onLogin() function, 277
openBrowser() function, 239
opened windows, referencing, 79
opening
    files, 105-106
    Window onscreen, mx.core.Window, 78
P-Q

parameters
  FlashVars, 293
  FlexUnit, 388
paste() function, 137-138
patch information, retrieving, 48-49
paths
  directory paths, 102
  file paths, 102
peer-to-peer networking, 421-422
  photo-sharing example application, 422
    Bonjour protocol, 422
    Java Mini Web Server, 423
  server sockets, 423-426
performAdd() method, 385
performance, 61-62, 64
persisting user login information, 277-279
perspectives
  Adobe Flex Builder, 322
  Eclipse, 22
photo-sharing example application (peer-to-peer networking), 422
  Bonjour protocol, 422
  Java Mini Web Server, 423
  server sockets, 423-426
plug-ins, installing Subversion, 457-462
Pop-Up Menu, 169-170
positioning windows
  based on Screen bounds, 93-96
  relative to screens, 96-99
PrepAddCartCmd class, 345-346
primary keys, retrieving of inserted rows, 199-200
Product class, 336-337
ProductDelegate class, 341-342
Profiler button, 62
PromptingTextInput, contact manager with integrated Yahoo! maps application, 414
properties
  currentTarget, 175
  keyEquivalent, 173-175
proxy images, 115
publications, 454
re-encrypting databases with new keys, 208
reading
  application settings, 43-45
  data in encrypted local store, 275
  data from tables, 200
  files, 105-106
    FileMode, 109
rebuilding CruiseControl, 406-407
records, inserting into database tables, 196-199
reducing coupling with design patterns, 325
references
  Screens, obtaining references to the main screen, 91-92

How can we make this index more useful? Email us at indexes@samspublishing.com
Window reference, 78
  current stage window, 79
  displaying objects on stage, 79
  referencing active window, 79
  referencing opened windows, 79
  Window constructor, 79
  referencing nativeWindow properties with mx.core.Window, 78
remote servers, updating Adobe AIR application, 314-316
RemoteObject class, 214
remoting, 213
  BlazeDS
    Adobe AIR messaging applications, creating, 229-238
    installing, 228-229
    messaging, 227-228
    with Flash Remoting and ColdFusion, 214-222
removing data in encrypted local store, 275
resetView() function, 355
resizing windows, 80-81
resolution, Screen resolution, 92-93
resolvePath, 104
restoring windows, 81
result() function, 345
results, database query results, 200-202
retrieving
  directory listings, 114
  primary keys of inserted rows, 199-200
  version and patch information, 48-49
rows, retrieving primary keys of inserted rows, 199-200
running
  Adobe AIR applications, 365
  build.xml, final project, 370
  test cases, FlexUnit, 379-380
  unit tests, 388-389
  update() method, 310
S
sample applications, storing login credentials, 276-279
sampling, 64
sandbox bridges, 266-270
  UserAPI, 270
sandboxes, 264
saveFileToDisk() method, 112
Screens
  determining number of, 91
  full-screen mode, 99-100
  obtaining reference to the main screen, 91-92
  positioning windows based on Screen bounds, 93-96
  positioning windows relative to screens, 96-99
  resolution, 92-93
  virtual desktop, 89-90
SDKs (software development kits), 22
  installing
    Adobe Flex, 26-27
    Java, 23
    Java, adding to system paths, 24-26
seamless application installation, 292-299
  badge.fla files, 292
  badge.swf files, 292
  default_badge.html file, 293-297
  FlashVars parameters, 293
  getApplicationVersion() method, 298
  getStatus() method, 297-298
  installApplication() method, 298-299
  launchApplication() method, 299
  overview, 291
security, certificates, 452
security sandboxes, 263-265
  application sandboxes, 266
  bridges, 266-270
    UserAPI, 270
  types of, 264-265
selectDefaultRecord(), 199
self-signed certificates, creating for testing, 367-368
server sockets for photo-sharing example application, 423-426
server-side technologies, 451
servers, remote servers, updating Adobe AIR application, 314-316
Service to Worker pattern, 330
ServiceCapture, 57
ServiceLocator, Cairngorm, 332-333
ServiceLocator class, 340-341
services, interacting with design patterns, 325
setClipBoard() function, 136
setDataHandler() method, 154
setup(), test cases (FlexUnit), 378-379
Shockwave Flash (SWF), 213
Shockwave Flash (SWF) files, 68, 363
shopping cart application (Cairngorm)
  AddItemCmd class, 347-348
  AddItemView.mxml file, 353-355
  CancelAddCmd class, 346
  CartController class, 348
  CartItem class, 337-338
  CartModel class, 338-339
  CartView.mxml file, 350-353
  ecart.mxml file, 349-350
  Java server-side component, 335
  LoadProductsCmd class, 344-345
  LoadProductsEvent class, 343-344
  overview, 335
  PrepAddCartCmd class, 345-346
  Product class, 336-337
  ProductDelegate class, 341-342
  ServiceLocator class, 340-341
shutting down, application shutdown process, 41-43
signature.xml file, 302-303
signatures, digital, 34, 303-306
  Certificate Wizard, 304-306
  overview, 299-301
  signature.xml file, 302-303
Singleton pattern, 328
socket connections for photo-sharing example application, 423-426
software development kits, 22
Spring Framework, 322
SQLite, 190-191
SQLite tool, 209-211
SquirrelFish, 439
stack traces, 64
starting CruiseControl, 402-403
startMove() method, 81
states, managing with design patterns, 324
statusBar notifications, NativeWindow class, 184-185
storing
data in encrypted local store, 275
login credentials, 276-279
subtabs, Dashboard (CruiseControl), 405
subversion, installing, 457-462
Subversion (SVN), 395
SWF (Shockwave Flash), 213
SWF (Shockwave Flash) files, 363
syncAction, 416
SyncContactsCmd command, 418
syncFlag, 416
synchronization, contact manager with integrated Yahoo! maps, 415
embedded databases, 415-417
offline scenarios, 417-419
synchronous database connections, establishing, 192
synchronous database operations
versus asynchronous database, 202-206
versus asynchronous database operations, 202
synchronous file operations versus asynchronous file operations, 107-108
system paths, adding Java to, 24-26
system tray icon ToolTips, NativeWindow class, 187-188

