

12

Introduction to pop-ups

This chapter covers:

- Creating a pop-up window
- Sizing and positioning pop-ups
- Communicating with a pop-up window
- Closing pop-up windows
- Using alerts

When you construct an application, your design will likely often call for a pop-up or editing window. Flex offers a convenient mechanism in the *pop-up manager* to help you create, delete, position, close, and destroy windows. When you're choosing the type of pop-up window to add to your application, it's a good idea to plan ahead and first determine what the purpose of the pop-up will be and what you'll need to do in the pop-up window.

12.1 Creating your first pop-up

Every pop-up window is created and manipulated through the pop-up manager. The pop-up manager is the class that handles the initialization of the window and manages window layering within the application, which relieves you of the responsibility of avoiding placement conflicts. The pop-up manager is simple to use and

doesn't carry a lot of properties to configure. Let's look at an example of how to create your first pop-up window.

12.1.1 First things first: create your title window

Several options are available to you for creating a pop-up. Each option requires at least one window file that is used for the display of the window. The pop-up manager calls this window and renders it in a layer above any existing layers. Additional new pop-up windows are created similarly and placed above other open pop-up or alert windows. When the top window layer is closed, the next highest window regains prominence. This continues until the last pop-up is closed, leaving the main layer, which is your parent application layer.

Listing 12.1 introduces the `TitleWindow` component, which enables all this to take place.

Listing 12.1 SimplePopupWindow.mxml: `TitleWindow` component in action

```
<?xml version="1.0" encoding="utf-8"?>
<mx:TitleWindow xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="vertical" width="400" height="100">
  <mx:Text text="Hello there! I'm a simple popup window.
    I don't do much.">
  </mx:Text>
</mx:TitleWindow>
```

When creating a pop-up window, you have a choice of the container component on which to base the pop-up (a.k.a. a *base tag*). One of the most common—and easiest to use—tags is `TitleWindow`. Only certain tags can act as the main base tag; tags you would normally use, such as `<Application>`, `<VBox>`, and `<HBox>`, aren't available for use as the base tag of a pop-up window.

Once you've configured `TitleWindow` as your base component window tag, you can add any other MXML or ActionScript tags. Listing 12.1 uses the `mx:Text` tag, which lets you display a simple text message on the screen. This is a bare-bones example of a user interaction message. As you progress, you'll see that you can do much more with it.

Now, let's look at how to present this window to the user via the `PopUpManager`.

12.1.2 Using `PopUpManager` to open the window

With your pop-up window text complete, let's put it to work by creating the code to use it. Follow these steps:

- 1 Create a project in Flex Builder called CH12.
- 2 Create a subfolder called windows under CH12's src folder.
- 3 Copy `SimplePopupWindow.mxml` from listing 12.1 into the src/windows folder.
- 4 Create a `testSimplePopup.mxml` file in the src folder, and copy the contents of listing 12.2 into it.

Listing 12.2 generates the pop-up window shown in figure 12.1, which opens the pop-up as soon as the application loads.

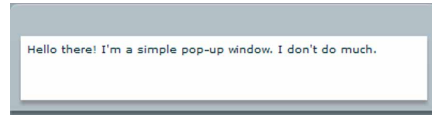


Figure 12.1 A basic pop-up window

Listing 12.2 testSimplePopup.mxml: showing a pop-up window

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  creationComplete="openSimpleWindow()" >
<mx:Script>
  <![CDATA[
    import mx.managers.PopUpManager;
    import windows.SimplePopupWindow;

