Sample Chapter

ADOBE AIR in Action

Joseph Lott
Kathryn Rotondo
Samuel Ahn
Ashley Atkins

MANNING

Adobe AIR in Action

by Joseph Lott

Chapter 2

brief contents

2 Applications, windows, and menus

This chapter covers

- Creating new windows
- Managing open windows
- Running application-level commands
- Adding system-level menus to applications

In chapter 1, you learned a lot of important foundational information that should provide context for understanding what AIR is and the general process for building and deploying AIR applications. Now we're ready to start looking at all the details of building AIR applications. We're going to start from the ground up. In this chapter, we'll cover the following:

- *Applications*—The first thing we'll look at is how to work with and understand an AIR application programmatically. In the first part of this chapter, you'll learn everything you need to know about programmatically creating AIR applications.

- *Windows*—Every AIR application, no matter how simple or complex, contains at least one window, and many contain more than one. Windows are fundamental but fairly sophisticated at the same time. AIR gives you a lot of control over windows, including window style, shape, behavior, and more. All of these topics are covered in this chapter.
- *Menus*—AIR applications allow you to create a variety of different types of menus, including system menus, application menus, context menus, and icon menus. We'll talk about all these types of menus in this chapter.

With what you learn in this chapter, you'll have much of what you need for the building of all sorts of amazing AIR applications. In fact, in this chapter you'll start work on an application that uses the YouTube service to allow users to search YouTube videos and play them back from their desktop. You'll be able to accomplish all of that using just the material contained in this chapter.

We'll get started by looking at the metaphor that AIR uses for how it organizes an application into an application object and window objects. In the next section, you'll learn how to work an application object and window objects using both Flash and Flex.

2.1 Understanding applications and windows

This may seem a bit obvious, but it's worth explicitly pointing out that AIR applications have programmatic constructs for everything that's represented visually or behaviorally in the application. That's true not only of things that you may already be familiar with from your Flex and Flash background (movie clips, buttons, UI controls) but also of AIR-specific concepts such as the concept of an application or a window.

There is just one application, and there are one or more windows per AIR application. Therefore, it follows that every Flex- or Flash-based AIR application has just one ActionScript object representing that application, providing access to application-level information (application descriptor data, user idle timeout) and behavior (registering file types, exiting the application). Furthermore, every window in a Flex- or Flash-based AIR application has an ActionScript object representing it. Those window objects provide access to window-specific information (width, height, screen placement) and behavior (minimize, restore).

Every AIR application has at least one window, the window specified as the content for the initial window in the application descriptor file. That window is what you see when you run the application. However, you can open more than one window per application. You can think of each window as a new thread of the application, much like new instances of a web browser, or you can think of each window as a panel in your application. Both are perfectly valid ways to think of and treat windows in an AIR application. It depends on what you're trying to accomplish. What's true in any case is that there's just one application ActionScript object per AIR application, and that object keeps track of all the windows in your application (see figure 2.1). We'll look at how to work with these objects and their relationships to one another throughout this chapter.

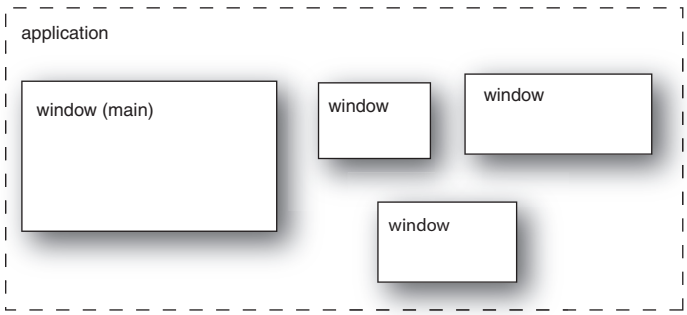


Figure 2.1 Every AIR application has one application object and one or more window objects.

The ways in which you work with an application and its windows are related, yet slightly different, depending on whether you're using Flash or Flex to build your AIR application. However, the core ActionScript principles used when building Flash-based AIR applications are fundamental both for Flash-based and Flex-based AIR application development. Therefore, if you use Flex to build AIR applications, you'll want to learn the underlying ActionScript principles as well as the Flex-specific concepts. In the following sections, we'll talk about these foundational concepts. If you use only Flash to build AIR applications, you need only read section 2.1.2. If you use Flex, read both that section and section 2.1.3.

2.1.1 ActionScript application and windows

When you're working with the intrinsic AIR ActionScript API for applications and windows, you need to understand two primary classes: `flash.desktop.NativeApplication` and `flash.display.NativeWindow`. If you're building Flash-based AIR applications, these are the only two application and window classes you'll need to work with.

CREATING AN APPLICATION

Every AIR application automatically has one instance of `NativeApplication`. You can't create more than one `NativeApplication` instance. In fact, you can't create a `NativeApplication` instance at all. The one instance is created for you when the application starts, and is accessible as a static property of the `NativeApplication` class as `NativeApplication.nativeApplication`. We'll see lots of ways you can use this `NativeApplication` instance later in the chapter.

CREATING WINDOWS

Every window in an AIR application is fundamentally a `NativeWindow` object. While the initial window is automatically created when the application starts, it's your responsibility as the application developer to programmatically create any additional windows your application requires. You can create new windows by constructing new `NativeWindow` objects and then opening them.

Before you can create a `NativeWindow` object, you first must create a `flash.display.NativeWindowInitOptions` object that the `NativeWindow` constructor uses to determine a handful of initial parameters such as the type of window, the chrome the

window should use, and so forth. Arguably the most important properties of a `NativeWindowInitOptions` object are the `type`, `systemChrome`, and `transparent` properties. These properties have dependencies on one another as well.

The `type` property has three possible values defined by three constants of the `flash.display.NativeWindowType` class: `STANDARD`, `UTILITY`, and `LIGHTWEIGHT`. The default `type` is `standard`, which means that the window uses full system chrome and shows up as a unique system window. (It shows up in the task bar for Windows or window menu for OS X.) Standard windows are most appropriate for opening things that are conceptually new instances, such as a new photo for editing in a photo-editing program. Utility windows have a thinner version of system chrome. Unlike standard windows, utility windows don't show up in the task bar or window menu. That makes utility windows best suited for content that's conceptually linked with the main window, such as tool palettes. Lightweight windows have no system chrome. Like utility windows, they don't show up in the task bar or window menu. Because lightweight windows don't have any system chrome, you must set the `systemChrome` property to `none` as well.

The `systemChrome` property determines the chrome that appears around the window. The possible values are defined by two constants of the `flash.display.NativeWindowSystemChrome` class: `STANDARD` and `NONE`. The standard chrome uses the system chrome for the operating system. That means that AIR windows using standard chrome will look just like other native applications running on the same computer. Setting the `systemChrome` property to `none` will remove any chrome from the window. (Note that the initial window is an exception to this rule, because it uses AIR chrome if the system chrome is configured to `none` in the descriptor file.) That means that windows initialized with `systemChrome` set to `none` won't have built-in mechanisms for maximizing, minimizing, restoring, closing, resizing, or moving. If you want to enable those behaviors on such a window, it's up to you to do that programmatically. (These topics are covered later in this chapter.)

The `transparent` property is a Boolean property that indicates whether or not the window can use alpha blending to allow transparency such that other windows can be seen beneath it. The default value for this property is `false`. Setting it to `true` enables alpha blending. Be aware that enabling transparency will use more system resources than would be used with a nontransparent window. Also be aware that, if you set transparent to `true`, you must also set the `systemChrome` to `none`. Unlike standard windows, a transparent window allows you to create nonrectangular shapes and fading effects.

NOTE By default, windows have a background color. Setting `transparent` to `true` will remove the background color, allowing for alpha blending. It also allows you to create irregularly shaped windows. See the section titled "Creating irregularly shaped windows" later in this chapter for more details.

You can also use the `minimizable`, `maximizable`, and `resizable` properties of a `NativeWindowInitOptions` object to specify whether or not the window will allow for

minimizing, maximizing, and resizing of the window. The default value for all these properties is `true`.

Once you've created a `NativeWindowInitOptions` object, you can construct a `NativeWindow` object by calling the constructor and passing the `NativeWindowInitOptions` object to the constructor, as shown in listing 2.1.

Listing 2.1 Creating a `NativeWindow` object

```
package {

   import flash.display.MovieClip;
   import flash.display.NativeWindow;
   import flash.display.NativeWindowInitOptions;
   import flash.display.NativeWindowType;

   public class Example extends MovieClip {

        public function Example() {
          var options:NativeWindowInitOptions =
          ➥new NativeWindowInitOptions();
          options.type = NativeWindowType.UTILITY;

          var window:NativeWindow = new NativeWindow(options);

          window.width = 200;
          window.height = 200;

       }

   }

}
```

1 Create the window options

2 Construct the new window

3 Set the initial width and height

In this example, we first create the window options and set the `type` and `chrome` values on that options object 1. Next we create the window itself, passing it the options 2. We also set the initial width and height to 200-by-200 3. See the "Adding content to windows" section for more information on how setting the width and height works.

We've successfully created a window, set options on the window, and even set the size of the window. However, the code up to this point won't actually display the window. We'll look at how to do that next.

OPENING WINDOWS

If you were to run the code in listing 2.1, you wouldn't see a new window appear. The reason is that, although you've constructed a new window, you haven't yet told the application to open it. You can open a window by calling the `activate()` method. Adding one line of code (see bold text in listing 2.2) to the code from listing 2.1 will launch a new 200-by-200 utility window.

Listing 2.2 Opening the new window

```
package {

   import flash.display.MovieClip;
   import flash.display.NativeWindow;
   import flash.display.NativeWindowInitOptions;
```

```
    import flash.display.NativeWindowType;

    public class Example extends MovieClip {

        public function Example() {

            var options:NativeWindowInitOptions =
            ➥new NativeWindowInitOptions();
            options.type = NativeWindowType.UTILITY;

            var window:NativeWindow = new NativeWindow(options);
            window.width = 200;
            window.height = 200;

            window.activate();

        }

    }

}
```

The new window that's created in this example is 200-by-200 pixels with a white background. But it doesn't have any other content yet. Most windows have some sort of content, and it's your responsibility to add it, as you'll see in the next section.

ADDING CONTENT TO WINDOWS

When you create a new window, it doesn't have any content other than a background (and even the background is absent if you've created a transparent window). It's your responsibility to add content to the window using the window's `stage` property.

You may be surprised to learn that `NativeWindow`, despite being in the `flash.display` package, isn't actually a display object. It doesn't inherit from `DisplayObject`, the base display type in ActionScript. Instead, windows *manage* display objects. A `NativeWindow` object has a `stage` property of type `flash.display.Stage`. The stage is a reference to the display object used as the container for the contents of the window. Because a `Stage` object is a display object container, it allows you to add and remove and manage content just as with any other display object container via the `addChild()`, `removeChild()`, and related methods.

Listing 2.3 uses the code from listing 2.2 as a starting point, and then adds a text field to the window. The new code is shown in bold.

Listing 2.3 Adding content to the window

```
package {

    import flash.display.MovieClip;
    import flash.display.NativeWindow;
    import flash.display.NativeWindowInitOptions;
    import flash.display.NativeWindowType;
    import flash.text.TextField;
    import flash.text.TextFieldAutoSize;

    public class Example extends MovieClip {

        public function Example() {
```

```
            var options:NativeWindowInitOptions =
            ➥new NativeWindowInitOptions();
            options.type = NativeWindowType.UTILITY;

            var window:NativeWindow = new NativeWindow(options);
            window.width = 200;
            window.height = 200;

            var textField:TextField = new TextField();
            textField.autoSize = TextFieldAutoSize.LEFT;
            textField.text = "New Window Content";

            window.stage.addChild(textField);

            window.activate();

        }

    }

}
```

1 Create new display object

2 Add content to the window

We change two things in this example. First we add a text field with the text New Window Content 1. In this example, we're using a text field, but you could also use any other type of display object. Then we add the text field to the window via its stage 2. We use the `addChild()` method to add the text field to the stage, which is the standard way to add content to a display object container. Figure 2.2 shows what the result of this code looks like (on Windows).

Figure 2.2 The new window with text scales its content, causing the text to appear differently than you might expect.

You'll probably notice that the text appears differently than you would have expected. That's because (perhaps unexpectedly) the stage of the new window is set to scale by default. That means that when you set the width and height of the window (as we have in this example), the content scales accordingly based on a default initial size for the window. In this case, the stage scaled larger considerably, and that causes the text to be larger than you might have expected

Using the `scaleMode` property of the stage, you can adjust that setting if appropriate. In this particular example, it would be better if the content didn't scale. As such, we can set the `scaleMode` property to the `StageScaleMode.NO_SCALE` constant, and it'll no longer scale. As soon as we do that and test the application, it's apparent that the `align` property of the stage needs to be set as well. In this case, it's best if the stage is always aligned to the top left. Listing 2.4 shows these additions in bold.

Listing 2.4 Adjusting the scale and alignment of the contents of a new window

```
package {

    import flash.display.MovieClip;
    import flash.display.NativeWindow;
```

```
import flash.display.NativeWindowInitOptions;
import flash.display.NativeWindowType;
import flash.text.TextField;
import flash.text.TextFieldAutoSize;
import flash.display.StageScaleMode;
import flash.display.StageAlign;

public class Example extends MovieClip {

   public function Example() {

      var options:NativeWindowInitOptions =
      ➥new NativeWindowInitOptions();
      options.type = NativeWindowType.UTILITY;

      var window:NativeWindow = new NativeWindow(options);
      window.width = 200;
      window.height = 200;

      var textField:TextField = new TextField();
      textField.autoSize = TextFieldAutoSize.LEFT;
      textField.text = "New Window Content";

      window.stage.scaleMode = StageScaleMode.NO_SCALE;
      window.stage.align = StageAlign.TOP_LEFT;

      window.stage.addChild(textField);

      window.activate();

   }

}
}
```

Figure 2.3 shows what this new window looks like.

Figure 2.3 By setting the stage's `scaleMode` and `align` properties, the content appears correctly in the new window.

NOTE If you test any of the preceding examples, you may discover that utility windows don't automatically close when you close the main application window. Even though the utility window isn't accessible from the task bar or window menu, it remains open until you close it. As long as the window is open, it may prevent you from testing your application again. Make sure to close the utility windows each time you close your main application window. For more information regarding how to manage utility windows, consult section 2.2.3.