tables, database tables, 194
creating, 191-192
example of creating an initial table structure, 194-196
tags, mx:TraceTarget/, 54
TaskBar highlighting, NativeWindow class, 185-187
tearDown(), test cases (FlexUnit), 378-379
test cases, FlexUnit
adding to test suites, 379-380
creating, 375-377
creating visual test case runners, 380-382
implementing setUp() and tearDown(), 378-379
naming conventions, 377
running, 379-380
testAddItem(), 384
testing, 452
  asynchronous testing (FlexUnit), 382-385
  self-signed certificates, creating, 367-368
tests, unit tests, 373
TEXT_FORMAT data type, 131-137
third-party components, contact manager with integrated Yahoo! maps application, 413
  Cairngorm Microarchitecture Framework, 413
  PromptingTextInput, 414
  Yahoo! maps, 414-415
third-party tools, debugging, 56-57
3D rotation effects, 440-445
Timer class, 316
TitleWindow, 67, 184
toast messages, NativeApplication class, 179-184
tools, SQLite, 209-211
Transfer Object pattern, 323
transferring data, design patterns, 325

unit tests, FlexUnit, 373
update() method, running, 310
Updater class, 309-311
updating Adobe AIR application, 309-314
  via remote servers, 314-316
URL_FORMAT data type, 151-154
user gestures, managing with design patterns, 324
user gestures, 330
USER IDLE, toast messages, 179
user notifications, 177
  bouncing the Mac OS X Dock icon, 178
  statusBar notifications, 184-185
  system tray icon ToolTips, 187-188
  TaskBar highlighting, 185-187
  toast messages, 179-184
user presence, detecting, 47-48
USER PRESENT, toast messages, 179

V
value objects, Cairngorm, 327-328
variables, creating bindable variables, 339
version control, 453
version control system, continuous integration, 394-395
  branches, 396
  checkout, 395
  commits, 395
  mainline, 395
  merging, 396

How can we make this index more useful? Email us at indexes@samspublishing.com
version information, retrieving, 48-49
video distribution systems
  application overview, 428
  ColdFusion
    LCDS, 433-436
    watch folder process, 429-432
views, 324
virtual desktop, 89-90
visibility, 75
visual test case runners, FlexUnit, 380-382

W-X

warn(), Adobe AIR HTML Introspector, 60
watch folder event gateway, configuration file for, 430
web servers, Java Mini Web Server, 423
Webkit HTML engine, 439
WebService, 225-227
websites
  Bonjour protocol information, 423
  Java Mini Web Server information, 423
  Zeroconf protocol information, 423
Webster, Steven, 326
window classes
  flash.display.NativeWindow, 68
  mx.core.Window, 68
  mx.core.WindowedApplication, 68
Window constructor, 79
window events, 82-83
  canceling, 83-85
Window Menu, 163-165
Window properties, mx.core.Window, 76-78
Window reference, 78
  current stage window, 79
  displaying objects on stage, 79
  referencing active window, 79
  referencing opened windows, 79
Window constructor, 79
windows, 67-68
  browser windows, 68
  closing, 81
  creating custom window chrome, 85-88
  creating with mx.core.Window, 75-76
    instances, 76
    opening Window onscreen, 78
    Window properties, setting, 76-78
  creating with NativeWindow, 68-69
    setting NativeWindowInitOptions, 69-73
  maximizing, 81
  minimizing, 81
  moving, 81
NativeWindow, 74
  positioning based on screen bounds, 93-96
  positioning relative to screens, 96-99
  resizing, 80-81
  restoring, 81
TitleWindow, 67
wizards, Certificate Wizard, 304-306
writing
   asynchronous database operations, 204-206
   build targets, 364
   files, 109-112
   synchronous database operations, 202-204

Y
   y-axis, 441
   Yahoo! maps
      contact manager with, 413
      overview of, 412
      contact manager with integrated Yahoo! maps application, 414-415

Z
   z-axis, 441
   display objects, 445-448
   Zero Configuration Networking, 422
   Zeroconf protocol, 422
   zip files, downloading, 462

How can we make this index more useful? Email us at indexes@samspublishing.com