Stacy Tyler Young
Michael Givens
Dimitrios Gianninas

Covers **version 1.5** of Adobe AIR

Adobe® AIR™ Programming

UNLEASHED

SAMS

ADOBE® AIR™

ADOBE® AIR™ Programming Unleashed

Copyright © 2009 by Pearson Education, Inc.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-672-32971-5

ISBN-10: 0-672-32971-9

Library of Congress Cataloging-in-Publication Data

Young, Stacy Tyler.
Adobe AIR programming unleashed/Stacy Tyler Young, Michael Givens, Dimitrios Gianninas.
p. cm.
ISBN 978-0-672-32971-5
1. Cross-platform software development. 2. Internet programming. 3. Web site development. I. Givens, Michael. II. Gianninas, Dimitrios. III. Title.
QA76.76.D47Y675 2008
006.7'6—dc22

2008041640

Printed in the United States of America

First Printing November 2008

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

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

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales
international@pearson.com

Associate Publisher
Greg Wiegand

Acquisitions Editor
Laura Norman

Development Editor
Songlin Qiu

Managing Editor
Kristy Hart

Project Editor
Betsy Harris

Copy Editor
Karen Annett

Indexer
Lisa Stumpf

Proofreader
San Dee Phillips

Tech Editor
Michael Givens

Publishing Coordinator
Cindy Teeters

Book Designer
Gary Adair

Senior Compositor
Jake McFarland

Introduction

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 v10.4.910 or 10.5.1 (PowerPC); Mac OS X v10.4.9 or later, 10.5.1 (Intel)
- 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.

CHAPTER 5

Working with Windows

IN THIS CHAPTER

- Windows in Adobe AIR
- Creating Windows Using `NativeWindow`
- Creating Windows Using `mx.core.Window`
- Getting a `Window` Reference
- Window Operations
- Understanding Window Events
- Creating Custom Window Chrome

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.

```
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 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`**

Property	Description
`systemChrome`	Specifies the type of system chrome used by the window
`type`	Specifies the type of the window to be created
`maximizable`	Specifies whether the window can be maximized
`minimizable`	Specifies whether the window can be minimized
`resizable`	Specifies whether the window can be resized
`transparent`	Specifies whether the window supports transparency and alpha blending against the desktop

`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.

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

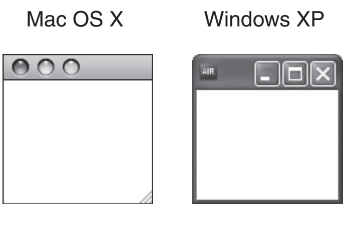

FIGURE 5.1 Standard system chrome on Mac OS X and Windows XP.

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:

```
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.

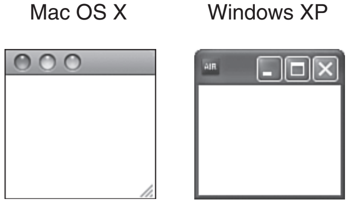

FIGURE 5.2 Default window type on Mac OS X and Windows XP.

`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`.

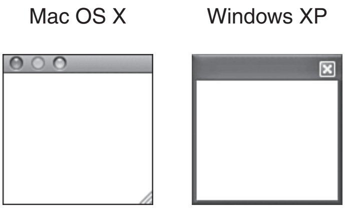

FIGURE 5.3 **`UTILITY`** windows have a slimmer title bar, and they don't show up in the Windows taskbar or the Mac OS X Dock (note the lack of a Minimize icon).

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.

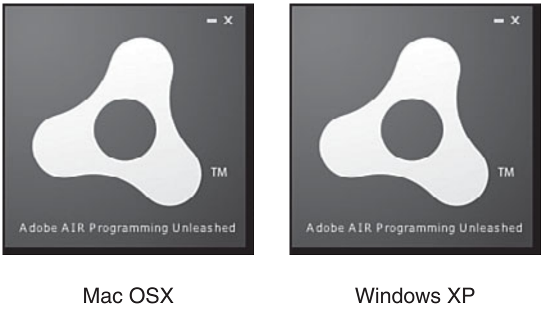

FIGURE 5.4 An Adobe AIR application implemented with custom window chrome.