Now that you've had a chance to see how to create and work with windows using ActionScript in a basic manner, we can look at how to create ActionScript classes for windows.

CREATING ACTIONSCRIPT CLASS–BASED WINDOWS

Thus far we've seen how to create new windows using the basic ActionScript concepts. As you create more and more sophisticated windows, it's generally advantageous to encapsulate the window code into ActionScript classes. Each window class should inherit from `NativeWindow`. The class can then contain all the code to add content, manage scale and alignment issues, and so on. Listing 2.5 shows an example of a simple window class.

Listing 2.5 Creating a basic window class

```
package {

   import flash.display.NativeWindow;
   import flash.display.NativeWindowType;
   import flash.display.NativeWindowInitOptions;

   public class ExampleWindow extends NativeWindow {

      public function ExampleWindow() {
         var options:NativeWindowInitOptions =
         ➥new NativeWindowInitOptions();

         options.type = NativeWindowType.UTILITY;

         super(options);

         width = 200;
         height = 200;

      }

   }

}
```

You can see that this window is responsible for setting its own chrome and type as well as its width and height. Note that the first thing it does is create a `NativeWindowInitOptions` object and then pass that in to the super constructor.

You can create an instance of this sort of window in much the same way you would any other `NativeWindow` instance: construct an instance and then call `activate()` to open it. Listing 2.6 shows what that code looks like.

Listing 2.6 Creating and opening an instance of ExampleWindow

```
package {

   import flash.display.MovieClip;

   public class Example extends MovieClip {

      public function Example() {

         var window:ExampleWindow = new ExampleWindow();
         window.width = 200;
         window.height = 200;
```

```
            window.activate();
        }
    }
}
```

We have just one more basic concept to discuss before moving on to new topics. So far you've seen how to create rectangular windows. Next we'll look at how to create irregularly shaped windows.

CREATING IRREGULARLY SHAPED WINDOWS

The ability to easily create irregularly shaped windows is a nice feature of AIR applications. Creating an irregularly shaped window is the same as creating a rectangular window except that you must turn off system chrome and set the window to be transparent. Then you can add a nonrectangular background to the window using the window's `stage` property. Listing 2.7 shows an example of this by modifying the code from listing 2.5.

Listing 2.7 Creating a nonrectangular window

```
package {

    import flash.display.NativeWindow;
    import flash.display.NativeWindowSystemChrome;
    import flash.display.NativeWindowType;
    import flash.display.NativeWindowInitOptions;
    import flash.display.Sprite;
    import flash.display.Stage;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    public class ExampleWindow extends NativeWindow {

        private var _background:Sprite;

        public function ExampleWindow() {
            var options:NativeWindowInitOptions =
            ➥new NativeWindowInitOptions();

            options.systemChrome = NativeWindowSystemChrome.NONE;
            options.type = NativeWindowType.LIGHTWEIGHT;

            options.transparent = true;

            super(options);

            _background = new Sprite();
            drawBackground(200, 200);
            stage.addChild(_background);

            width = 200;
            height = 200;

            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;
```

1 Set systemChrome to none

2 Make window transparent

3 Create background

```
        }

        private function drawBackground(newWidth:Number, newHeight:Number):
        ➥void {
            _background.graphics.clear();
            _background.graphics.lineStyle(0, 0, 0);
            _background.graphics.beginFill(0x0000FF, .5);
            _background.graphics.drawRoundRectComplex(0, 0, newWidth,
                                    newHeight, 20, 20, 20, 1);
            _background.graphics.beginFill(0xFFFFFF, .9);
            _background.graphics.drawRoundRectComplex(5, 5, newWidth - 10,
                                    newHeight - 10, 20,  20, 20, 1);
            _background.graphics.endFill();

        }

    }

}
```

There's a lot of code in this example, but it's not difficult to understand when we break it down. The first thing we do is make sure we set the `systemChrome` property of the options object to `none` (1). This removes any chrome from the window, which would otherwise force a rectangular border. Next we set the window to use transparent mode (2). This is important because normally the window has a solid rectangular background. To create a nonrectangular shape, we need to hide the background. Then we create a background display object, draw a nonrectangular shape in it, and add it to the stage (3). In this example, we're drawing a rounded-corner rectangle that is a subtle variation on the standard, square-cornered rectangular background.

Figure 2.4 shows what the result of this example looks like.

If you create irregularly shaped windows, it's your responsibility to add the necessary user interface elements and code for the behaviors that are usually provided automatically by the system chrome: closing, minimizing, maximizing, and moving. See section 2.2 for more information on how to do this.

Figure 2.4 An irregularly shaped window with transparency overlaps desktop icons and the main application window.

2.1.2 Flex application and windows

When you create AIR applications using Flex, the workflow is a little different when it comes to creating and managing windows. But the underlying essentials are the same as those used by Flash-based AIR applications using `NativeApplication` and `NativeWindow`. The difference is that, when using Flex, there are two Flex components that simplify the process of working with an application or windows programmatically. The `WindowedApplication` component is how you work with applications, and the `Window` component is how you work with windows.

CREATING AN APPLICATION

All Flex-based AIR applications must be compiled from application MXML documents that use `WindowedApplication` as the root element. That's why, when you create a new MXML application document in an AIR project in Flex Builder, you see the following stub code:

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

</mx:WindowedApplication>
```

Because you can only have one MXML application document per Flex project, it follows that you can only have one `WindowedApplication` object per application. The `WindowedApplication` component extends the `Application` component normally used by web-based Flex applications. Therefore, the properties and methods of `Application` are accessible to `WindowedApplication` as well. However, static properties aren't inherited by subclasses. Therefore the `Application.application` property, which references the main application object, isn't inherited. If you want to reference the one `WindowedApplication` instance (outside of the MXML document itself, within which you can simply reference it using `this`), you must use `Application.application`. The `Application.application` object is typed as `Application` rather than `WindowedApplication`. Therefore you must cast the object if you intend to reference it as a `WindowedApplication`:

```
var windowedApplication:WindowedApplication =
➥Application.application as WindowedApplication;
```

`WindowedApplication` instances have a bunch of properties and methods, many of which we'll look at in more detail later in this chapter. However, for the most part, the principal property that you need to know about to access core underlying values and behaviors is the `nativeApplication` property, which is a reference to the underlying `NativeApplication` object.

CREATING WINDOWS

When creating Flex-based AIR applications, all windows should be based on the `Window` component. Although it's possible to create windows directly using `NativeWindow`

(and although NativeWindow is still used behind the scenes), the Flex-specific Window component integrates well with the rest of the Flex framework and simplifies aspects of creating windows, as you'll see in the "Adding content to windows" section that follows shortly.

Creating a window using Flex is even simpler and more direct than creating a window using Flash. When working directly with NativeWindow objects, as you've learned, you have to first create a NativeWindowInitOptions object. The Window component in Flex hides that from you. You only need to create a new Window object (or an object from a subclass of Window), then set a few properties directly on that object if appropriate. For example, the code in listing 2.8 creates a new utility window.

Listing 2.8 Creating a new window using Flex

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute" creationComplete="creationCompleteHandler();">
    <mx:Script>
       <![CDATA[
          import mx.core.Window;

          private function creationCompleteHandler():void {
             var window:Window = new Window();

             window.width = 200;

             window.height = 200;

             window.type = NativeWindowType.UTILITY;
          }

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

Construct new window

Set initial dimensions

Make it a utility window

NOTE Just as WindowedApplication objects have nativeApplication properties that reference the underlying NativeApplication object, Window objects have nativeWindow properties that reference the underlying NativeWindow. It's worth noting as well that WindowedApplication objects also have a nativeWindow property that references the underlying NativeWindow object for the main window.

You may have noticed, assuming you tested the preceding code, that no window appears when running the code. As when working with NativeWindow objects directly, you need to explicitly open the window once you've created it.

OPENING WINDOWS

As we just saw, you still need to programmatically open a window once you've created it. For Window objects, you need to call the open() method to open the window. Listing 2.9 modifies the code from listing 2.8 by simply adding a call to open() in order to open a new window.

Listing 2.9 Opening a window is as simple as calling the `open()` method

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
 layout="absolute" creationComplete="creationCompleteHandler();">
   <mx:Script>
      <![CDATA[
         import mx.core.Window;

         private function creationCompleteHandler():void {
            var window:Window = new Window();

            window.width = 200;
            window.height = 200;

            window.type = NativeWindowType.UTILITY;

            window.open();
         }

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

In this example, the new window that appears is a 200-by-200 pixel window with the (Flex) default gray background and no other content. Next we'll look at how to add content to windows using Flex.

ADDING CONTENT TO WINDOWS

Adding content to windows in Flex is typically different from adding content to `NativeWindow` objects in Flash. When working with the latter, you must programmatically add content to the stage of the `NativeWindow` object after you've created it. When working with Flex-based windows, it's more common to simply create new MXML components based on `Window`, place the content in those components, and then open instances of those components as windows. Let's look at an example of this.

In the section on working with windows using `NativeWindow`, you saw an example in which you created a new window and then added text content to it. We'll now achieve a similar result, but this time using the Flex approach. Start by creating a new MXML component and call it SimpleTextWindow.mxml. The code for that component is shown in listing 2.10.

Listing 2.10 Creating a simple window component

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Window xmlns:mx="http://www.adobe.com/2006/mxml" width="200"
 height="200" type="utility">
   <mx:Label text="New Window Content" />
</mx:Window>
```

Note that this component uses `Window` as the root element. This is important. All components that you want to use as windows must extend `Window`. Also note that we're setting the `width` and `height` as well as the `type` of the component in the MXML document itself. You *could* set those properties in the ActionScript instead, but in this

case it's more sensible to set them in the window component itself if they should be consistent across all instances of the window.

Next we create an instance of the window in the main application MXML document, as shown in listing 2.11.

Listing 2.11 Creating an instance of the window component

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute" creationComplete="creationCompleteHandler();">
    <mx:Script>
       <![CDATA[

          private function creationCompleteHandler():void {
             var window:SimpleTextWindow = new SimpleTextWindow();
             window.open();
          }

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

Note that in this code we construct a new instance of the `SimpleTextWindow` component instead of creating a generic `Window` object. Note also that, because we set the `width`, `height`, and `type` properties in the MXML component itself, we don't need to set them when instantiating it. Figure 2.5 shows what the window looks like.

Figure 2.5 A simple text window created in Flex

That wraps up our discussion of basic, rectangular windows in Flex. Before we move on to an entirely different topic, we'll discuss how to create irregularly shaped windows using Flex.

CREATING IRREGULARLY SHAPED WINDOWS

You've already learned how to create irregularly shaped windows using ActionScript, and therefore it seems only fair to discuss how to do it using Flex. The basic concept is the same in ActionScript and Flex: create a window that has a transparent background and no system chrome. The steps to achieve this (set `systemChrome` to `none` and `transparent` to `true`) are similar in both cases. However, with Flex windows, there's a small wrench thrown in the works. Listing 2.12 shows a window component MXML document. The code sets the `systemChrome` property to `none` and the `transparent` property to `true`. It then uses ActionScript to draw a circle and add it to the display list.

Listing 2.12 Creating a transparent window in Flex with `systemChrome` set to `none`

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Window xmlns:mx="http://www.adobe.com/2006/mxml" systemChrome="none"
  type="lightweight" transparent="true" width="200" height="200"
  creationComplete="creationCompleteHandler();">
```

```
    <mx:Script>
       <![CDATA[

          private function creationCompleteHandler():void {

             var shape:Shape = new Shape();
             shape.graphics.lineStyle(0, 0, 0);
             shape.graphics.beginFill(0xFFFFFF, 1);
             shape.graphics.drawCircle(100, 100, 100);
             shape.graphics.endFill();

             rawChildren.addChild(shape);

          }

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

If you create an instance of this window, you'll discover that there's still chrome applied to the window. You can see what this looks like in figure 2.6.

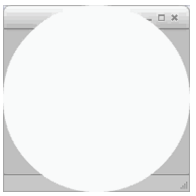


Figure 2.6 Flex window with `systemChrome` set to `none`

By default, Flex applies Flex chrome when `systemChrome` is set to `none`. If you want to remove all chrome, as is the goal in this example, you must take one additional step and set the `showFlexChrome` property of the window to `false`. All the other properties we've looked at for Flex windows thus far can be set on the instance using ActionScript or in the MXML using attributes. The `showFlexChrome` property can only be set using MXML, because it must be set before the window instantiates. The code in listing 2.13 shows this change to the code. With the addition of setting this one property, the Flex chrome is also removed and the window is now shaped like a circle.