    private var simpleWindow:Object;
    private function openSimpleWindow():void
    {
      simpleWindow =
        PopUpManager.createPopUp(this, windows.SimplePopupWindow, false);
    }
  ]]>
</mx:Script>
</mx:Application>
```

Looks in
windows folder

This is the most commonly used and simplest mechanism for calling a new pop-up window. The keyword `this` (the first option in the `createPopUp()` method) indicates that the parent of the pop-up is the component that launched the window.

The next option is the class, or file name, you're using to create the pop-up window. The final option determines whether the window should be *modal*. Modal windows constrain the user to clicking only within the confines of the pop-up window. Modal windows don't permit clicks beyond the borders of the window itself. Attempts to click back to the main application window (or anywhere outside the pop-up) are ignored.

Notice that in listing 12.2 you assigned the result of the `createPopUp()` method call to a variable. This is because `createPopUp()` returns an initialized window object that represents the component you created in listing 12.1. If you save the returned object to a variable, you'll be able to interact with it, which we'll show you how to do a little later in this chapter.

Pay attention to the scope, which you learned about in previous chapters. In listing 12.2, the result is saved to a private function scope that won't allow access from outside the function. If you wish to access the pop-up window from outside the function, you need to make the variable available as a class- or component-scoped variable.

Now that you've seen how to create a pop-up, you need to know how to close it when the user is finished with it.

12.1.3 Closing the pop-up

In figure 12.2, you'll notice an X in the upper-right corner of the pop-up window. As you already know, this is the universally recognized *close* button used by the vast majority of web and desktop applications.

The close button is available as an option when you use a `TitleWindow` tag. You can access this option through `TitleWindow`'s `showCloseButton` property, which accepts either `true` or `false` as a value. Setting `showCloseButton` to `true` displays the X button in the corner of the pop-up. When the button is clicked, `TitleWindow` dispatches a `close` event that can be used as a trigger by components designated to listen for it.

It doesn't do all of the work for you, but it gets you close. Listing 12.3 illustrates how the `closeMe()` function is used to remove the pop-up from view.



Figure 12.2 Notice the close button in the upper-right corner of the pop-up window.

Listing 12.3 Closing a pop-up window

```
private function closeMe():void
{
    PopUpManager.removePopUp(this);
}
```

After closing the window, the next thing to look at is controlling the position of the pop-up.

12.2 Controlling the window location

When you first launch a window, you may see that it's not automatically positioned in the middle of the screen. You need to give your pop-up layout instructions, or it will appear in the upper-left corner of the parent object in which it was created.

12.2.1 Using the `centerPopUp()` method

`PopUpManager` contains a `centerPopUp()` method to center (to no surprise) the location of a pop-up window. The `centerPopUp()` method takes a single argument of type `IFlexDisplayObject`, which is any visual component—in other words, it's the name of a pop-up window, as the following line describes:

```
PopUpManager.centerPopUp(IFlexDisplayObject);
```

The `IFlexDisplayObject` can be any pop-up window variable you define. For example, when you create the window, if you set the result of the create function call to a

variable, you can use that in the parent application to center the pop-up. To apply this to the previous example, you could do something like this:

```
simpleWindow =
    PopUpManager.createPopUp(this, windows.SimplePopupWindow, false);
    PopUpManager.centerPopUp(simpleWindow as IFlexDisplayObject);
```

Because `simpleWindow` is declared as a generic object, Flex Builder doesn't know if `simpleWindow` is an `IFlexDisplayObject`. In the `centerPopUp()` method, you convince Flex to believe all is well by using casting.

NOTE The `centerPopUp()` method centers a pop-up relative to its parent container, not the center of the page.

The centering method operates somewhat counter-intuitively. Initially, most Flex coders assume that centering a pop-up means it will display in the center of the browser window. This isn't the case. Centering a pop-up window establishes a position over the center of the parent object in which you created the pop-up. If your parent object occupies a small area in the upper-right corner of the browser window, the `centerPopUp()` method will center the pop-up over that window, not the middle of the screen as defined by the main browser window.

The easiest strategy to deal with the placement issue is to name a parent as close to the application root as possible and define this parent with the largest view possible within the window space. Or, have your main application perform all the window openings (for example, you can have a global function that components can call to ask the main application to create pop-ups).

In most cases, using `centerPopUp()` will work fine for noncritical window positioning. In the majority of circumstances, it's all you'll need to manage the placement of your windows.

12.2.2 Calculating window placement

Centering a window is easy, but you can get fancy with manipulating the position and dimensions of a pop-up. Let's demonstrate by creating another pop-up in your `src/windows` directory called `MoveWindow.mxml`. In this project (see figure 12.3), the pop-up has buttons that, when clicked, move the pop-up to each corner of the window; another button toggles visibility properties.



Figure 12.3 `MoveWindow.mxml`: clicking the buttons manipulates the *x* and *y* coordinates to reposition this pop-up.

Although the manual method of placing windows on a screen gives you more control over the exact location of the window, it requires more calculation and greater attention to where you position the window. Listing 12.4 presents the code snippet used to determine where the window is currently located and its dimensions (which you'll later manipulate to move the pop-up to each of the four window corners). The

variables `this.x` and `this.y` correspond to the horizontal and vertical coordinates, respectively.

Listing 12.4 Accessing a pop-up's position and dimensions

