Let’s put our newly acquired understanding of layout and ActionScript to work and apply it to a fundamental operation of any application—capturing user input.

As mentioned in chapter 4, even though Flex offers a `Form` component, its use is optional and you’ll find it functions best as a layout tool. In the land of Flex you are equipped with control components (usually referred to as controls) that display information and accept user input. Alongside controls are events and event handlers that recognize and respond to user actions, such as clicking a mouse.

When event handler functions run, they access data from whatever source they’ve been instructed; there’s no master `Form` tag that contains all the inputs as in HTML.

NOTE Say goodbye to the notion of HTML forms. The optional `<Form>` tag in Flex does nothing more than lay out UI components called controls.
CHAPTER 5  Displaying forms and capturing user input

Remember, you’re not restricted to the set of controls that come with Flex. Unlike HTML, which limits your UI controls to the HTML specification and browser implementation, Flex encourages you to extend an existing control to add more functionality, or create your own, completely new, UI controls from scratch.

One of the teaching approaches this book employs is to show many ways of doing the same thing. From example to example, we’ll use an idea from the previous one, but then change a portion of it to demonstrate an alternative. The alternatives aren’t necessarily any better, they’re just different, but more importantly help you to think with the ActionScript mentality.

A core piece to that understanding is the invaluable id attribute, a handle to an MXML component that allows you to access the values contained by the id attribute.

5.1 The id attribute

You were first introduced to the id attribute in chapter 2, but it is worth taking the time to review and expand what you learned. The id attribute can be used on any component and you can access it the same as you would any other variable. It gives you a mechanism to uniquely name any instances of a component, which allows you to refer to the component explicitly using its unique identifier.

HTML Form tags also have id attributes that are similar in their role and use. Unlike HTML, MXML does not require you to use functions like JavaScript’s getElementById() to access an id.

When it comes to building Forms, the id property will be the key to retrieving the values from target components.

5.2 Text controls

The basic purpose for the UI of any application is to display text. Flex provides a variety of components that allow you to capture and display textual information. Table 5.1 presents these controls and their descriptions.

<table>
<thead>
<tr>
<th>Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>A simple control that displays textual information. Its goal is to appear as if the text is embedded in the application, and not some arbitrary visual element floating around the window. This means it has no scrollbars (other components enable scrollbars if they run out of space), it is borderless, and the background is transparent (by default). Textual data stored by Text can always wrap, and it supports HTML text.</td>
</tr>
<tr>
<td>Label</td>
<td>The same as Text, except the textual data it contains doesn’t wrap. Common uses for this include the title to a section of your application, or the label of an input field.</td>
</tr>
<tr>
<td>TextInput</td>
<td>Presents a single-line text entry field. This control is similar to the &lt;input type=&quot;text&quot;&gt; tag in HTML.</td>
</tr>
</tbody>
</table>

Licensed to Manning Marketing <mkt@manning.com>
Table 5.1  Text controls for displaying and capturing text (continued)

<table>
<thead>
<tr>
<th>Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TextArea</td>
<td>Presents a multiple-row text entry field. Similar to the <code>&lt;textarea&gt;</code> tag in HTML.</td>
</tr>
<tr>
<td>RichTextEditor</td>
<td>As the name implies, a robust editor that allows the user to format their text. This includes parameters like color, font, and text size. HTML does not offer a similar feature to RichTextEditor natively.</td>
</tr>
</tbody>
</table>

Listing 5.1 demonstrates each of these in a single application in which we’ll put together our first form. Our form will employ a Panel component to automatically lay out all the controls.

Listing 5.1  This code makes use of all of the text-based control components

```xml
<?xml version="1.0"?>
  <mx:Script>
    public function showMsg(msg:String):void
    {
      mx.controls.Alert.show(msg);
    }
  </mx:Script>

  <mx:Panel title="Profile" width="400" height="400">
    <mx:Label text="Enter your name" fontWeight="bold" />
    <mx:TextInput id="yourName" width="250" valueCommit="showMsg(yourName.text)" />
    <mx:Text text="Profile Summary" />
    <mx:TextArea id="aboutYou" textAlign="center" width="250" valueCommit="showMsg(aboutYou.text)" />
    <mx:Label text="Enter your profile" fontWeight="bold" color="#ff0000" />
    <mx:RichTextEditor id="rte" height="150" valueCommit="showMsg(rte.text)" />
  </mx:Panel>
</mx:Application>
```

Try compiling and running the application and you’ll see a display similar to figure 5.1.