Listing 2.13 Setting `showFlexChrome` to `false` to hide the Flex window chrome

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Window xmlns:mx="http://www.adobe.com/2006/mxml" showFlexChrome="false"
  systemChrome="none" type="lightweight" transparent="true" width="200"
  height="200" creationComplete="creationCompleteHandler();">
    <mx:Script>
       <![CDATA[

          private function creationCompleteHandler():void {

             var shape:Shape = new Shape();
             shape.graphics.lineStyle(0, 0, 0);
             shape.graphics.beginFill(0xFFFFFF, 1);
             shape.graphics.drawCircle(100, 100, 100);
             shape.graphics.endFill();

             rawChildren.addChild(shape);

          }

       ]]>
```

```
    </mx:Script>
</mx:Window>
```

Now that you've had a chance to learn how to create windows using lower-level ActionScript and Flex-specific techniques, we'll look at how to manage those windows.

2.2 Managing windows

Creating and opening windows is only the first step in effectively using windows. There are a handful of core skills that you must master to be proficient at working with windows. In the next few sections, we'll look at how to position, order, move, resize, and close windows.

2.2.1 Retrieving window references

Generally it's useful to be able to retrieve references to open windows for an application. Retrieving references is important for a variety of purposes, including (though not limited to) positioning windows, ordering windows, and communicating between windows.

The `NativeApplication` object for an AIR application keeps track of all the windows open for that application. The `mainWindow` property references the main application window (the initial window). The `openedWindows` property holds an array of all the windows currently opened in the application. You can also retrieve a reference to a `NativeWindow` object that corresponds to a `Stage` object by using a `Stage` object's `nativeWindow` property. We'll see how to use these properties in a variety of ways throughout the next few sections.

2.2.2 Positioning windows

Positioning windows is such an essential task that you must understand the basics before working with windows extensively. Otherwise, you're likely to open windows at seemingly random locations, sometimes hidden by other windows and sometimes overlapping existing windows in unintended ways.

POSITIONING NATIVEWINDOW OBJECTS

You can set the `x` and `y` properties of a `NativeWindow` object to set the x and y coordinates of the window in the desktop. When you position `NativeWindow` objects directly, this feat is achieved simply. You can merely set the `x` and `y` properties immediately after creating the object or at any time after that. However, when you create a `Window` component using Flex, you need to be aware that the underlying `NativeWindow` object for a `Window` component doesn't get created immediately when you construct a new `Window` object. We'll look at how to address this issue in the next section.

POSITIONING WINDOW OBJECTS

As already mentioned, the `NativeWindow` object that's associated with a `Window` component doesn't get instantiated immediately when you construct a `Window` component. Instead, you need to wait for that `Window` object to dispatch a `windowComplete` event before you can access the underlying `NativeWindow` object and set the `x` and `y`

properties. There are two common ways you can go about setting the x and y coordinates of a window: from the window that instantiates the new window or from within the new window itself.

When positioning the new window from the window that instantiates it, you want to register an event listener to listen for the `windowComplete` event, and then set the `x` and `y` properties of the new window's `nativeWindow` object when that event occurs. Listing 2.14 illustrates this. Listing 2.14 is a modification to listing 2.9, and changes are shown in bold.

Listing 2.14 Positioning a new window from the window that instantiates it

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
 layout="absolute" creationComplete="creationCompleteHandler();">
  <mx:Script>
    <![CDATA[
      import mx.events.AIREvent;

      private function creationCompleteHandler():void {
        var window:SimpleTextWindow = new SimpleTextWindow();

        window.addEventListener(
                AIREvent.WINDOW_COMPLETE,
                windowCompleteHandler);

        window.open();
      }

      private function windowCompleteHandler(event:AIREvent):void {
        event.target.stage.nativeWindow.x = 0;
        event.target.stage.nativeWindow.y = 0;
      }

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

1 Register listener

2 Set window coordinates

In this example, you can see that, right after we create the window instance, we register a method as a listener for the `windowComplete` event 1. The event handler method itself 2 sets the x and y properties of the window.

In the preceding example, you saw how to position a window from the window that opened it. If you prefer to position a window from within that new window itself, you can do that by simply listening for the `windowComplete` event within that component instead. We'll next look at an example of that. Listing 2.15 shows the `SimpleTextWindow` from listing 2.8 with a few changes in bold. In this example, you can assume that the application document still looks just as it did in listing 2.11.

Listing 2.15 Positioning a window from within the window component

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Window xmlns:mx="http://www.adobe.com/2006/mxml" width="200"
 height="200" type="utility"
```

```
windowComplete="windowCompleteHandler();">
  <mx:Script>
    <![CDATA[

      private function windowCompleteHandler():void {
        nativeWindow.x = 0;
        nativeWindow.y = 0;
      }

    ]]>
  </mx:Script>
  <mx:Label text="New Window Content" />
</mx:Window>
```

In these examples, we've looked at how to position windows when they first open. Of course, you can also use the same `x` and `y` properties of the `nativeWindow` object to position a window at any time, not just when it initially opens. Furthermore, you can use these same techniques to position the main application window when it initializes. Listing 2.16 shows one way to center the application on the screen.

Listing 2.16 Centering the application on the screen

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute"
  windowComplete="windowCompleteHandler();"
  creationComplete="creationCompleteHandler();">

  <mx:Script>
    <![CDATA[
      import mx.events.AIREvent;

      private function creationCompleteHandler():void {
        var window:SimpleTextWindow = new SimpleTextWindow();
        window.open();
      }

      private function windowCompleteHandler():void {
        nativeWindow.x = (Capabilities.screenResolutionX - width) / 2;
        nativeWindow.y = (Capabilities.screenResolutionY - height) / 2;
      }

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

1 Listen for windowComplete

2 Center window

In this example, the first thing we do is tell the window how to handle the `windowComplete` event 1. In this case, we tell the window to call a method named `windowCompleteHandler()` 2 that we've defined in the script block. The event handler method uses the screen resolution that it retrieves from the intrinsic `Capabilities` object in order to move the window to the center of the screen.

Next we'll look at how to do a slightly more sophisticated version of this same thing by working with a virtual desktop.

WORKING WITH A VIRTUAL DESKTOP

It's not uncommon for systems to have more than one monitor these days. If a user of your application has more than one monitor, it's a good idea for you to allow her to take advantage of the extra screen space when using your application. AIR applications allow users to drag windows anywhere on the desktop, including additional monitors. However, you should also consider extra monitors when programmatically placing windows. Frequently, users opt to place the main application window on their main monitor while utility windows go on a second monitor. But a user might just as well choose to move the main application window onto her second monitor. In any case, it's courteous for your AIR application to remember where a user has placed windows previously and put them there again when restarting the application. You can store user preferences such as window placement in a shared object (this is a basic ActionScript skill), in a local database (see chapter 5), or even in a text file (see chapter 3). The actual storage of the values is not particularly relevant to this chapter. What *is* relevant is how you can determine the correct values and how you can figure out whether a window is on one screen or another.

AIR applications treat the entire space of any and all monitors as one virtual desktop. Figure 2.7 illustrates this idea.

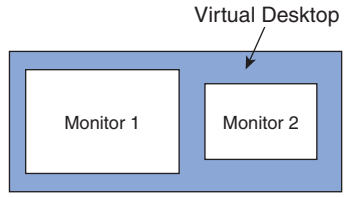


Figure 2.7 AIR applications treat all monitors as one virtual desktop.

In programmatic terms, each of these monitors has a representation as a `flash.display.Screen` object. A `Screen` object provides information about the size of the screen by way of a `bounds` property and a `visibleBounds` property. Because AIR applications can't change the resolution of monitors, the `bounds` and `visibleBounds` properties are read-only. Both these properties are `flash.geom.Rectangle` values with x and y values relative to the upper-left corner of the virtual desktop.

The `Screen` class also has two static properties that allow you to retrieve references to the `Screen` objects available to an AIR application. The `Screen.mainScreen` property returns a reference to the primary screen for the computer. The `Screen.screens` property returns an array of all the screens, the first of which is the same as the `mainScreen` reference.

The `Screen` class also has a static method called `getScreensForRectangle()`. This method allows you to retrieve an array of `Screen` objects over which a given `Rectangle` (relative to the virtual desktop) overlaps. The practical significance of this is that you can find out if a particular region (perhaps a window) exists on one screen or another, or both, or none.

The example in listing 2.17 illustrates a few of the virtual desktop and screen concepts by way of a crudely implemented window-snapping algorithm. In this example, the window snaps to the edge of the screen it's on if it's within 100 pixels of the edge. Much of this example is the same as listing 2.2. The changes are shown in bold.

Listing 2.17 Snapping a window to the edge of the screen

```
package {

   import flash.display.MovieClip;
   import flash.display.NativeWindow;
   import flash.display.NativeWindowInitOptions;
   import flash.display.NativeWindowType;
   import flash.display.NativeWindowSystemChrome;
   import flash.events.NativeWindowBoundsEvent;
   import flash.display.Screen;

   public class Example extends MovieClip {

      public function Example() {

         var options:NativeWindowInitOptions =
         ➥new NativeWindowInitOptions();
         options.type = NativeWindowType.UTILITY;

         var window:NativeWindow = new NativeWindow(options);
         window.width = 200;
         window.height = 200;

         window.activate();

         window.addEventListener(NativeWindowBoundsEvent.MOVE,
                                 moveHandler);

      }

      private function moveHandler(event:NativeWindowBoundsEvent):void {

         var window:NativeWindow = event.target as NativeWindow;

         var screens:Array =
         ➥Screen.getScreensForRectangle(window.bounds);
         var screen:Screen;

         if(screens.length == 1) {
            screen = screens[0];
         }
         else if(screens.length == 2) {
            screen = screens[1];
         }

         if(window.x < screen.bounds.x + 100) {
            window.x = screen.bounds.x;
         }
         else if(window.x > screen.bounds.x + screen.bounds.width -
         ➥window.width - 100) {
            window.x = screen.bounds.x + screen.bounds.width -
            ➥window.width;
         }
         if(window.y < screen.bounds.y + 100) {
            window.y = screen.bounds.y;
         }
         else if(window.y > screen.bounds.y + screen.bounds.height -
         ➥window.height - 100) {
            window.y = screen.bounds.y + screen.bounds.height -
            ➥window.height;
```

1 Listen for move event

2 Determine screen overlap

3 Select screen for snapping

4 Snap to edges within 100 pixels

```
            }
        }
    }
}
```

The first thing that's necessary in this example is to listen for the move event ❶ that the window dispatches when the user drags it. When the move event is handled, we tell the window to take action. The first thing it needs to do is retrieve an array of all the screens that the window currently overlaps using the `Screen.getScreensForRectangle()` method ❷. We then make a judgment: if the window overlaps just one screen, get a reference to that screen; otherwise use the second screen of the two that the window overlaps ❸. The remaining logic ❹ tests which edges of the screen the window is nearest and then snaps to them if it's within 100 pixels.

You can see that this example illustrates several key skills in working with screens, including determining which screen a window resides on and getting the bounds of a screen. Next we'll look at the important concept of closing windows.

2.2.3 Closing windows

Closing windows may seem like a simple task, but it's an important one with nuances you need to consider. In the next few sections, we'll examine how to deal with several window-closing issues, including reopening closed windows, closing all windows on application exit, and closing windows with no chrome.

REOPENING CLOSED WINDOWS

As we've been working through the concepts in this chapter, you've likely followed along with the example code and created quite a few windows. In doing so, you've probably noticed that, when chrome is applied to a window, the close button automatically gets wired up to allow the user to close the window. As convenient as that is, it's also potentially problematic because, in an AIR application, a window can't be reopened once it's been closed. While that may be the intended behavior in some cases, in many cases you intend to allow a user to reopen a window after he's closed it. Consider an example of a utility window that shows a color palette for a drawing program. The user should be able to show and hide the window. If you use the default close behavior for a window in such a case, AIR would prevent you from showing the window once the user closed it. Listing 2.18 shows an example that illustrates the problem.

Listing 2.18 You can't reopen a window that has been closed

```
package {

    import flash.display.NativeWindowType;
    import flash.display.NativeWindowInitOptions;
    import flash.display.NativeWindow;
    import flash.display.MovieClip;
    import flash.events.MouseEvent;

    public class WindowExample extends MovieClip {
```

```
        private var _window:NativeWindow;

        public function WindowExample() {

            var options:NativeWindowInitOptions =
            ➥new NativeWindowInitOptions();
            options.type = NativeWindowType.UTILITY;

            _window = new NativeWindow(options);
            _window.width = 200;
            _window.height = 200;

            _window.activate();

            stage.addEventListener(MouseEvent.CLICK, openWindow);
        }

        private function openWindow(event:MouseEvent):void {
            _window.activate();
        }

    }

}
```

In this example, the intent is that, any time the user clicks on the main window, the application should open the utility window by calling the `activate()` method. However, if you were to test this example, you'd discover that, once you've closed the utility window, any attempt to reopen it results in a runtime error stating that you can't open a window that's been closed.

What's the solution to this dilemma? The answer is simpler than you might think: rather than closing the window when the user clicks on the close button, you should instead set its visibility to false. That hides the window but allows it to be reopened. In order to achieve this, you must catch the window after the user clicks on the close button but before AIR has had a chance to take the default action and really close the window. You can do that by listening for a closing event. All windows dispatch an event of type `Event.CLOSING` an instant after the user clicks on the close button but just before the window actually closes. When handling the closing event, you must then do two things: set the `visible` property to `false` and cancel the default behavior. Canceling the default behavior is critical because otherwise AIR will still close the window. You can cancel the default behavior for cancelable events in ActionScript (of which the closing event is one) by calling the `preventDefault()` method on the event object. Listing 2.19 shows how you can listen for the closing event, toggle the visibility, and prevent the default behavior.

Listing 2.19 Allowing a user to hide and show a window

```
package {

    import flash.display.NativeWindowType;
    import flash.display.NativeWindowInitOptions;
    import flash.display.NativeWindow;
    import flash.display.MovieClip;
    import flash.events.MouseEvent;
```

```
import flash.events.Event;

public class WindowExample extends MovieClip {

   private var _window:NativeWindow;

   public function WindowExample() {

      var options:NativeWindowInitOptions =
      ➥new NativeWindowInitOptions();
      options.type = NativeWindowType.UTILITY;

      _window = new NativeWindow(options);
      _window.width = 200;
      _window.height = 200;

      _window.activate();

      stage.addEventListener(MouseEvent.CLICK, openWindow);

      _window.addEventListener(Event.CLOSING, closingHandler);
   }

   private function closingHandler(event:Event):void {
      _window.visible = false;
      event.preventDefault();
   }

   private function openWindow(event:MouseEvent):void {
      _window.activate();
   }

}
}
```

You can see that basic closing of windows is something you must consider when building applications if users need to be able to reopen windows that they've hidden.

CLOSING ALL WINDOWS ON APPLICATION EXIT

As you may have noticed throughout earlier examples in this chapter, windows you open from the initial application window don't automatically close when the initial application window closes. That is expected behavior if the additional windows are standard windows that show up in the task bar or window menu. However, if the additional windows are utility or lightweight windows, you likely want them to close when the main application window with which they're associated closes.

You can use the `openedWindows` property of a `NativeApplication` object to retrieve an array of references to all the `NativeWindow` objects for the application. By looping through the `openedWinodows` array, you can programmatically close all the utility and lightweight windows. Typically you'll want to do this when closing the application. You can detect when an application is closing by listening for the exiting event that the `NativeApplication` object dispatches. You can programmatically close a window by calling the `close()` method on the `NativeWindow` object. Listing 2.20 shows an example that programmatically closes all other windows when the main application window is closed.