```
var currentX:Number = this.x;
var currentY:Number = this.y;
var currentWidth:Number = this.width;
var currentHeight:Number = this.height;
```

Imagine that the window is a grid with lines that run up and across the window, as illustrated in figure 12.4. X and y are points on that grid for the window or whatever object it is that you're measuring. In listing 12.4, `this.x` and `this.y` are points on the grid.

X and y are points that are available in every object that can be displayed on the screen, and they match up to the current location of the object on the screen. Thus the x and y values let you identify any single pixel on the screen.

If you evaluate the x and y properties of a newly created window that doesn't have any positioning applied to it, you'll see that the window has an x value of zero (0) and a y value of zero (0) (see the upper left corner of figure 12.4). If you combine that with the height and width of the window, you can arrive at the placement of the object on the screen.

Leveraging that as the premise for moving a pop-up window, listing 12.5 shows how to use the dimensions of the pop-up against the dimensions of the parent component (which is the main application in this case) to calculate a new position for the window. To make it more interesting, you add a 10-pixel *buffer* from the edges.

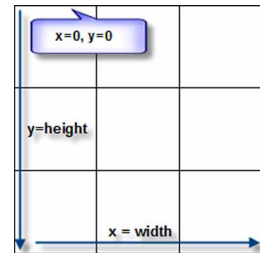


Figure 12.4 Visualizing the positioning parameters of a pop-up

Listing 12.5 MoveWindow.mxml: manipulating the pop-up's position

```
<?xml version="1.0" encoding="utf-8"?>
<mx:TitleWindow xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="vertical" width="340" height="150"
  showCloseButton="true" close="closeMe()" >

<mx:Script>
<! [CDATA[
  import mx.managers.PopUpManager;

  private function moveWindow(where:String):void
  {
    var newX:Number = 0;
    var newY:Number = 0;
    var buffer:Number = 10;
    switch (where)
    {
      case "toprightcorner" :
        newY = buffer;
        newX = (parent.width - this.width)-buffer;
```

```

        break;
    case "topleftcorner" :
        newY = buffer;
        newX = buffer;
        break;
    case "bottomrightcorner" :
        newY = (parent.height - this.height)-buffer;
        newX = (parent.width - this.width)-buffer;
        break;
    case "bottomleftcorner" :
        newY = (parent.height - this.height)-buffer;
        newX = buffer;
        break;
    }
    this.move(newX,newY);
}

private function closeMe():void
{
    PopUpManager.removePopUp(this);
}
]]>
</mx:Script>

<mx:HBox>
  <mx:Button label="Top left corner"
    click="moveWindow('topleftcorner');"/>
  <mx:Button label="Top right corner"
    click="moveWindow('toprightcorner');"/>
</mx:HBox>
<mx:HBox>
  <mx:Button label="Bottom left corner"
    click="moveWindow('bottomleftcorner');"/>
  <mx:Button label="Bottom right corner"
    click="moveWindow('bottomrightcorner');"/>
</mx:HBox>
</mx:TitleWindow>

```

Calculates x and y coordinates based on window dimensions

As we mentioned earlier, when you're calculating and setting window placement by coordinates, it's particularly important to remember that when centering a pop-up, all coordinate calculations are relative to the size and location of the parent object—not the coordinates of the main window.

The issues surrounding window location become important when you're dealing with modal windows. When you define a modal window, the user is forced to work within that window only and is unable to click any items outside the modal window's boundaries without first closing that window. As a worst-case scenario, if you define a modal window that appears outside of the visible screen, it renders the application unusable because the user can't close the window or click anywhere else on the screen to release the modal window's control of the application.

Specifying the location of pop-up windows in your application is a critical visual and functional responsibility. Flex provides several more ways to control pop-up

attributes, to help simplify your programming burden and help with your design's visual appearance. The next aspect we'll look at is setting the window's transparency level by managing the alpha property.

12.3 **Setting window transparency**

One of the nice features of pop-up windows is that you can give your application some high-end pizzazz and set varying levels of window transparency. You set transparency by changing the alpha property of the pop-up window. This property is available for all visible objects, but it's particularly noticeable and useful when employed on pop-up windows.

The alpha property is defined using a numeric value between 0 and 1 that varies the opacity of your window. A value of zero (0) renders the window invisible, whereas a value of 1 makes it fully visible. Transparency settings apply only to the window itself, not necessarily to the items contained within it.

From within the window, you can set the window's transparency along the lines of this code snippet:

```
this.alpha = 0.4;
```

From outside the window (for example, from the component that created the pop-up), the code is similar. It references the name of the pop-up window instance:

```
moveWindow.alpha = 0.4;
```

To add the ability to control transparency in your sample application, let's add a Text-Input that accepts a number for the alpha and a function that sets the value. Add the following MXML components to the previous code:

```
<mx:HBox>
  <mx:TextInput id="visibilityText" text="{this.alpha}" restrict="0-9\"
    width="60" />
  <mx:Button label="Set Visibility" click="setVisibility();" />
</mx:HBox>
```

Then, add the following ActionScript function to provide the logic:

```
private function setVisibility():void
{
  this.alpha = Number(visibilityText.text);
}
```

As illustrated in figures 12.5 and 12.6, varying the transparency of a window allows for unique pop-up designs and gives you flexibility in how you manage the look and feel of your application.

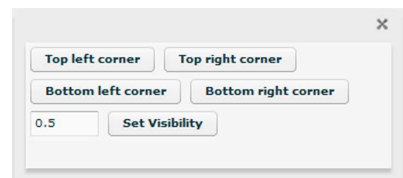


Figure 12.5 A pop-up window with transparency set to 0 (completely invisible border)



Figure 12.6 The same pop-up with its transparency set to a value of 0.5, which renders the window 50% opaque

Now that we've dealt with the visual aspects of creating and displaying a pop-up window, we'll look at one of the more common use cases: the pop-up form.

12.4 Data integration with pop-ups

In this section, you'll take your knowledge of pop-ups one step further by learning how to retrieve data inside the pop-up (usually, information the user enters on a form) or send information to a pop-up. To set up the example, you'll create a pop-up with a text input that will continually append text to a variable, as shown in figure 12.7.

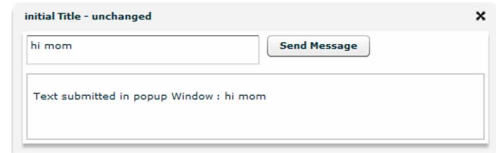


Figure 12.7 This pop-up window accepts user input and saves it to a variable that will be retrieved later.

Let's start by creating a new MXML file called `AdvancedPopupWindow.mxml` in the `src/windows` folder. Enter the code shown in listing 12.6.

Listing 12.6 `AdvancedPopupWindow.mxml`: pop-up that captures user input

```
<?xml version="1.0" encoding="utf-8"?>
<mx:TitleWindow xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical"
    width="500" height="300" showCloseButton="true"
    close="closeMe()" title="initial Title - unchanged" >

    <mx:Script>
    <![CDATA[

        import mx.managers.PopUpManager;

        [Bindable] public var myPopUpVar:String = "";
        [Bindable] public var myPopupLog:String = "";

        private function closeMe():void
        {
            PopUpManager.removePopUp(this);
        }

        public function setTitle(title:String):void
        {
            this.title = title;
        }

        private function sendMessage():void
        {
            var outgoingEvent:Event = new Event("popupMessage", false);
            myPopUpVar = windowInput.text;
```

```

        myPopupLog += "\n Text submitted in popup Window : " +
            windowInput.text;
    }
    ]]>
</mx:Script>
<mx:HBox width="100%" height="40%">
  <mx:TextInput id="windowInput" name="windowInput" height="30"
    width="240" text="" />
  <mx:Button id="sendMessageButton" label="Send Message"
    click="sendMessage()" />
</mx:HBox>
<mx:TextArea id="pageDescription" name="pageDescription"
  text="{myPopupLog}" width="100%" height="60%" />
</mx:TitleWindow>

```

One of the greatest advantages of using a component object to open a window is that the object can be reused and retains all of its state values.

In the example presented in listing 12.7, you open the window, make changes to it, and close the window and open it again. The window retains the changes entered by a user, because the object is never erased or reinitialized.

Listing 12.7 Using a component object as a window