Now let’s look at what we’ve created and how it works. The Label component is used to display a descriptive label above each input control. To give you a sense of what Label can do we’ve modified some of its styling properties. When a user enters a value and commits it—for example, by clicking Enter or Tab, or an element such as a Submit button—Flex will recognize this event and run the `showMsg()` function as instructed.
These text fields can be used to capture a broad range of data. You could even use them to capture a date. This would likely be your natural inclination when coming from an HTML background because there isn’t a native date form input available. A better choice would be to use Flex’s Date controls.

### 5.3 Date controls

When using HTML, a common technique to capture date-related input is to use the form’s `<input type="text">` tag, paired with a calendar image displayed directly after the tag. When invoked, a calendar appears in a pop-up window. When a user selects a date from the calendar, a bit of JavaScript is executed to copy the selected value to the input field.

Fortunately, there’s an easier way to capture calendar-related data in Flex, which offers two date-based fields (listing 5.2).

<table>
<thead>
<tr>
<th>Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateChooser</td>
<td>Displays a small calendar similar to most HTML-based calendar selection widgets. A noteworthy feature is you can control the display of ranges of dates, as well as determine which ranges are selectable. This is quite useful if you wanted to block out dates, such as holidays.</td>
</tr>
<tr>
<td>DateField</td>
<td>The input control that appears similar to the <code>&lt;input type=&quot;text&quot;&gt;</code> tag/calendar image combination used in HTML. It allows the user to enter a date, or, if the icon is selected, it displays the DateChooser control.</td>
</tr>
</tbody>
</table>

The code in listing 5.2 demonstrates how to use both of these components.

```xml
<?xml version="1.0"?>
  <mx:Script>
    public function showMsg(msg:String):void {
      mx.controls.Alert.show(msg);
    }
  </mx:Script>
  <mx:Panel title="Profile" width="400" height="400">
    <mx:DateField id="thisDateField" change="showMsg(thisDateField.selectedDate.toString())" text="12/05/2008"/>
    <mx:DateChooser id="thisDateChooser" maxYear="2010" minYear="2006" selectedDate="{new Date(2008,10, 15)}">
      change="showMsg(thisDateField.selectedDate.toString())" /
    </mx:DateChooser>
  </mx:Panel>
</mx:Application>
```
Notice how the id property’s set value is used to access the value of that component? The intention was merely to show you how the id property could be used; later in the chapter we’ll show alternative ways to accomplish the same thing.

Did you also notice how the change event is used instead of valueCommit? The difference between these two specific events is valueCommit is triggered when the user performs an action that indicates she’s completed setting the value (for example, pressing the Enter key). The change event responds as the value changes, so feedback occurs in real time with each keystroke.

Compile and run the application and you’ll see the result of Flex’s two date-based components illustrated in figure 5.2.

NOTE Why can’t you default the date on the DateChooser to a simple text representation of a date?

If you pull up the Flex API Reference for this object you’ll see that selectedDate expects a Date object—in our example we’re creating one on the fly, and inline. The DateField also supports this attribute.

Just as date-based controls can be used more effectively to acquire calendar data than a text input box, Flex also has controls that are specific to capturing numeric values.

5.4 Numeric controls

Numeric controls are a great example of how the benefits of Flex move beyond HTML. Nothing similar to Numeric controls exists in the land of traditional web applications.

The user input controls presented in table 5.3 allow you to capture numeric data from the user. As you would expect, they can be used for the more obvious tasks—such as obtaining someone’s age—but you can also use them to resize an image in real time, or to filter a set of data.

Table 5.3 Controls used for capturing numeric values

<table>
<thead>
<tr>
<th>Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumericStepper</td>
<td>A simple control that lets the user increment and decrement values. You can specify the minimum and maximum allowed values, as well as the unit size of the increment/decrement step.</td>
</tr>
</tbody>
</table>
CHAPTER 5  Displaying forms and capturing user input

Let’s put these controls to work in listing 5.3 and capture some numbers.

Listing 5.3 Using Flex’s Numeric input controls to capture values

```xml
<?xml version="1.0"?>
  <mx:Script>
    public function showMsg(msg:String):void
    {
      mx.controls.Alert.show(msg);
    }
  </mx:Script>
  <mx:Panel title="Profile" width="360" height="240" layout="horizontal">
    <mx:VBox>
      <mx:Text fontWeight="bold" text="How many kids do you have?">
        <mx:NumericStepper id="kids"
          minimum="0" maximum="15"
          stepSize="1"
          change="showMsg(kids.value.toString())"/>
      </mx:Text>
      <mx:Text fontWeight="bold" text="How long is your commute (mins)?">
        <mx:HSlider id="commuteTimeRange" minimum="0" maximum="180"
          snapInterval="5"
          tickInterval="15"
          labels="[0 mins, 180 mins]"
          thumbCount="2"
          change="showMsg(commuteTimeRange.values.toString())"/>
      </mx:Text>
      <mx:Text fontWeight="bold" text="How tall are you (cm)?">
        <mx:VSlider id="yourHeight" minimum="0" maximum="300"
          tickInterval="50" snapInterval="1"
          labels="[0, 50, 100, 150, 200, 250, 300]"
          change="showMsg(yourHeight.values.toString())"/>
      </mx:Text>
    </mx:VBox>
  </mx:Panel>
</mx:Application>
```

Let’s put these controls to work in listing 5.3 and capture some numbers.