Listing 2.20 Closing windows on application exit

```
package {

   import flash.display.MovieClip;
   import flash.display.NativeWindow;
   import flash.display.NativeWindowInitOptions;
   import flash.display.NativeWindowType;
   import flash.desktop.NativeApplication;
   import flash.events.Event;

   public class Example extends MovieClip {

      public function Example() {

         var options:NativeWindowInitOptions =
         ➥new NativeWindowInitOptions();
         options.type = NativeWindowType.UTILITY;

         var window:NativeWindow = new NativeWindow(options);
         window.width = 200;
         window.height = 200;

         window.activate();

         this.stage.nativeWindow.addEventListener(
         ➥Event.CLOSING, closingHandler);
      }

      private function closingHandler(event:Event):void {
         var windows:Array =
         ➥NativeApplication.nativeApplication.openedWindows;
         for(var i:Number = 0; i < windows.length; i++) {
            windows[i].close();
         }
      }

   }

}
```

(1) Listen for exiting event

(2) Close all open windows

In order to know when to close all the windows, we need to listen for the closing event that the `NativeApplication` instance dispatches (1). Then, when handling that event (2), we loop through the `openedWindows` array of the `NativeApplication` instance and call the `close()` method for all the windows.

We've looked at closing all open windows when the closing event occurs for the main window. If you don't close all the windows when you close an application, you can potentially cause problems for users. If a window is hidden but still technically open, the user won't have any way to close the window once she's exited the application. That would unnecessarily tie up system resources.

ADDING CUSTOM CLOSE MECHANISMS FOR WINDOWS

As you learned in the preceding section, you can close a window using the `close()` method. You can use that same method to close a window via a custom user interface element designed to allow the user to close a window. This is an important consideration when building windows that have no chrome (such as an irregularly shaped window) because such windows have no default close button.

2.2.4 Ordering windows

You can change the z-axis values of windows in a variety of ways. Typically, the window with focus appears in front of all other windows, and users generally expect to have control over the stacking order of windows on their desktop. However, there are legitimate reasons to control the order of windows programmatically, some of which are as follows:

- When creating new windows, you might want to intentionally open them in front of or behind existing windows.
- You might want to create a window that always stays in front of all other windows, even if it doesn't have focus. This is useful for windows that require user attention or that contain information that should always be available to the user even if he is using other applications.
- You might want to make sure utility windows are brought to the front of other applications when the corresponding AIR application receives focus.

There are a handful of methods and a property that control ordering. All of these methods and the property are available for `NativeWindow` objects in ActionScript as well as `Window` components in Flex. These methods and the property are as follows:

- `orderToFront()`—Move the window in front of all other windows in the same AIR application.
- `orderToBack()`—Move the window behind all other windows in the same AIR application.
- `orderInFrontOf(window:IWindow)`—Move the window in front of another window.
- `orderInBackOf(window:IWindow)`—Move the window behind another window.
- `alwaysInFront`—This window should always appear in front of all other windows on the desktop.

As you may have noticed in earlier examples, when you have a utility window running for an application and another application gets focus on your system, returning focus to the AIR application with the utility window doesn't return the utility window to the front of other running application windows on your system. That makes it easy to "lose" utility windows behind other applications. A useful feature to build into AIR applications with utility windows is automatically bringing those utility windows to the front along with the application's main window when it receives focus. When an application loses focus, the `NativeApplication` object dispatches a deactivate event; when the application receives focus, it dispatches an activate event. Therefore, if you listen for the activate event, cycle through all the opened windows, and move each opened window to the front, you'll make sure no utility windows get lost behind other applications. Listing 2.21 shows a simple example of this.

Listing 2.21 Moving utility windows to the front along with the main window

```
package {

   import flash.display.MovieClip;
   import flash.display.NativeWindow;
   import flash.display.NativeWindowInitOptions;
   import flash.display.NativeWindowType;
   import flash.desktop.NativeApplication;
   import flash.events.Event;
   import flash.text.TextField;
   import flash.text.TextFieldAutoSize;
   import flash.display.StageScaleMode;
   import flash.display.StageAlign;

   public class Example extends MovieClip {

      public function Example() {

         var options:NativeWindowInitOptions =
         ➥new NativeWindowInitOptions();
         options.type = NativeWindowType.UTILITY;

         var window:NativeWindow = new NativeWindow(options);
         window.width = 200;
         window.height = 200;

         var textField:TextField = new TextField();
         textField.autoSize = TextFieldAutoSize.LEFT;
         textField.text = "New Window Content";

         window.stage.scaleMode = StageScaleMode.NO_SCALE;
         window.stage.align = StageAlign.TOP_LEFT;

         window.stage.addChild(textField);

         window.activate();

         this.stage.nativeWindow.addEventListener(Event.ACTIVATE,
                                                  activateHandler);

      }

      private function activateHandler(event:Event):void {
         var windows:Array =
         ➥NativeApplication.nativeApplication.openedWindows;
         for(var i:Number = 0; i < windows.length; i++) {
            windows[i].orderToFront();
         }
      }

   }

}
```

If you were to test the preceding code, you'd see that, every time you bring the application window to the foreground, all of the utility windows also move to the foreground.

2.2.5 Moving and resizing windows

When a window has chrome around it, the user is able to move and resize the window through the standard user interface elements (dragging the title bar to move the window and dragging the borders to resize the window). When you don't display any system chrome for a window, the user won't have those options. Instead, if you want the user to be able to move or resize the window, you need to programmatically add that behavior. The good news is that AIR makes it simple to do so.

All `NativeWindow` objects have `startMove()` and `startResize()` methods. When you call these methods from an event handler for a `mouseDown` event, a `mouseUp` event will automatically stop the move or resize behavior—exactly the way a user would expect the feature to work.

The `startMove()` method doesn't require any parameters. You can simply call `startMove()` on the window when the user clicks on an object. The window will then move with the mouse until the user releases the mouse button. Listing 2.22 shows an example of this by modifying the `ExampleWindow` code from listing 2.7.

Listing 2.22 Enable dragging on a window using `startMove()`

```
package {

   import flash.display.NativeWindow;
   import flash.display.NativeWindowSystemChrome;
   import flash.display.NativeWindowType;
   import flash.display.NativeWindowInitOptions;
   import flash.display.Sprite;
   import flash.display.Stage;
   import flash.display.StageAlign;
   import flash.display.StageScaleMode;
   import flash.events.MouseEvent;
   import flash.events.Event;

   public class ExampleWindow extends NativeWindow {

      private var _background:Sprite;

      public function ExampleWindow() {
         var options:NativeWindowInitOptions =
         ➥new NativeWindowInitOptions();

         options.systemChrome = NativeWindowSystemChrome.NONE;
         options.type = NativeWindowType.LIGHTWEIGHT;
         options.transparent = true;

         super(options);
         _background = new Sprite();
         drawBackground(200, 200);
         stage.addChild(_background);

         width = 200;
         height = 200;

         stage.align = StageAlign.TOP_LEFT;
         stage.scaleMode = StageScaleMode.NO_SCALE;
```

```
            _background.addEventListener(MouseEvent.MOUSE_DOWN,
                                         startMoveWindow);

        }

        private function drawBackground(newWidth:Number, newHeight:Number):
        ➥void {
            _background.graphics.clear();
            _background.graphics.lineStyle(0, 0, 0);
            _background.graphics.beginFill(0x0000FF, .5);
            _background.graphics.drawRoundRectComplex(0, 0, newWidth,
                                    newHeight, 20, 20, 20, 1);
            _background.graphics.beginFill(0xFFFFFF, .9);
            _background.graphics.drawRoundRectComplex(5, 5, newWidth - 10,
                                  newHeight - 10, 20, 20, 20, 1);
            _background.graphics.endFill();

        }

        private function startMoveWindow(event:MouseEvent):void {
            startMove();
        }

    }

}
```

The `startResize()` method requires that you specify the side or corner of the window from which to resize. Valid values are constants of the `flash.display.NativeWindowResize` class: `TOP`, `BOTTOM`, `LEFT`, `RIGHT`, `TOP_LEFT`, `TOP_RIGHT`, `BOTTOM_LEFT`, and `BOTTOM_RIGHT`. For example, if you call `startResize()` with a value of `NativeWindowResize.BOTTOM_RIGHT`, the bottom-right corner of the window will move along with the mouse while the top-left corner stays fixed. You can call `startResize()` when the user clicks on an object. The window will start to resize along with the mouse movement until the user releases the mouse button. Listing 2.23 shows an example of resizing a window. The code is based on the window from listing 2.22.

Listing 2.23 Resizing a window

```
package {

    import flash.display.NativeWindow;
    import flash.display.NativeWindowSystemChrome;
    import flash.display.NativeWindowType;
    import flash.display.NativeWindowInitOptions;
    import flash.display.Sprite;
    import flash.display.Stage;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.MouseEvent;
    import flash.events.Event;
    import flash.display.NativeWindowResize;

    public class ExampleWindow extends NativeWindow {

        private var _background:Sprite;
```

```
private var _resizer:Sprite;

public function ExampleWindow() {
   var options:NativeWindowInitOptions =
   ➥new NativeWindowInitOptions();

   options.systemChrome = NativeWindowSystemChrome.NONE;
   options.type = NativeWindowType.LIGHTWEIGHT;
   options.transparent = true;

   super(options);
   _background = new Sprite();
   drawBackground(200, 200);
   stage.addChild(_background);

   width = 200;
   height = 200;

   stage.align = StageAlign.TOP_LEFT;
   stage.scaleMode = StageScaleMode.NO_SCALE;

   _background.addEventListener(MouseEvent.MOUSE_DOWN,
                                    startMoveWindow);

   _resizer = new Sprite();
   _resizer.graphics.lineStyle(0, 0, 0);
   _resizer.graphics.beginFill(0xCCCCCC, 1);
   _resizer.graphics.drawRect(0, 0, 10, 10);
   _resizer.graphics.endFill();
   _resizer.x = 180;
   _resizer.y = 180;
   stage.addChild(_resizer);

   _resizer.addEventListener(MouseEvent.MOUSE_DOWN,
                                   startResizeWindow);

   addEventListener("resizing", resizingHandler);

}

private function drawBackground(newWidth:Number, newHeight:Number):
➥void {
   _background.graphics.clear();
   _background.graphics.lineStyle(0, 0, 0);
   _background.graphics.beginFill(0x0000FF, .5);
   _background.graphics.drawRoundRectComplex(0, 0, newWidth,
                              newHeight, 20, 20, 20, 1);
   _background.graphics.beginFill(0xFFFFFF, .9);
   _background.graphics.drawRoundRectComplex(5, 5, newWidth - 10,
                           newHeight - 10, 20, 20, 20, 1);
   _background.graphics.endFill();

}

private function startMoveWindow(event:MouseEvent):void {
   startMove();
}

private function resizingHandler(event:Event):void {
   drawBackground(width, height);
   _resizer.x = width - 20;
```

```
            _resizer.y = height - 20;
        }

        private function startResizeWindow(event:MouseEvent):void {
            startResize(NativeWindowResize.BOTTOM_RIGHT);
        }

    }
}
```

You'll likely notice in this example that the window dispatches a resizing event as it resizes. You can use that event to redraw or rearrange the contents of the window appropriately.

We've now covered all the topics related to window behavior, and we're ready to move on to a more macroscopic view by looking at application-wide behavior. In the next section, we'll look at a variety of these topics, including detecting idleness and running an application in full-screen mode.

2.3 Managing applications

You've now learned all the basic window-specific information for building AIR applications. Next we'll look at application-level issues related to managing the application. We'll look at the following topics:

- Detecting idle users
- Launching AIR applications when the system starts
- Creating file associations
- Alerting the user
- Running applications in full-screen mode

AIR makes each of these tasks simple, as we'll see in the next few sections.

2.3.1 Detecting idleness

Often it's useful or important to detect when a user is no longer interacting with an application. For example, an instant messenger application can automatically set a user's status to away or idle if the user has stepped away from the computer. AIR applications automatically detect when a user hasn't interacted with the application for a specified amount of time. Once that threshold has been reached, the `NativeApplication` object dispatches a `userIdle` event. When the user returns to the application, the `NativeApplication` object dispatches a `userPresent` event. Both events are of type `flash.events.Event`, and the `Event.USER_IDLE` and `Event.USER_PRESENT` constants define the event names.

Default timeout value is 300 seconds. You can adjust this value by setting the `idleThreshold` property of the `NativeApplication` object. The value must be an integer representing the number of seconds without any detected activity before the `NativeApplication` object should dispatch a `userIdle` event.

In addition, you can request the amount of time since user interaction was last detected. The `timeSinceLastUserInput` property of the `NativeApplication` object will tell you how many seconds have elapsed since any mouse or keyboard input was received by the application.

2.3.2 Launching applications on startup

Typically, an AIR application runs only when the user double-clicks the application or a file of a type that's associated with it. However, you can also flag an AIR application to automatically launch when the user logs on to her computer system. Simply set the `startAtLogin` property of the `NativeApplication` object to `true`.

Note that, if `startAtLogin` is `true`, the application must be installed on the system. If you simply test this setting by testing your application from Flash or from Flex Builder, you'll receive a runtime error.

2.3.3 Setting file associations

By setting a file association, you allow the user to automatically launch the AIR application when he double-clicks a file of that type. In chapter 1, you learned how to define file types for an AIR application using the descriptor file. When you list file types in the descriptor, there are two possible options: if the file type isn't already associated with another application on the system, it's automatically registered with the AIR application; if the file type is registered with another application, no new association is made. In the second case, what happens is that the file type is then available for you to programmatically override the existing association. You can only do so at runtime using the methods listed in table 2.1.

When you want to create associations with file types that are commonly used by other applications, it's generally a best practice to request the user's permission to make the association. Consider, for example, if you associated .mp3 files with your AIR application using the descriptor file (hence not asking the user's permission). Because many users already have a preferred association for .mp3 files, you'd be likely to upset users if you changed that automatically without ever asking them. Therefore, although you can programmatically create associations for file types without first asking the user's permission, it's advisable that you do ask first.

Table 2.1 Methods for working with file associations

Method name	Description
`setAsDefaultApplication()`	Set the current AIR application as the default application for a particular file type.
`removeAsDefaultApplication()`	Remove the association between the current AIR application and a file type.
`isSetAsDefaultApplication()`	Find out if the current AIR application is already the default application for a file type.
`getDefaultApplication()`	Get the name of the default application for a file type.