```

import windows.AdvancedPopupWindow;

[Bindable]
public var advancedWindow:AdvancedPopupWindow =
  new AdvancedPopupWindow();

```

Windows that aren't generated within a component object are re-created and initialized every time the window is opened. This approach can be both good and bad, depending on how you intend to use the window. On one hand, your application can benefit from the object maintaining its state if it's to be displayed often; on the other hand, storing the window and its constituent data can be a drain on memory resources—particularly if the window is used only once.

To launch and use an existing object as a pop-up window, you only need to add the pop-up to the list managed by the pop-up manager:

```

PopupManager.addPopUp(advancedWindow, this, false);

```

The script in listing 12.7 creates a component object as a variable that you can access from within the scope of the component, class, or application. Once the object has been created in the main application or component scope, it's available outside the calling function.

As listing 12.7 demonstrates, you use the `PopupManager.addPopUp()` method to create your pop-up window. The `addPopUp()` method lets you add an existing object or component to the display list. In more technical terms, the `createPopUp()` method creates an object and initializes it, whereas `addPopUp()` expects to receive a previously initialized object, which it adds to its display list.

If you have a previously initialized object, the process of reinitializing it wipes out

any stored data. Using the `addPopUp()` method, you can display and hide the window repeatedly without affecting the data inside. This is an important distinction, particularly as you get into extracting user-edited data from a window.

12.4.1 Getting data out of your pop-up window

Half the effort involved in manipulating pop-ups is creating them; the other half is the extraction of the information they contain. A popular way of working with pop-ups is to add a form with which users can interact. A number of different ways exist for you to construct a form and manage the information entered into it.

When you're first getting started with Flex, most developers have the impulse to include a data-services tag, such as `WebService`, `RemoteObject`, or `HTTPService`, to communicate user-entered form data directly to a server. This approach is easy to manage but not reusable and can quickly become a maintenance headache if your back-end web service changes.

A better way to capture form data is to present it as public variables, then send out an event to notify the parent application that information is available to process. In the following sections, we'll show you how to do this; but first, we need to revisit events.

12.4.2 Sending events

When sending events from your pop-up window, remember that they aren't dispatched in exactly the same manner as with other components.

In general, events work much as you would expect, with one variation: events from pop-ups don't travel down the inheritance chain to the main application root. Instead, events dispatched from pop-up windows are terminated at the window root.

This is because the pop-up manager is the parent of the pop-up window. As a result, events launched in the pop-up don't bubble up to the main application page, which is the eventual parent of any object in the main application.

Any events launched from within the pop-up window start and end with the `<mx:TitleWindow/>` tag. In practical terms, this means in order to receive events from the pop-up window, you need to register any object you intend to designate as a listener.

Event listeners are objects that respond to events generated by the window (see listing 12.8). These events can be either custom events or system-type events generated by the application. For more information about events and how to use them, we recommend reviewing chapter 10. Listing 12.8 shows event listeners registered on the advanced window object.

Listing 12.8 Registering events on a window

```
advancedWindow.addEventListener("popupMessage", getWindowData);
advancedWindow.addEventListener(CloseEvent.CLOSE, windowClose);
```

In this scenario, two event listeners are configured for the `advancedWindow` object. These event listeners will monitor for the custom events to be dispatched and react to them by gathering user-supplied information from the window (listing 12.9).

Listing 12.9 Dispatching events from a window

```
private function closeMe():void
{
    PopUpManager.removePopUp(this);
    dispatchEvent(new Event("popupClose", false));
}
private function sendMessage():void
{
    var outgoingEvent:Event = new Event("popupMessage", false);
    myPopUpVar = windowInput.text;
    myPopupLog += "\n Text submitted in popup Window : " + windowInput.text;
    dispatchEvent(outgoingEvent);
}
```

← **Dispatches close event to simulate closing window**

Signals message update ←

In this case, when a user types in data and clicks the button to commit the information, whatever has been entered in the text box is copied to a public variable, after which an event is dispatched. This custom event can be monitored by other components, which can then repeat the dispatch or process the data it contains.

Listing 12.9 dispatches a custom event and uses the event as a trigger to get data from the window, as we'll explain in more detail in the next section.

12.4.3 Getting data out

To extract data from the parent, you can access the object's public properties, or you can access the object's public functions. For example, referring back to section 12.4.2, you can see when the object arrives, you can access the target's properties and public functions. Listing 12.10 sets up the event listener in the main window and calls the function when the `popupMessage` event is received.

Listing 12.10 Responding to event from the window