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.)

```
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:

```
NativeWindow.activate()
```

or

```
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.

```
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:

```
var newWindow:NativeWindow = new NativeWindow( windowOptions );
newWindow.visible = true
```

NOTE

An invisible window isn't displayed on the desktop, but all the properties and methods are still available.

On Mac OS X, turning off visibility on a minimized window does not remove it from the Dock. The user is still able to click that Dock icon, which causes the window to be visible, restore, and have focus.

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`.

5

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 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 );
      }
      ]]>
   </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).

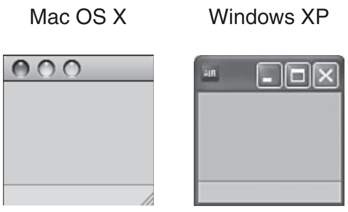

FIGURE 5.5 `mx.core.Window` of type `NORMAL` with standard system chrome.

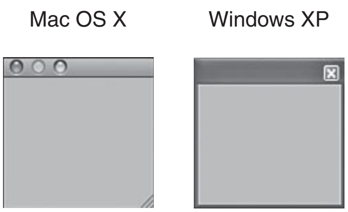

FIGURE 5.6 `mx.core.Window` of type `UTILITY` with standard system chrome.

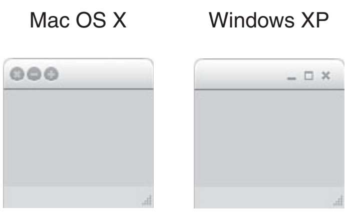

FIGURE 5.7 `mx.core.Window` of type `NORMAL` with `NONE` system chrome. By default Adobe Flex displays its own chrome.

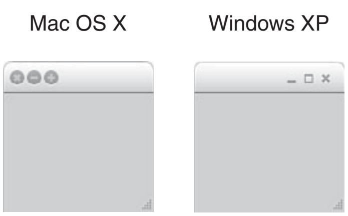

FIGURE 5.8 `mx.core.Window` of type `UTILITY` with `NONE` system chrome.

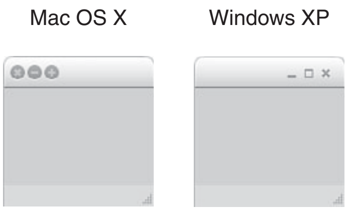

FIGURE 5.9 `mx.core.Window` of type `LIGHTWEIGHT` with `NONE` system chrome.

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`

```
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:

```
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 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 );
      }
```

```
      private function resizeWindow( event:MouseEvent ):void
      {
         myWindow.nativeWindow.startResize( NativeWindowResize.BOTTOM_RIGHT );
      }
      ]]>
   </mx:Script>
   <mx:Button label="Create Window" click="openWindow()" />
</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:


```
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 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 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();
```

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

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>[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. —>
```

5

```
   <systemChrome>none</systemChrome>
   <!— Whether the window is transparent. Only applicable when systemChrome
➥is false. Optional. Default false. —>
   <transparent>true</transparent>
   <!— Whether the window is initially visible. Optional. Default false. —>
   <visible>true</visible>
   <!— Whether the user can minimize the window. Optional. Default true. —>
   <!— <minimizable></minimizable> —>
   <!— Whether the user can maximize the window. Optional. Default true. —>
   <!— <maximizable></maximizable> —>
   <!— Whether the user can resize the window. Optional. Default true. —>
   <!— <resizable></resizable> —>
   <!— The window's initial width. Optional. —>
   <!— <width></width> —>
   <!— The window's initial height. Optional. —>
   <!— <height></height> —>
   <!— The window's initial x position. Optional. —>
   <!— <x></x> —>
   <!— The window's initial y position. Optional. —>
   <!— <y></y> —>
   <!— The window's minimum size, specified as a width/height pair,
➥such as "400 200". Optional. —>
   <!— <minSize></minSize> —>
   <!— The window's initial maximum size, specified as a width/height pair,
➥such as "1600 1200". Optional. —>
   <!— <maxSize></maxSize> —>
</initialWindow>
```

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 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 ):void
           {
              stage.nativeWindow.startMove();
           }

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

            private function onClose():void
            {
               stage.nativeWindow.close();
            }
```

5

```
        ]]>
    </mx:Script>

    <mx:Canvas
        id="myCanvas"
        width="200" height="200"
        backgroundImage="@Embed(source='assets/air.png')"
        horizontalScrollPolicy="off" verticalScrollPolicy="off">

        <mx:Image source="@Embed(source='assets/minimize.png')"
➥click="onMinimize()" x="168" y="6" alpha="0.8" />
        <mx:Image source="@Embed(source='assets/close.png')" click="onClose()"
➥x="180" y="6" alpha="0.8" />
        <mx:Label text="Adobe AIR Programming Unleashed" styleName="scores"
➥x="10" y="176" />
    </mx:Canvas>
</mx:WindowedApplication>
```

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, initiatied 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

D

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

E

F

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

J

K

L

M

N

O

P-Q

R

S

T

U

V

W-X

Y

Z