To create associations for file types at runtime, use the `setAsDefaultApplication()` method of the `NativeApplication` object. The method requires that you pass it one parameter specifying the file extension for the type of file you want to associate with the AIR application. The parameter value should include just the file extension as a string, not including an initial dot. For example, the following code associates .mp3 files with the application that is currently running:

```
NativeApplication.nativeApplication.setAsDefaultApplication("mp3");
```

If you'd like to remove an association, you can simply call the `removeAsDefaultApplication()` method, passing it the extension of the file type for which you'd like to remove the association:

```
NativeApplication.nativeApplication.removeAsDefaultApplication("mp3");
```

The `isSetAsDefaultApplication()` method returns a Boolean value indicating whether the AIR application is the default application for a specified file extension:

```
var isDefault:Boolean = NativeApplication.nativeApplication.
➥isSetAsDefaultApplication("mp3");
```

You can also use the `getDefaultApplication()` method to retrieve the name of the application with which a file type is associated. The method requires one parameter specifying the file extension as a string:

```
var defaultApplication:String = NativeApplication.nativeApplication.
➥getDefaultApplication("mp3");
```

All of these methods will only work for file types that have been included in the descriptor file in the `fileTypes` section. Also note that these methods will only work for an AIR application once it's been installed. That means these methods won't work correctly when testing the application in Flash or Flex Builder.

2.3.4 Alerting the user

Occasionally an AIR application needs to notify the user that something has occurred requiring the user's attention, even though the application may be minimized or not have focus. Operating systems have standard ways of alerting users about these sorts of things. For example, on Windows the corresponding item in the task bar flashes, and on OS X the item bounces in the application dock. AIR allows you to alert the user in these standard ways with a `notifyUser()` method on a `NativeWindow` object.

The `notifyUser()` method requires that you pass it a parameter with one of the two constants of the `flash.desktop.NotificationType` class: `NotificationType.INFORMATIONAL` or `NotificationType.CRITICAL`. These two types correspond to the two types of notifications that are allowed by the operating system.

The following example (listing 2.24) illustrates notification. Every five seconds, the application tests to see if the main window is active (has focus). If not, it notifies the user.

Listing 2.24 Alerting the user if the main window isn't active

```
package {

   import flash.display.MovieClip;
   import flash.display.NativeWindow;
   import flash.desktop.NativeApplication;
   import flash.utils.Timer;
   import flash.events.TimerEvent;
   import flash.desktop.NotificationType;

   public class Example extends MovieClip {

      private var _timer:Timer;

      public function Example() {

         _timer = new Timer(5000);

         _timer.addEventListener(TimerEvent.TIMER, timerHandler);
         _timer.start();

      }

      private function timerHandler(event:TimerEvent):void {
         var mainWindow:NativeWindow =
➥NativeApplication.nativeApplication.openedWindows[0] as NativeWindow;
         if(!mainWindow.active) {
            mainWindow.notifyUser(NotificationType.INFORMATIONAL);
         }
      }

   }

}
```

As you can see, alerting or notifying a user that a window requires her attention is simple.

One sure-fire way to get a user's attention is to launch an application in full-screen mode. Continue to the next section, where we'll learn how to do just that.

2.3.5 Full-screen mode

You can launch application windows in full-screen mode using the `displayState` property of a window's `Stage` object. Set the `displayState` property to `StageDisplayState.FULL_SCREEN` as in this example in listing 2.25. Note that because these applications run in full-screen mode without the standard mechanisms to close the applications, you must close the application using the standard keyboard shortcuts for your operating system.

Listing 2.25 Opening a window in full-screen mode

```
package {

   import flash.display.MovieClip;
   import flash.text.TextField;
   import flash.display.StageDisplayState;
```

```
    public class Example extends MovieClip {

        public function Example() {

            stage.displayState = StageDisplayState.FULL_SCREEN;

            var textField:TextField = new TextField();
            textField.text = "Full screen example";
            addChild(textField);

        }

    }
}
```

This example uses a text field to illustrate a point. The default settings for a `Stage` object specify that the object should scale. Unless you want the content to scale, you should set the `scaleMode` property. In many cases, you'll also want to modify the `align` property. In this example, the text scales noticeably in most cases (depending on the starting dimensions of the window and the screen resolution). Listing 2.26 makes adjustments to the code, telling it not to scale the contents and to align them to the upper left.

Listing 2.26 Adjusting the scale mode and alignment of a full-screen window

```
package {

    import flash.display.MovieClip;
    import flash.text.TextField;
    import flash.display.StageDisplayState;
    import flash.display.StageScaleMode;
    import flash.display.StageAlign;

    public class Example extends MovieClip {

        public function Example() {

            stage.displayState = StageDisplayState.FULL_SCREEN;

            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            var textField:TextField = new TextField();
            textField.text = "Full screen example";
            addChild(textField);

        }

    }
}
```

We've now covered all the basic application-level management topics. Next we'll move on to a new subject altogether: working with menus.

2.4 Menus

AIR applications can use menus in a variety of ways. Menus can appear at the application or window level, they can appear on application icons, they can appear as context menus, and they can appear as pop-up menus. In the following sections, we'll look at how to create menus and then apply them in each of these ways.

2.4.1 Creating menus

All menus in AIR applications are of type `flash.display.NativeMenu`. You can create a new menu by constructing a new `NativeMenu` object. The constructor doesn't require any parameters:

```
var exampleMenu:NativeMenu = new NativeMenu();
```

That's all there is to creating menus. Next we'll look at adding elements to menus.

2.4.2 Adding elements to menus

Once you've created a menu, you can add elements to it. Elements can generally be one of two types: menu items or other menus (submenus).

Menu items are of type `flash.display.NativeMenuItem`. You can create a new `NativeMenuItem` using the constructor and passing it the label for the item. The label is what gets displayed in the menu:

```
var item:NativeMenuItem = new NativeMenuItem("Example Item");
```

Add a menu item to a menu using the `addItem()` method (or the `addItemAt()` method if you want to insert the item at an index other than last):

```
exampleMenu.addItem(item);
```

As already mentioned, you can add other menus to menus in order to create submenus. To do this, use the `addSubmenu()` (or `addSubmenuAt()`) method, passing it the menu to add:

```
exampleMenu.addItem(submenu);
```

You've now seen how to create menus and add elements to them. In order for a menu to be functional, though, you have to be able to detect when the user has selected an option. Read on and we'll discuss how to do that in the next section.

2.4.3 Listening for menu selections

When a user selects an item in a menu, that item (a `NativeMenuItem` object) dispatches a select event. It's up to you to add listeners to items such that your application can respond when the user selects the item.

The select event is of type `flash.events.Event`. The `target` property of the event object references the `NativeMenuItem` object that dispatched the event. Sometimes menu items need to have data associated with them in order for the application to take meaningful or contextual action when the user selects the item. To do that, you

can assign data of any type to the `data` property of the `NativeMenuItem` object. You can see examples of this in the example that follows later in listing 2.27.

There's just one more topic we need to cover before we start making menus appear in your application: special menu items such as checked items or separator lines. We'll cover that next.

2.4.4 Creating special menu items

Normal menu items appear simply as text. However, there are a few special types of menu items you can also use: checked items and separator items.

Checked items are useful when you have menu items that allow the user to toggle options on or off or to select among a list of items. You can display checks next to items by setting the `checked` property of the `NativeMenuItem` object.

Separator items are special types of menu items that serve simply to divide sections of a menu logically. Separator items are standard `NativeMenuItem` objects for which the `isSeparator` property is set to `true`. If the `isSeparator` property is `true` for an item, the item will appear as a line, and it won't be selectable.

We've covered enough theory. Now we need to see some practical examples of menus in use. In the next section, you'll learn how to use menus in a variety of ways.

2.4.5 Using menus

You can use menus in a handful of ways in AIR applications: application or window menus, dock application or system tray icon menus, context menus, and pop-up menus. In all cases the menus are instances of `NativeMenu`. The difference in each case is simply the way in which the menu is applied.

USING APPLICATION OR WINDOW MENUS

Application and window menus serve the same general purpose. Application menus are available only on OS X, and window menus are only available on Windows. Figure 2.8 shows an example of a window menu.

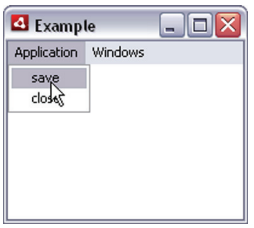


Figure 2.8 Window menus appear at the top of a window in the manner that's familiar to most computer users.

To apply a window menu, set the `menu` property of the `NativeWindow` object to which you want to apply the menu. To apply an application menu, set the `menu` property of the `NativeApplication` object. To determine whether the operating system supports one or the other, you can use the static `supportsMenu` property of the `NativeWindow` and `NativeApplication` classes. If `NativeWindow.supportsMenu` is `true`, you know that you can set the `menu` property of a `NativeWindow` object. If the `NativeApplication.supportsMenu` property is `true`, you can set the `menu` property of the `NativeApplication` object for the application. Because application menus and window menus typically are intended to accomplish the same thing, you'll usually want to use code such as the following after you've created the `NativeMenu` object. In the following code, you can assume that `customMenu` is a `NativeMenu` object you've already created:

```
if(NativeApplication.supportsMenu) {
   NativeApplication.nativeApplication.menu = customMenu;
}
else if(NativeWindow.supportsMenu) {
   NativeApplication.nativeApplication.openedWindows[0].menu = customMenu;
}
```

Listing 2.27 shows a complete, working example of a window or application menu that uses many of the concepts discussed through this section. This example creates a menu that allows the user to save (nonfunctional) or close the application, open a new window, and then toggle focus between opened windows.

Listing 2.27 Creating application or window menus

```
package {

   import flash.display.MovieClip;
   import flash.display.NativeWindow;
   import flash.display.NativeWindowInitOptions;
   import flash.display.NativeWindowType;
   import flash.desktop.NativeApplication;
   import flash.display.NativeMenu;
   import flash.display.NativeMenuItem;
   import flash.events.Event;

   public class Example extends MovieClip {

      private var _windowsMenu:NativeMenu;

      public function Example() {

         var mainMenu:NativeMenu = new NativeMenu();

         var applicationMenu:NativeMenu = new NativeMenu();

         var save:NativeMenuItem = new NativeMenuItem("save");
         var close:NativeMenuItem = new NativeMenuItem("close");

         close.addEventListener(Event.SELECT, selectHandler);

         applicationMenu.addItem(save);
         applicationMenu.addItem(close);

         _windowsMenu = new NativeMenu();

         var newWindow:NativeMenuItem =
         ➥new NativeMenuItem("new window");
            newWindow.addEventListener(Event.SELECT, selectHandler);

            var line:NativeMenuItem = new NativeMenuItem("", true);

         _windowsMenu.addItem(newWindow);
         _windowsMenu.addItem(line);

         mainMenu.addSubmenu(applicationMenu, "Application");
         mainMenu.addSubmenu(_windowsMenu, "Windows");

         var mainWindow:NativeWindow =
NativeApplication.nativeApplication.openedWindows[0] as NativeWindow;

         if(NativeApplication.supportsMenu) {
```

1 Create main menu

Create applications menu 2

3 Listen for select event

4 Add menu items

5 Create windows menu

Listen for select events 6

Create separator item 7

8 Add submenus

Reference main window 9

```
            NativeApplication.nativeApplication.menu = mainMenu;
        }
        else if(NativeWindow.supportsMenu) {
            mainWindow.menu = mainMenu;
        }

        mainWindow.addEventListener(Event.CLOSE, closeAll);

    }

    private function closeAll(event:Event = null):void {
        var windows:Array =
        ➥NativeApplication.nativeApplication.openedWindows;
        for(var i:Number = 0; i < windows.length; i++) {
            windows[i].removeEventListener(Event.CLOSE, closeHandler);
            windows[i].close();
        }
    }

    private function selectHandler(event:Event):void {
        if(event.target.label == "close") {
            closeAll();
        }
        else if(event.target.label == "new window") {

            var windowTitle:String = "Window " +
NativeApplication.nativeApplication.openedWindows.length;

            var options:NativeWindowInitOptions =
new NativeWindowInitOptions();
            options.type = NativeWindowType.UTILITY;
            var window:NativeWindow = new NativeWindow(options);
            window.width = 200;
            window.height = 200;
            window.title = windowTitle;

            window.addEventListener(Event.ACTIVATE, activateHandler);
            window.addEventListener(Event.CLOSE, closeHandler);

            var menuItem:NativeMenuItem = new NativeMenuItem();
            menuItem.label = windowTitle;
            menuItem.data = window;
            menuItem.addEventListener(Event.SELECT,
                                      selectWindowHandler);

            _windowsMenu.addItem(menuItem);

            window.activate();
        }
    }

    private function selectWindowHandler(event:Event):void {
        event.target.data.activate();
    }

    private function activateHandler(event:Event):void {
        var item:NativeMenuItem;
        for(var i:Number = _windowsMenu.numItems - 1; i >= 0; i--) {
            item = _windowsMenu.getItemAt(i);
            item.checked = (item.data == event.target);
```

Apply menu 10

11 Listen for close

12 Close all opened windows

Remove listener to prevent error 13

14 Close all windows

15 Open new window

16 Create unique window title

Create new utility window 17

Handle events 18

Create window menu item 19

20 Add item to menu

Activate corresponding window 21

22 Update checked state in menu

```
            }
        }

        private function closeHandler(event:Event):void {
            var item:NativeMenuItem;
            for(var i:Number = 0; i < _windowsMenu.numItems; i++) {
                item = _windowsMenu.getItemAt(i);
                if(item.data == event.target) {
                    _windowsMenu.removeItem(item);
                    break;
                }
            }
        }
    }

}
```

23 **Remove corresponding menu item**

The preceding is a long example. We'll break it down and look at each piece. The first thing we do is create a new menu 1, because that's essential to everything that follows. We'll call this the main menu, because it's the container for all the rest of the menus that we'll add to it. Next we create another menu and two items 2. This menu will be a submenu of the main menu. We add an event listener to one of the items 3 in order to be able to respond when the user selects that menu item. We've created the menu items, but we haven't yet added them to the menu. We do that next 4 using the `addItem()` method. Our main menu is going to have one more submenu, and we create that next 5 along with a menu option. As with the application menu's close option, we add a listener to the new window option 6. The windows menu is a dynamic menu because, as the application runs, we'll add more options to it each time the user opens a window. In order to differentiate between the static and dynamic items in the windows menu, we want a separator line. We can create that line by creating a new item with the second parameter set to `true` 7. With the two submenus created, we can add them to the main menu 8. Now that we've created the menu, we need only to apply it either as a window or an application menu. We get a reference to the main window 9, then test what type of menu the system supports and assign the menu accordingly 10. We also listen for a close event in order to close all the open windows when the main window closes 11.