```
advancedWindow.addEventListener("popupMessage", getWindowData);

public function getWindowData(event:Event):void
{
    mainWindowText.text += "\n Window submitted: " +
    advancedWindow.myPopUpVar;
}
```

After the event is received, you can read the public property on the window and retrieve the necessary data. This can just as easily be a function that returns the correct data. Once you've extracted the data from the system, you can then proceed to do whatever you need with it. The full example provided with this book appends a log to show what transpired in the window.

When many first-time coders arrive at the point where they are extracting data from a pop-up window, the temptation exists to pull the data directly from the source form object. We strenuously caution against taking this approach: it violates best-practice object-coding principles by potentially forcing you to rescript your calling code in several spots if you need to update your form.

The better way to pull your data out of a window is via a user-triggered event. When the event is fired, you can copy the data from the form element into a window object property and extract the data from the property. The other approach is to include the information in an event object dispatched from the window when editing is complete.

Employing either of these approaches lets you reuse your window component to a much greater extent and also insulates your application from excessive changes. These techniques also reduce the amount of time spent testing, debugging, and recoding. These approaches can be used reciprocally to send data into your window as well.

12.4.4 Sending data to the window

You have several options available for getting data into or out of a window. First, when the window is treated as an object, all the public properties of the window can be accessed from the parent. In addition, any public functions defined on the window can be accessed from the parent. In listing 12.11, notice places where the parent application is accessing the window for the purpose of extracting or placing data.

To send data in, generally the easiest and most simple method is to call a public function on the window to which to pass the data. For example, look how the title is set for the window in listing 12.11.

Listing 12.11 Sending data to a pop-up window

```
public function launchSimplePopup():void ← Function lives in
{                                       main application
    var simpleWindow:Object =
    PopUpManager.createPopUp(this, windows.SimplePopupWindow, false);
    simpleWindow.setTitle("How about a snazzy new title?");
}

public function setTitle(newTitle:String):void ← Function lives
{                                       in window
    pageTitle = newTitle;
}
```

This is a simple example, but you can pass more complex variables, such as classes, arrays, or even objects, then use them to arrange your pop-up window.

One thing to remember about the example in listing 12.11 is that if you scope the window object to the function this way, you'll only be able to access the object within that specific function. In this case, you can only access the `launchSimplePopup()` function.

If you wish to access the window from anywhere other than the creation function, you need to make your variable a component or declare it an application-visible variable in the main component body, as shown in listing 12.12.

Listing 12.12 Creating a pop-up window object for use outside a function

```
private var simpleWindow:Object;

public function launchSimplePopup():void
{
```

```
simpleWindow =  
  PopUpManager.createPopUp(this, windows.SimplePopupWindow, false);  
  simpleWindow.setTitle("How about a snazzy new title?");  
}
```

The object in listing 12.12 combines both methods of accessing an object outside the caller function while allowing you to have the flexibility of creating the object on the fly whenever you need it.

Now that you have a good understanding of pop-up windows as well as how to work with windows using `PopUpManager`, we'll move on to another type of pop-up window you've most likely already seen and used: alerts.

12.5 Using alerts

The `Alert` class is a specialized type of class designed to rapidly create a specific type of pop-up window. You've already had some exposure to alerts by going through the examples in the book. Alerts let you quickly advise users about important information (generally errors), but they can also serve as great tools for debugging your code and messaging back values. If you're familiar with JavaScript, you'll be intimately familiar with this method of debugging.

12.5.1 Creating a simple alert

Creating an alert is as easy as pie. Listing 12.13 constructs an alert with code you've likely already used by now.

Listing 12.13 Creating a simple alert

```
import mx.controls.Alert;  
  