The `closeAll()` method 12 is a typical method for what it does. It loops through all the open windows and calls the `close()` method for each. The one distinction in this example is that, in addition to calling the `close()` method, `closeAll()`also removes event listeners for the close event from each window 13. This is important because, as you'll see, when we create new windows in this example, we listen for the close event. If we didn't remove the listener before closing the window, we'd create an infinite loop.

The `selectHandler()` method is the event handler method we're using for the menu's options. We first determine what option the user selected. If he selected the close option 14, we call the `closeAll()` method. Otherwise, if the user selected the new window option 15, we run the code to open a new window. To create a new window, we first create a unique title for the window 16 by utilizing the count of existing

windows, making names such as Window 1, Window 2, and so on. Then we create a new utility window that is 200-by-200 pixels (17). We need to take action when the user closes or activates the window, and therefore we register listeners for the corresponding events (18). Each window should have a menu item in the windows menu. We add the menu item (19) using the window title as the item name, and we store a reference to the window in the menu item's `data` property. The reference is important because it allows the application to activate the window when the user selects it from the menu. Then we simply add the new item to the windows menu (20).

There are three events that we still need to handle: selecting a menu item corresponding to a window, activating a window, and closing a window. Selecting a window's menu item results in activating the window via the menu item's `data` property (21). Activating a window (22) results in looping through the windows menu items and updating the checked state of each, such that only the activated window's menu item is checked. Closing a window (23) requires looping through all the menu items to remove the menu item that corresponds to the window.

When you test this application, you have the option to add new windows by selecting the new window item from the windows menu. When you add new windows, you'll see new items added to the windows menu. Selecting windows changes the selected window item in the menu. Closing windows removes the menu items to which they correspond.

USING ICON MENUS

Icon menus are the menus that you can access from the system tray icon (Windows) or dock application icon (OS X) for the application. You can configure and control the icon menu using the `icon.menu` property of the `NativeApplication` object.

```
NativeApplication.nativeApplication.icon.menu = customMenu;
```

This is all that's necessary to customize the icon menu.

With window, application, and icon menus, we've seen how to use menus that can appear external to your application's content. Next we'll look at ways to use menus that appear within your application's content.

USING CONTEXT MENUS

You can display context menus for any display object in a Flash- or Flex-based AIR application. The mechanism for adding context menus to display objects in AIR applications is exactly the same as for web applications. You need to assign the menu to the `contextMenu` property of the display object. This is a standard Flash or Flex skill, even for web-based application development. However, when building AIR applications, there are two important differences:

- Context menus for AIR applications are instances of `NativeMenu`, whereas context menus for web applications are instances of `ContextMenu`.
- Context menus for AIR applications are system-level, meaning they can appear in front of and outside the boundaries of the application window.

Listing 2.28 illustrates a simple context menu.

Listing 2.28 Adding a context menu

```
package {

   import flash.display.MovieClip;
   import flash.display.NativeMenu;
   import flash.display.NativeMenuItem;
   import flash.display.Sprite;

   public class Example extends MovieClip {

      public function Example() {

         var rectangle:Sprite = new Sprite();
         rectangle.graphics.lineStyle(0, 0, 1);
         rectangle.graphics.beginFill(0, 1);
         rectangle.graphics.drawRect(0, 0, 100, 100);
         rectangle.graphics.endFill();
         addChild(rectangle);

         var menu:NativeMenu = new NativeMenu();
         var item:NativeMenuItem = new NativeMenuItem("copy");
         menu.addItem(item);

         rectangle.contextMenu = menu;
      }
   }

}
```

Figure 2.9 shows the result of this example.

There's just one more way to use menus with AIR applications. Next we'll look at using menus as pop-up menus.

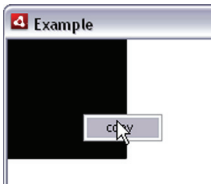


Figure 2.9 AIR context menus can appear outside the boundaries of the application window.

POP-UP MENUS

You can display pop-up menus at any time programmatically. Often you'll want to open pop-up menus when the user clicks the mouse or presses a key. Whatever you use as the trigger to open the menu, the code you use to actually display the menu is the same. You simply need to call the `display()` method of a `NativeMenu` object. The `display()` method requires that you specify three parameters: the `Stage` object on which to display the menu, and the x and y coordinates at which to display the menu. Listing 2.29 illustrates a simple pop-up menu.

Listing 2.29 Adding a pop-up menu

```
package {

   import flash.display.MovieClip;
   import flash.display.NativeMenu;
   import flash.display.NativeMenuItem;
   import flash.display.Sprite;
   import flash.events.MouseEvent;

   public class Example extends MovieClip {
```

```
        private var _menu:NativeMenu;

        public function Example() {

            _menu = new NativeMenu();
            var item:NativeMenuItem = new NativeMenuItem("a");
            _menu.addItem(item);
            item = new NativeMenuItem("b");
            _menu.addItem(item);
            item = new NativeMenuItem("c");
            _menu.addItem(item);
            item = new NativeMenuItem("d");
            _menu.addItem(item);

            var rectangle:Sprite = new Sprite();
            rectangle.graphics.lineStyle(0, 0, 1);
            rectangle.graphics.beginFill(0, 1);
            rectangle.graphics.drawRect(0, 0, 100, 100);
            rectangle.graphics.endFill();
            addChild(rectangle);

            rectangle.addEventListener(MouseEvent.MOUSE_DOWN, showMenu);
        }

        private function showMenu(event:MouseEvent):void {
            _menu.display(stage, mouseX, mouseY);
        }
    }
}
```

1 Create new menu

2 Create rectangle

3 Listen for mouseDown event

4 Display pop-up menu

In this example, we first create a new menu and add four options to it 1. Then we create a display object 2 that we'll use to launch the pop-up menu. We'll launch the pop-up menu when the user clicks the rectangle. Therefore we register an event listener for the `mouseDown` event 3. When the user clicks the rectangle, we display the pop-up menu at the point where the user is clicking 4.

You've now seen all the ways you can use menus in AIR applications: window menus, application menus, icon menus, context menus, and pop-up menus. Not only have we wrapped up our discussion of menus, but we've also covered all of the theoretical material in this chapter. Next we'll start applying some of this knowledge in building a sample application.

2.5 Starting the AirTube application

Throughout this book, we'll use a variety of smaller example applications. However, we'll also have one central, larger example application that we'll revisit and add to as we cover more and more topics. We're calling the application AirTube because it uses Adobe AIR to provide access to the popular YouTube video service. In this chapter, we'll build the foundation for the rest of the application by creating the windows that the application will use. Furthermore, we'll create some of the other foundational classes and components used by the application.

2.5.1 Overview of AirTube

AirTube uses the public YouTube developer API to create an AIR application that allows users to search the YouTube catalog by keywords, play videos, and store videos for offline playback. AirTube highlights the following features:

- Desktop application with multiple windows
- Search YouTube videos by keyword
- Play back videos
- View YouTube page for videos
- Save videos locally
- Search and play back local videos
- Detect network availability
- Save shortcuts to videos on the desktop

Before we get into the details of how to build the application, we'll first look at two screenshots from the completed application. In figure 2.10, you can see the main screen. The main screen allows users to search for videos using keywords/tags. The results are displayed in a two-column tile list.

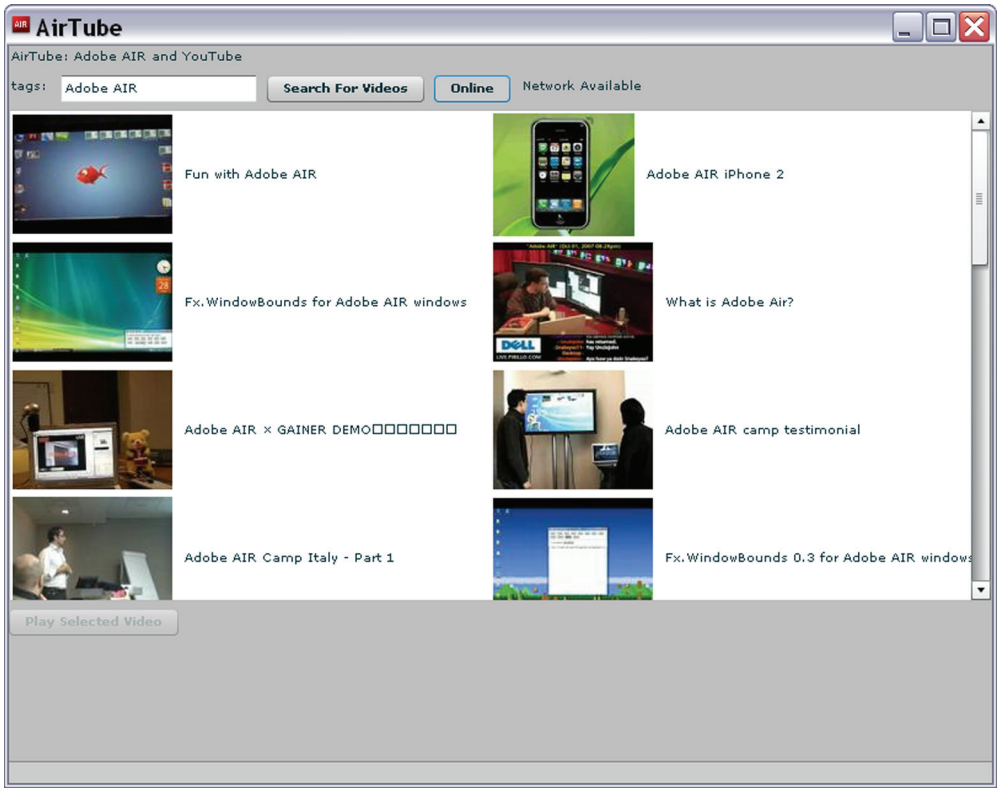


Figure 2.10 The main screen of the AirTube application allows for searching and browsing through videos.

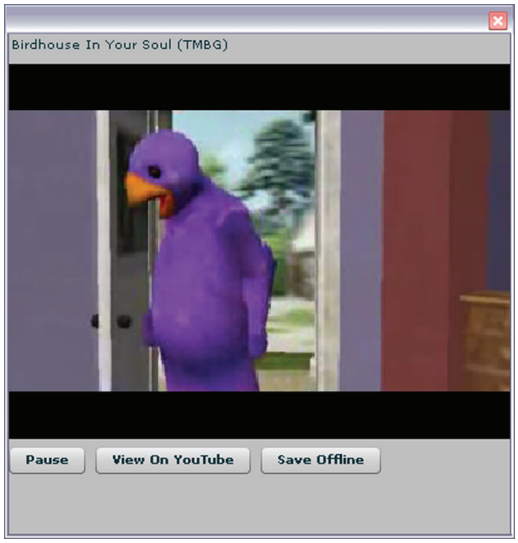


Figure 2.11 The video screen allows you to view a video, go to the YouTube page, or save the video locally.

Next we look at the video window. Figure 2.11 shows the video window, which not only plays back the selected video, but also allows the user to click through to the YouTube page for the video or to download the video for offline viewing.

In the next sections, we'll walk through the first few steps to begin building the application.

2.5.2 Getting started

Before we can start writing code, there are a few preliminary steps you'll need to take. They are as follows:

- Sign up for a YouTube developer API key
- Download two ActionScript libraries
- Configure a new AIR project

This application relies on the YouTube service. (See www.youtube.com/dev.) This is a free public service that allows developers to build applications that search YouTube's video library and play back the videos. Although the service is free, you do need an account to be able to access it.

1. If you don't have one already, you can sign up for a free YouTube account at www.youtube.com/signup. Simply fill out the form and click Sign Up.
2. Once you've created an account and logged in, you should go to your account page at www.youtube.com/my_account.
3. From your account page, click on the Developer Profile option, or you can go directly to www.youtube.com/my_profile_dev.
4. Request a new developer ID. This is the key that you'll need to access the YouTube service.

Now that you have the necessary YouTube account and key, you'll next need to get set up with the necessary ActionScript libraries. Although you could write your own code in ActionScript to work with the YouTube service directly, it's simpler to leverage existing libraries that are built expressly for that purpose. You'll need to download the as3youtubelib library as well as the as3corelib library. The official sites for these libraries are code.google.com/p/as3youtubelib and code.google.com/p/as3corelib, respectively. But to make sure you're working with the same version of the libraries that we use in this book, you can download those versions from this book's official site at www.manning.com/lott.

The next step is to create a new AIR project for the AirTube application. There's nothing unusual about this project, so you can create a new project in the way you normally would, whether using Flex Builder or configuring the project manually. Once you've created the project, add the as3youtubelib and as3corelib libraries. If you've downloaded the libraries from this book's site, you'll be downloading the source code, and the easiest way to add the code to your project is to unzip the code to the project's source directory.

That's all you need to do to configure your project. Now we can get started building the application.

2.5.3 Building the data model

The AirTube application centers around a data model locator that we'll call `ApplicationData`. The `ApplicationData` class uses the Singleton design pattern to ensure there's only one globally accessible instance of the class throughout the application. We use the `ApplicationData` instance to store all the information about the application state: is the application online or offline, are there any video results to a query, is a video currently downloading, and so forth. All that information gets stored in `ApplicationData`.

We're going to build `ApplicationData` (or at least the start of it) in just a minute. But first we have to create another class that we'll use as part of the application's data model. Because AirTube is essentially a video searching and viewing application, the one model class we're going to build is a video model class that we'll call `AirTubeVideo`. The `AirTubeVideo` class is a wrapper for the `Video` class from the as3youtube library. The as3youtube `Video` class is how all the data from the API calls gets serialized when it's returned, and contains information such as video title, ID, thumbnail image, and so on. We're creating a wrapper class (`AirTubeVideo`) for modeling video data because, in addition to the information returned by the YouTube API, we're also going to want to store a few extra pieces of data about each video, including the URL for the .flv file and whether we've stored the particular video locally. Follow these steps to create the data model for the AirTube application:

1. Create a new ActionScript class document and save it to com/manning/airtube/data/AirTubeVideo.as relative to your project's source directory.

2 Add the code from listing 2.30 to the `AirTubeVideo` class. As you can see, `AirTubeVideo` requires a `Video` parameter when constructing a new instance. The class has an accessor (getter) method for the `Video` object as well as two additional pieces of data: the URL for the .flv file and the offline status of the video (whether or not it's been downloaded locally).

Listing 2.30 The `AirTubeVideo` class

```
package com.manning.airtube.data {

   import com.adobe.webapis.youtube.Video;

   import flash.events.Event;
   import flash.events.EventDispatcher;

   public class AirTubeVideo extends EventDispatcher {

      private var _video:Video;
      private var _flvUrl:String;
      private var _offline:Boolean;

      public function get video():Video {
         return _video;
      }

      [Bindable(event="flvUrlChanged")]
      public function get flvUrl():String {
         return _flvUrl;
      }

      public function set flvUrl(value:String):void {
         _flvUrl = value;
         dispatchEvent(new Event("flvUrlChanged"));
      }

      [Bindable(event="offlineChanged")]
      public function set offline(value:Boolean):void {
         _offline = value;
         dispatchEvent(new Event("offlineChanged"));
      }

      public function get offline():Boolean {
         return _offline;
      }

      public function AirTubeVideo(value:Video) {
         _video = value;
      }

   }
}
```

1 **Enable Flex data binding**

In the code, you can see that we're using a `[Bindable]` metadata tag 1 that's used by Flex to enable data binding. We use the same convention throughout the book for the event names and `[Bindable]` metadata tags: the event name is the name of the getter/setter plus `Changed`. For example, in this case, the getter/setter is named `flvUrl` and the event is therefore `flvUrlChanged`.

3 Create a new ActionScript class document and save it to com/manning/airtube/data/ApplicationData.as relative to your project's source directory.

4 Add the code from listing 2.31 to the `ApplicationData` class. The `ApplicationData` class uses the Singleton pattern, which is why it has a static `_instance` property and an accessor method (`getInstance()`) to retrieve the one instance of the class. Otherwise, `ApplicationData` only has two pieces of data at this time: an array of videos and a reference to a currently selected video.

Listing 2.31 The `ApplicationData` class

```
package com.manning.airtube.data {

   import flash.events.Event;
   import flash.events.EventDispatcher;

   public class ApplicationData extends EventDispatcher {

      static private var _instance:ApplicationData;

      private var _videos:Array;
      private var _currentVideo:AirTubeVideo;

      [Bindable(event="videosChanged")]
      public function set videos(value:Array):void {
         _videos = value;
         dispatchEvent(new Event("videosChanged"));
      }

      public function get videos():Array {
         return _videos;
      }

      [Bindable(event="currentVideoChanged")]
      public function set currentVideo(value:AirTubeVideo):void {
         _currentVideo = value;
         dispatchEvent(new Event("currentVideoChanged"));
      }

      public function get currentVideo():AirTubeVideo {
         return _currentVideo;
      }

      public function ApplicationData() {

      }

      static public function getInstance():ApplicationData {
         if(_instance == null) {
            _instance = new ApplicationData();
         }
         return _instance;
      }

   }
}
```

1 Managed instance of class

2 Accessor to managed instance

If you're unfamiliar with the Singleton pattern, there are two key things to look at in this code. First, there's a static property with the same type as the class itself

(1). We use this property to store the one instance of the class. Next, the `getInstance()` method (2) is a public static method that returns a reference to the one instance of the class. You'll see this same pattern used throughout several other classes in the AirTube application.

That's all that's necessary to build the basics of the data model for the AirTube application. Next we'll start building the service that the AirTube uses.

2.5.4 Building the AirTube service

The AirTube application is built primarily around the YouTube developer API. We'll write a class called `AirTubeService` that acts as a proxy to the YouTube API. We'll also add other functionality into the service class over time. The following steps walk you through building the initial stages of the service class:

1. Open a new ActionScript class document and save it to com/manning/airtube/services/AirTubeService.as relative to your AirTube project's source directory.
2. Add the code from listing 2.32 to the `AirTubeService` class. You'll notice that `AirTubeService` also uses the Singleton design pattern.

Listing 2.32 The `AirTubeService` class

```
package com.manning.airtube.services {

   import com.adobe.webapis.youtube.YouTubeService;
   import com.adobe.webapis.youtube.events.YouTubeServiceEvent;
   import com.manning.airtube.data.AirTubeVideo;
   import com.manning.airtube.data.ApplicationData;

   import flash.events.Event;

   public class AirTubeService {

      static private var _instance:AirTubeService;

      public function AirTubeService() {
      }

      static public function getInstance():AirTubeService {
         if(_instance == null) {
            _instance = new AirTubeService();
         }
         return _instance;
      }
   }
}
```

3. Add a `_proxied` property that references an instance of the `YouTubeService` class from the as3youtube library. This instance allows `AirTubeService` to make calls to methods of `YouTubeService`. Listing 2.33 shows `AirTubeService` with the changes in bold.

Listing 2.33 The `AirTubeService` class with the proxied service requests

```
package com.manning.airtube.services {

   import com.adobe.webapis.youtube.YouTubeService;
   import com.adobe.webapis.youtube.events.YouTubeServiceEvent;
   import com.manning.airtube.data.AirTubeVideo;
   import com.manning.airtube.data.ApplicationData;

   import flash.events.Event;

   public class AirTubeService {

      static private var _instance:AirTubeService;

      private var _proxied:YouTubeService;

      public function set key(value:String):void {

         _proxied.apiKey = value;
      }

      public function AirTubeService() {

         _proxied = new YouTubeService();
         _proxied.addEventListener(
➥YouTubeServiceEvent.VIDEOS_LIST_BY_TAG, getVideosByTagsResultHandler);
      }

 static public function getInstance():AirTubeService {
         if(_instance == null) {
            _instance = new AirTubeService();
         }
         return _instance;
      }

      public function getVideosByTags(tags:String):void {
         if(_proxied.apiKey.length == 0) {
            throw Error("YouTube API key not set");
         }
         _proxied.videos.listByTag(tags);
      }

      private function getVideosByTagsResultHandler(
      ➥event:YouTubeServiceEvent):void {
         var videos:Array = event.data.videoList as Array;
         for(var i:Number = 0; i < videos.length; i++) {
            videos[i] = new AirTubeVideo(videos[i]);
         }
         ApplicationData.getInstance().videos = videos;
      }

   }
}
```

1 Reference to as3youtube service

Set developer key

2 Create service and listen to event

3 Search for videos

Video search response handler

4 Loop through results

Update data model 5

The `_proxied` property 1 is what we'll use to store a reference to an instance of the `YouTubeService` class from the as3youtube library. We'll use the instance to make calls to the YouTube service. You can see that we create an instance of the service in the constructor 2.

The YouTube service requires a developer key that you created in the getting started section. The YouTubeService class requires that you pass it the developer key via a property called `apiKey`. We're creating a setter for the AirTubeService that allows you to pass along the developer key to the YouTubeService instance.

The `getVideosByTags()` method (3) makes the request to retrieve videos that contain one or more of the tags/keywords specified. This method merely makes a call to the `listByTag()` method that's available from the YouTube service via the `_proxied` instance. The only thing we're doing other than relaying the request is ensuring that the developer key is defined. If the key isn't yet defined, the service won't work. Therefore, we throw an error if the key isn't defined.

The `getVideosByTagsResultHandler()` method is the event handler when a response is returned from the `YouTubeService`'s `listByTag()` method. This is where we'll take the result set, transform it into usable data, then assign that to the `ApplicationData` instance. The data is returned as an array of `Video` objects (from the as3youtube library). The array is stored in a `data.videoList` property of the `event` object. We want to loop through all the results and wrap them in `AirTubeVideo` objects (4). Once the videos are properly formatted as `AirTubeVideo` objects, we can assign the array to the `videos` property of the `ApplicationData` object (5). That will cause `ApplicationData` to dispatch an event, notifying listeners that they should update themselves based on the new data.

We've now built the service class for the AirTube application. Next we'll build the windows for the application and wire everything up.

2.5.5 Retrieving .flv URLs

The YouTube API at the time of this writing doesn't return a direct URL to the .flv files for the videos. Instead, when you request videos from YouTube, it returns a URL to the Flash-based video player that in turn accesses the .flv file. In order to retrieve videos that we can download and save locally, the AirTube application needs to get a direct URL to the .flv file for a video. To achieve this feat, we have to resort to a bit of magic. In this section, we'll build a class that retrieves the actual .flv URL for a given video based on its YouTube player URL.

NOTE The mechanism by which we retrieve the URL for an .flv on YouTube is, to put it bluntly, a hack. As a result, we can't guarantee that YouTube will continue to support access to files in this way. Should the system change, please know that we'll make every reasonable effort to find a new working solution and make that available through the book's web site.

At the time of this writing, the URL to retrieve an .flv file from YouTube requires two pieces of information that YouTube calls `video_id` and `t`. These two pieces of information can be retrieved by making a request to the player URL as returned by the YouTube service, and then reading the `video_id` and `t` values from the URL to which the

player URL redirects. An example might help clarify this. The following is an example of a player URL returned from the YouTube service:

```
http://www.youtube.com/v/llRw9UG48Dw
```

If you view that in a web browser, you'll notice that it redirects to the following URL:

```
http://www.youtube.com/swf/l.swf?video_id=llRw9UG48Dw&rel=1&eurl=&i
➥url=http%3A//i.ytimg.com/vi/llRw9UG48Dw/default.jpg&
➥t=OEgsToPDskJtLebBhzjJbUnpN-uo9iSI
```

In the preceding URL, we've shown in bold the `video_id` and `t` values to make them easier to see. These are the values we want to retrieve from the URL. With those two values, we can retrieve the .flv file using the following URL, with the `video_id` and `t` values we've retrieved being substituted for the italicized text:

```
http://www.youtube.com/get_video.php?video_id=video_id&t=t
```

In order to achieve our goal, we write a helper class called `YouTubeFlvUrlRetriever` and add a method to the `AirTubeService` class. Go ahead and complete the following steps:

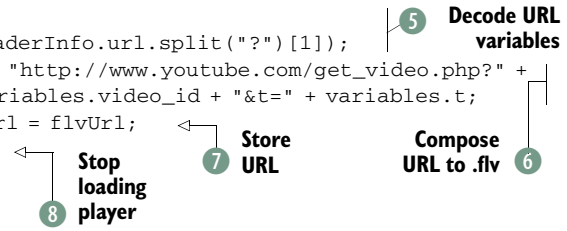
1. Open a new ActionScript class document and save it as com/manning/airtube/utilities/YouTubeFlvUrlRetriever.as relative to the source directory.
2. Add the code from listing 2.34 to the `YouTubeFlvUrlRetriever` class.

Listing 2.34 The `YouTubeFlvUrlRetriever` class

```
package com.manning.airtube.utilities {
   import com.manning.airtube.data.AirTubeVideo;
   import flash.display.Loader;
   import flash.events.Event;
   import flash.net.URLRequest;

    import flash.net.URLVariables;


   public class YouTubeFlvUrlRetriever {

      private var _currentVideo:AirTubeVideo;
      private var _loader:Loader;

      public function YouTubeFlvUrlRetriever() {
         _loader = new Loader();
      }

      public function getUrl(video:AirTubeVideo):void {
         _currentVideo = video;
         var request:URLRequest = new
URLRequest(video.video.playerURL);
         _loader.contentLoaderInfo.addEventListener(Event.INIT,
         ➥videoInitializeHandler);
         _loader.load(request);
      }

      private function videoInitializeHandler(event:Event):void {
         var variables:URLVariables = new URLVariables();
```

1 Create loader
2 Store current video
3 Compose request to player URL
4 Listen for init event

```
            variables.decode(
            ➥_loader.contentLoaderInfo.url.split("?")[1]);
             var flvUrl:String = "http://www.youtube.com/get_video.php?" +
                "video_id=" + variables.video_id + "&t=" + variables.t;
             _currentVideo.flvUrl = flvUrl;
             _loader.unload();
          }
       }
    }
```



In this code, the first thing we do is construct a new `Loader` object 1. We use a `Loader` object to make the HTTP request to the YouTube player URL and retrieve the `video_id` and `t` variables.

When we're ready to request a URL, the first thing we do is store a reference to the video 2 in order to update its `flvUrl` property once we've retrieved the URL. Next we can create a request that points to the URL of the YouTube player for the video 3, and then we load the URL 4. This will make the request and receive the redirected URL once it initializes.

When the application receives an init response to the request, the `Loader` object's `contentLoaderInfo.url` property will be the redirect URL containing the `video_id` and `t` variables. We're splitting the URL on the question mark to get just the querystring portion, then running that through `decode()` 5 in order to have the `URLVariables` object parse out the variables. That enables us to construct the URL to the .flv file using the variables we just decoded 6. Once we've composed the correct URL, we can update the `flvUrl` property of the current video 7. And we also need to unload the `Loader` object 8 to stop downloading the YouTube video player because we only needed to make the request to retrieve the variables, not the player itself.

3 Open `AirTubeService` and add the method from listing 2.35. This method sets the `currentVideo` property of `ApplicationData` in order to keep track of the video that the user has selected, then it tests to see whether the `flvUrl` property of the video is null. The property will be null if the video hasn't yet been configured by the `YouTubeFlvUrlRetriever`. If the property is null, we run it through the `YouTubeFlvUrlRetriever`.

Listing 2.35 The `configureVideoForPlayback()` method

```
public function configureVideoForPlayback(video:AirTubeVideo):void {
   ApplicationData.getInstance().currentVideo = video;
   if(video.flvUrl == null) {
      new YouTubeFlvUrlRetriever().getUrl(video);
   }
}
```

We've now successfully written the code to retrieve the .flv URL. Next we'll start building the windows that allow the user to search for videos, see the results, and even play back video.