Alert.show("Hello World!");
```

Undoubtedly, that is a simple alert. The only item required to make an alert pop up is the message to be communicated to the user. Figure 12.8 shows the output of this code.

The alert in figure 12.8 and listing 12.13 performs only one action; you can't do much more with it except click OK to close the window.

This limited interactivity is sufficient for communicating an error and directing a user to some other location to address the problem, but the user can't interact with the alert. This isn't the end of the story; you can do much more with alerts.

12.5.2 Doing more with alerts

Alerts in general are like pop-up windows. You can perform most of the same tricks with them: listen to and move events, change transparency, and so on.

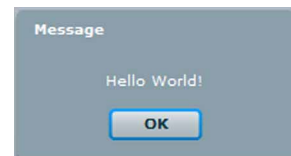


Figure 12.8 An alert issuing a simple “Hello World!” message. Click OK to close the window.

But alerts are more specialized entities than pop-ups—you can only add or remove buttons. In addition to providing a message for the alert, the following properties can also be manipulated:

- Title
- Buttons (Yes, No, OK, Cancel)
- Owning object
- Custom handler method

Any of these properties can be specified or omitted. If you use more than one button, you'll need to include the custom handler method.

12.5.3 A more advanced alert

A one-button-fits-all approach to alerts doesn't work for every occasion. Consider the scenario in which you want to prompt a quick agreement from the user, perhaps to confirm that she wants to save a file or configuration before moving to a new screen.

To do this, you may want to display a small confirmation dialog asking her if she wants to save changes before closing a window. Listing 12.14 creates an alert containing multiple buttons.

Listing 12.14 Setting up a more advanced alert

```
import mx.controls.Alert;
import mx.events.CloseEvent;

private function simpleAlert(event:Event):void {
    Alert.show("Do you want to save your changes?", "Save Changes",
        Alert.YES|Alert.NO, this, simpleAlertHandler);
}
```

In this case, the pop-up is created with the `Alert.show()` method, in which you can include message and title information. To add more buttons, you can use the `Alert` static variables to indicate which buttons to show. In listing 12.14, `Alert.YES` presents a Yes button and `Alert.NO` displays a No button.

You can code the button callouts in any order, but it won't affect the order in which the buttons are displayed in the alert. The alert buttons always appear in the following order: OK, Yes, No, Cancel. Figure 12.9 shows the output of the code from listing 12.14.

In listing 12.14, the alert handler, `simpleAlertHandler`, will be the called function when the user clicks a button. With alert windows, clicking a button automatically closes the window and fires a `CloseEvent`.

You can respond to this event by comparing the `event.detail` value against the static values on the alert object. For example, listing 12.15 tests whether the

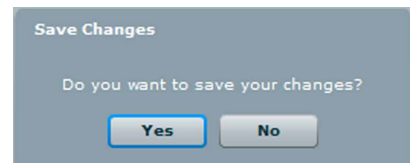


Figure 12.9 An alert message showing Yes and No buttons. You have your choice of what to do.

event.detail value is equal to `Alert.YES`. This listing shows the alert handler code for figure 12.9.

Listing 12.15 Alert handler

```
private function simpleAlertHandler(event:CloseEvent):void {
    if (event.detail==Alert.YES)
        status.text="You answered Yes";
    else
        status.text="You answered No";
}
```

NOTE It's considered good practice to use the object static variables whenever possible, because doing so makes your code easier to read and understand. Also, should those values ever change, you'll need to change them in only one place. For example, consider which is easier to understand: `Alert.YES` or `1`. Intrinsically, they mean the same thing; but `Alert.YES` is much easier to read.

In listing 12.15, you respond to the close event that's fired from the alert window. The way you respond to an alert won't change much with any other event. For instance, to respond to an alert window, you always listen for `CloseEvent` and evaluate the result of the close event to determine which button was clicked.

What can and does frequently change is the appearance of the alert window, which lends itself to a fair amount of rework.

12.5.4 Pimp this alert

You can select from a variety of appearance options for your alert window. You can resize it, change the labels on the buttons, and include an icon in the alert window. Listing 12.16 combines all these options in one tricked-out message.

Listing 12.16 Launching the pimped-out alert

```
import mx.core.IFlexDisplayObject;
import mx.controls.Alert;
import mx.events.CloseEvent;
import mx.managers.PopUpManager;