2.5.6 Building the AirTube main window

The main window for the AirTube application (which you can see in figure 2.10) consists of a search control bar, a list view for video search results, and a button to launch the selected video for playback. In this section, we'll build the main window and wire it up to allow the user to make requests to the YouTube service for video results. Follow these steps:

1. Create a new MXML document and save it as AirTube.mxml to the source directory of the AirTube project.
2. Add the code from listing 2.36 to the document. You'll notice that we've designed AirTube.mxml such that it also uses the Singleton pattern.

Listing 2.36 AirTube.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
 layout="absolute" width="800" height="600"
 creationComplete="creationCompleteHandler();">
   <mx:Script>
      <![CDATA[
         import com.manning.airtube.services.AirTubeService;
         import com.manning.airtube.data.ApplicationData;

         static private var _instance:AirTube;

         private var _service:AirTubeService;

         static public function getInstance():AirTube {
            return _instance;
         }

         private function creationCompleteHandler():void {
            _service = AirTubeService.getInstance();
            _service.key = "YourAPIKey";
            _instance = this;
         }
      ]]>
   </mx:Script>
   <mx:VBox width="100%">
      <mx:Label text="AirTube: Adobe AIR and YouTube" />
      <mx:HBox>
         <mx:Label text="tags:" />
         <mx:TextInput id="tags" text="Adobe AIR" />
         <mx:Button label="Search For Videos" />
      </mx:HBox>
         <mx:TileList id="videoList"
         dataProvider="{ApplicationData.getInstance().videos}"
         width="100%" height="400"
         columnCount="2" horizontalScrollPolicy="off" />
   </mx:VBox>
</mx:WindowedApplication>
```

1 No lazy instantiation
2 Run initialization
3 Get service
Set developer key
Set Singleton instance
Component to display results

In most cases of the Singleton pattern, the `getInstance()` method uses lazy instantiation, creating an instance of the class if it hasn't been created already.

In this case, that isn't necessary. We can simply return a reference to `_instance` (1) without testing whether `_instance` is null. Because AirTube.mxml is the `WindowedApplication` instance for the AirTube application, we know that it'll always exist before any other code tries to reference it.

The AirTube.mxml code is wired up to call the `creationCompleteHandler()` method (2) when the `creationComplete` event occurs. In the `creationCompleteHandler()` method, we add the code that needs to occur when the application starts. We'll use the `AirTubeService` instance a lot in this document. Therefore, we'll just store a reference to it (3). We then need to set the developer key to use for the YouTube service. You must replace *YourAPIKey* with your YouTube API key for this to work. And we also need to assign the `this` instance to the `_instance` property. This is part of the Singleton pattern. We know that there's only one instance of AirTube.mxml ever created, and we know that it's automatically created when the application starts.

We use a tile list component to display the search results when the user searches for videos. Note that we're using databinding to wire up the component with the `videos` property of the `ApplicationData` instance. Any time the `videos` property updates, the tile list will refresh.

3 Add the code that makes the service request for the videos. This requires adding an event handler to the search button. The changes to the code are shown in bold in listing 2.37.

Listing 2.37 Adding search behavior to AirTube.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
 layout="absolute" width="800" height="600"
 creationComplete="creationCompleteHandler();">
   <mx:Script>
      <![CDATA[
         import com.manning.airtube.services.AirTubeService;
         import com.manning.airtube.data.ApplicationData;

         static private var _instance:AirTube;

         private var _service:AirTubeService;

         static public function getInstance():AirTube {
            return _instance;
         }

         private function creationCompleteHandler():void {
            _service = AirTubeService.getInstance();
            _service.key = "YourAPIKey";
            _instance = this;
         }

         private function getVideosByTags():void {
            _service.getVideosByTags(tags.text);
         }
```

```
        ]]>
    </mx:Script>
    <mx:VBox width="100%">
        <mx:Label text="AirTube: Adobe AIR and YouTube" />
        <mx:HBox>
            <mx:Label text="tags:" />
            <mx:TextInput id="tags" text="Adobe AIR" />
            <mx:Button label="Search For Videos" click=
            ➥"getVideosByTags();" />
        </mx:HBox>
        <mx:TileList id="videoList"
            dataProvider="{ApplicationData.getInstance().videos}"
            width="100%" height="400"
            columnCount="2" horizontalScrollPolicy="off" />
    </mx:VBox>
</mx:WindowedApplication>
```

4 Test your application at this point, and you'll probably see something similar to figure 2.12 (assuming you run the search). You'll notice that the search results show up simply as text in the list—and not particularly meaningful or useful text to most users. We'd like to change that by creating a custom item renderer for the tile list. To do that, first open a new MXML document and save it as com/manning/airtube/ui/VideoTileRenderer.mxml relative to your project source directory.

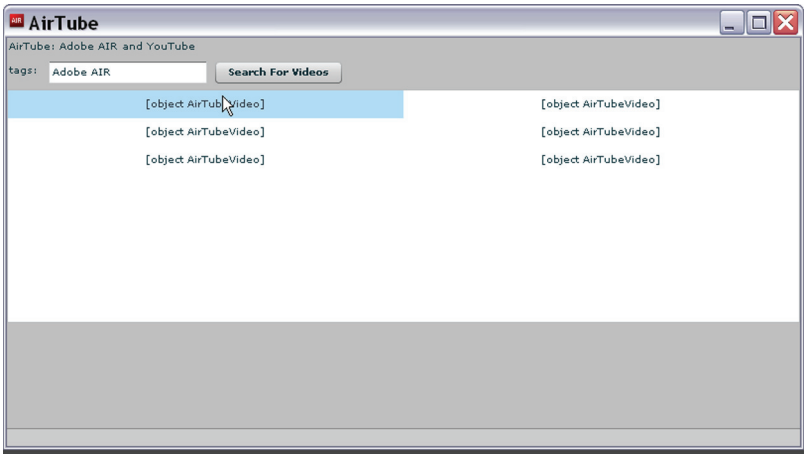


Figure 2.12 The AirTube main window before customizing the item renderer for the search results

5 Add the code from listing 2.38 to the `VideoTileRenderer` component. The code is simple. It uses an image and a label to display the thumbnail and the title of the video that's passed to the renderer via the `data` property.

Listing 2.38 VideoTileRenderer.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<mx:HBox xmlns:mx="http://www.adobe.com/2006/mxml" width="200"
➥height="100"
 verticalScrollPolicy="off" horizontalScrollPolicy="off">
   <mx:Image source="{data.video.thumbnailUrl}" />
   <mx:Label text="{data.video.title}" />
</mx:HBox>
```

6 Update AirTube.mxml to tell it to use the correct item renderer. All we need to do is update one line of code by adding an `itemRenderer` attribute to the `TileList` tag, as shown in listing 2.39.

Listing 2.39 Using a custom item renderer with the tile list

```
<mx:TileList id="videoList"
         dataProvider="{ApplicationData.getInstance().videos}"
         width="100%" height="400"
         columnCount="2" horizontalScrollPolicy="off"
         itemRenderer="com.manning.airtube.ui.VideoTileRenderer" />
```

Now that we've created the main window, we're going to add two more windows: the video and HTML windows.

2.5.7 Adding the video and HTML windows

The AirTube application has two windows in addition to the main window. One allows for the playback of videos and one allows for the viewing of HTML pages. (In this case we're allowing the user to view the YouTube page for a video.) To add these windows to the application, complete the following steps:

1 Create a new MXML document and save it as com/manning/airtube/windows/VideoWindow.mxml relative to the project's source directory.
2 Add the code from listing 2.40 to the `VideoWindow` component.

Listing 2.40 VideoWindow.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Window xmlns:mx="http://www.adobe.com/2006/mxml"
   width="400" height="400"
   type="utility"
   closing="closingHandler(event);"
   creationComplete="creationCompleteHandler();">

   <mx:Script>
      <![CDATA[
         import com.manning.airtube.services.AirTubeService;
         import com.manning.airtube.data.ApplicationData;

         [Bindable]
         private var _applicationData:ApplicationData;

         private function creationCompleteHandler():void {
            _applicationData = ApplicationData.getInstance();
```

Set width and height

Set type to utility

Handle closing event

Handle creationComplete event

```
            }

            private function closingHandler(event:Event):void {
               event.preventDefault();
               visible = false;
            }

            private function togglePlayback():void {
               if(videoDisplay.playing) {
                  videoDisplay.pause();
                  playPauseButton.label = "Play";
               }
               else {
                  videoDisplay.play();
                  playPauseButton.label = "Pause";
               }
            }

         ]]>
      </mx:Script>
      <mx:VBox>
         <mx:Label text=
         ➥"{_applicationData.currentVideo.video.title}" />
         <mx:VideoDisplay id="videoDisplay"
              source="{_applicationData.currentVideo.flvUrl}"
              width="400" height="300" />
         <mx:HBox>
            <mx:Button id="playPauseButton" label="Pause"
                    click="togglePlayback();" />
         </mx:HBox>
      </mx:VBox>
   </mx:Window>
```

Prevent window from closing 1

Display video title 2

Wire up video display 3

Notice that when the closing event occurs, we handle it to prevent the default behavior 1 and instead set the visibility of the window. Also note that the label and video display components use data binding to wire themselves up to properties of the current video 2 3.

3 Create a new MXML document and save it as com/manning/airtube/windows/HTMLWindow.mxml relative to the project's source directory.

4 Add the code from listing 2.41 to HTMLWindow.mxml. This window has no content at this time. We'll add content in chapter 7. You'll notice that this component also handles the closing event in the same way as `VideoWindow`.

Listing 2.41 HTMLWindow.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Window xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
   width="800" height="800" closing="closingHandler(event);">
   <mx:Script>
      <![CDATA[

         private function closingHandler(event:Event):void {
            event.preventDefault();
            visible = false;
```

```
        }

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

5 Update AirTube.mxml to contain references to instances of the two windows we just created, and allow the user to launch the video window and view a selected video. Also, add code to close all the windows when the user closes the application. Listing 2.42 shows the changes in bold.

Listing 2.42 Updating AirTube.mxml with the windows

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
 layout="absolute" width="800" height="600"
 creationComplete="creationCompleteHandler();"
closing="closingHandler();">
  <mx:Script>
    <![CDATA[
      import com.manning.airtube.services.AirTubeService;
      import com.manning.airtube.data.ApplicationData;

      static private var _instance:AirTube;

      private var _service:AirTubeService;
      private var _videoWindow:VideoWindow;
      private var _htmlWindow:HTMLWindow;
      static public function getInstance():AirTube {
        return _instance;
      }

      private function creationCompleteHandler():void {
        _service = AirTubeService.getInstance();
        _service.key = "YourAPIKey";
        _instance = this;
        _videoWindow = new VideoWindow();
        _htmlWindow = new HTMLWindow();
      }

      private function getVideosByTags():void {
        _service.getVideosByTags(tags.text);
      }

      private function playVideo():void {
        var video:AirTubeVideo =
        ➥videoList.selectedItem as AirTubeVideo;
        _service.configureVideoForPlayback(video);
        if(_videoWindow.nativeWindow == null) {
          _videoWindow.open();
        }
        else {
          _videoWindow.activate();
        }
      }

      public function launchHTMLWindow(url:String):void {
```

1 Create instance of windows

2 Retrieve selected video

3 Get .flv URL

4 Open video window

5 Open HTML window

```
            if(_htmlWindow.nativeWindow == null) {
                _htmlWindow.open();
            }
            else {
                _htmlWindow.activate();
            }
        }

        private function closingHandler():void {
              for(var i:Number = 0; i <
              ➥nativeApplication.openedWindows.length; i++) {
                 nativeApplication.openedWindows[i].close();
              }
         }

    ]]>
  </mx:Script>
  <mx:VBox width="100%">
     <mx:Label text="AirTube: Adobe AIR and YouTube" />
     <mx:HBox>
        <mx:Label text="tags:" />
        <mx:TextInput id="tags" text="Adobe AIR" />
        <mx:Button label="Search For Videos"
         click="getVideosByTags();" />
     </mx:HBox>
     <mx:TileList id="videoList"
        dataProvider="{ApplicationData.getInstance().videos}"
        width="100%" height="400"
        columnCount="2" horizontalScrollPolicy="off" />
     <mx:Button label="Play Selected Video" click="playVideo();"
        enabled="{videoList.selectedItem != null}" />
  </mx:VBox>
</mx:WindowedApplication>
```

6 **Close all windows**

7 **Button to play video**

The application only allows one video window and one HTML window. We reuse these instances for every video and every HTML page the application opens, thus creating instances of each of them 1. We open them based on user action.

When the user wants to play a video, we need to do several things. First we need to retrieve the video that the user has selected from the list 2. Then we need to run the video through the `configureVideoForPlayback()` method to make sure it has the URL to the .flv file 3. And then we can open the video window instance 4.

In our code, we've added a method that doesn't get called yet. We're defining the `launchHTMLWindow()` method 5 here in order to call it later. This method simply opens the HTML window.

As with most of the applications we build, we want to make sure that, when the user closes the main window, it closes all the windows in the application. We use the `closingHandler()` method 6 for this purpose.

We also add a button to play the selected video 7. When the user clicks on the button, it calls `playVideo()`. The button is only enabled when a video has been selected from the list.

With that, we've completed the first phase of the AirTube application. Thus far, the user can search for videos and play back a video in the video window. As we continue through subsequent chapters, we'll add more functionality to the application.

2.6 Summary

In this chapter, you've learned all the fundamentals of working with applications, windows, and menus. That's a lot of information, and much of it's probably new to you, because the concepts are more relevant to desktop applications than they are to web applications.

Take your time and reread any material from this chapter that you feel you need to review. Here's a summary of some of the key points you'll want to be familiar with:

- Applications and windows are represented programmatically using `NativeApplication` and `NativeWindow` objects in pure ActionScript, and `WindowedApplication` and `Window` objects in Flex.
- Creating windows, opening windows, and adding content to windows are independent actions that you need to manage.
- You can create windows that have irregular shapes.
- Applications allow you to do a variety of application-level functions such as detecting user idleness and launching an application in full-screen mode.
- AIR allows you to create native operating system menus and use them in a variety of ways, including window menus, application menus, icon menus, context menus, and pop-up menus.

When you're ready, go ahead to the next chapter, where you'll learn all about working with the file system to do things such as read and write files.