[Embed(source="assets/warning.gif")]
[Bindable]
public var iconWarning:Class;

private function customAlert(event:Event):void {
    var AlertObj:Alert;
    Alert.buttonWidth = 150;
    Alert.okLabel = "Disneyland";
    Alert.yesLabel = "Kennedy Space Port";
    Alert.noLabel = "Six Flags";
    Alert.cancelLabel = "Marine World";

    AlertObj = Alert.show("Where do you want to go today?", "Destination"
        ,Alert.OK|Alert.YES|Alert.NO|Alert.CANCEL,
```

**Assigns custom
button labels**

```

this, customAlertHandler, iconWarning, Alert.YES); ← Uses imported
AlertObj.height = 150;                               image
AlertObj.width = 700;
PopupMenu.centerPopUp(AlertObj as
    IFlexDisplayObject); ← Centers
}                                                       alert

```

You can set the labels for each of the buttons by using the `buttonLabel` property (one for each button). Setting the `buttonWidth` property allows you to determine the size of the buttons. Unfortunately, it's a one-size-fits-all situation; whatever size you choose in `Alert.buttonWidth` applies to all the buttons in that window.

In listing 12.16, you also import an image and use the image as the icon in the pop-up. The `Alert` class lets you set an icon to the right side of the alert message. In addition, you can designate which button displays as the default selection. In this example, you set `Alert.YES` (the last argument in the `Alert.show` call) as the default button. You can see what the pimped-out alert looks like in figure 12.10.



Figure 12.10 Pimped-out alert with a custom icon, custom button labels, and a default button

Listing 12.16 also sets the returned result from the `Alert.show` function to an object. This object is an initialized `Alert` object. Once the alert has opened, it acts similarly to a standard pop-up. You can do nearly everything to an alert window that you can do with a normal pop-up window, including changing its height, width, position, and transparency.

To establish which button was clicked, listen for the close event and trap the value using the `event.detail` value, as in listing 12.15.

It's also important to remember to change the button labels back to their original values if you've changed them, as shown in listing 12.17. If you don't restore the `buttonLabel` values, any other alert will display the same labels.

Listing 12.17 Pimped-out alert handler

```

private function customAlertHandler(event:CloseEvent):void {
    var message:String = "Woohoo! Looks like we're going to ";
    switch (event.detail) {
        case Alert.YES :
            status.text = message + Alert.yesLabel;
            break;
        case Alert.NO :
            status.text = message + Alert.noLabel;
            break;
        case Alert.OK :
            status.text = message + Alert.okLabel;
    }
}

```

```

        break;
    case Alert.CANCEL :
        status.text = message + Alert.cancelLabel;
    }
    Alert.okLabel = "OK";
    Alert.yesLabel = "Yes";
    Alert.noLabel = "No";
    Alert.cancelLabel = "Cancel";
}

```

**Resets
labels**

Changing the look and feel of the alert is one way you can furnish your application with a little extra sophistication. But it's not strictly necessary to change the alert function much at all. In most cases, programmers show a Yes/No type of alert because it's quick and easy and a modal way of interacting with the user. Now that you've seen how you can modify alert messages, you can include some of these extras to give your application polish.

12.6 Summary

Many applications use pop-up windows in some form or another. Pop-up windows can be easy to manage if you remember a few things:

- Creating a pop-up can be done with or without a companion MXML file, but it's most often easier using an MXML component.
- Events broadcast from a pop-up window always terminate at the pop-up window root. Events aren't broadcast through the parent up to the application root.
- Using a public-level object is an effective way of maintaining window state even when the window is closed. Creating this object often makes it easier to access the window in case you need to send data.
- A window can have public properties and public functions, which can both be used to set and retrieve data from the window.
- Pop-up placement in the application depends on the location and placement of the parent that generated the pop-up. In most cases, you should try to use a parent as close to the root as possible, or at least a component that has the widest visible range on the screen.
- You can modify alerts to include an icon image and up to four buttons. The button labels can be changed. Using `closeEvent` along with the `event.detail` item lets you tell which button was clicked.
- Alert windows act like pop-up windows. Most of the actions you expect on a pop-up window can also be done on an alert, including moving, resizing, and varying the transparency.

You've made it through another chapter, and we're happy you've stuck with us this far. The next chapter builds on the component functionality we presented here and adds view states, which allow you to modify the visual look of your components (including pop-up windows).