Table 5.3 Controls used for capturing numeric values (continued)

<table>
<thead>
<tr>
<th>Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSlider</td>
<td>Allows the user to slide what’s known as a thumb horizontally along a track. You can control the minimum and maximum allowed values, snapInterval increments (positions on the slide to which the thumb will snap), and visible increments (called ticks). You also have the option to set up more than one thumb if you want to allow the user to specify multiple values (for example, a range).</td>
</tr>
<tr>
<td>VSlider</td>
<td>Identical to the HSlider, except the orientation of the track is vertical.</td>
</tr>
</tbody>
</table>

Let’s put these controls to work in listing 5.3 and capture some numbers.
We can select in units of 5 and display markers every 15 units. The display is two evenly spaced labels (one at each end). In the vertical slider, the markers are further apart than in the horizontal slider. Finally, we display seven evenly spaced labels.

After compiling and running the application, you'll see the results shown in figure 5.3. This is a collection of user input controls that you certainly have never seen in HTML—they're not even that common among desktop applications.

Many developers fail to take full advantage of these components, usually because they come from an HTML background and are not used to this level of utility. This is unfortunate, because you can use numeric-based components in ways more creative than simply capturing a value. You can use that value to provide real-time control to your end user. As another example, if you're designing a dashboard, you can have the display change in real time as the user moves a slider back and forth—effectively filtering data in and out.

5.5 **Buttons**

A whole section on buttons? When it comes to Flex, the answer is a resounding, yes! Flex goes beyond the plain old button offered by HTML and adds a collection of interesting options (see table 5.4). Buttons are one of stalwarts of GUIs, but they can be a bit boring. Flex takes them to a higher level with a number of variations.

Table 5.4 Flex offers an expanded range of button components

<table>
<thead>
<tr>
<th>Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
<td>The standard, all-purpose button for accepting a mouse click interaction. Flex buttons have built-in support that let you easily add an image into the button. In HTML you can use an image to create a pseudo-Text+Image button, but your text will be static. With Flex, the text remains dynamic.</td>
</tr>
<tr>
<td>ButtonBar</td>
<td>Dynamically generates a series of buttons based on an array. You could do the same thing by dynamically generating Buttons on your own, but not only is a ButtonBar easier to implement, it adds some flair by rounding the edges of the outer most buttons to lend the appearance of a single Menu bar, split into sections.</td>
</tr>
<tr>
<td>LinkButton</td>
<td>Flex's version of an HTML link. If you pull up the class reference for this object (in the Flex API Reference) you'll see it is a descendent of the Button, but styled and extended to behave like an HTML link by having a transparent background and no border.</td>
</tr>
<tr>
<td>LinkBar</td>
<td>Similar to the relationship between a Button and a ButtonBar, the LinkBar is an easy way to create a series of LinkButtons.</td>
</tr>
</tbody>
</table>
Listing 5.4 presents a look at how to invoke all these button choices.

Listing 5.4  A collection of Button-based components in action

```xml
<?xml version="1.0"?>

    <![CDATA[
    public function showMsg(msg:String):void
    {
        mx.controls.Alert.show('You just clicked on ' + msg);
    }
    ]]>

    <mx:Panel title="Profile" width="360" height="240" layout="vertical">
        <mx:VBox>
            <mx:Button id="thisBtn" label="Button" click="showMsg('button')"/>
            <mx:LinkButton id="thisLinkBtn" label="LinkButton" click="showMsg('linkbutton')"/>
        </mx:VBox>
        <mx:VBox>
            <mx:ButtonBar id="thisBtnBar" dataProvider="{["One","Two","Three"]}" itemClick="showMsg(event.label)"/>
            <mx:LinkBar id="thisLinkBar" dataProvider="{["One","Two","Three"]}" itemClick="showMsg(event.label)"/>
            <mx:ToggleButtonBar id="thisToggleBar" dataProvider="{["One", "Two", "Three"]}" itemClick="showMsg(event.label)"/>
        </mx:VBox>
    </mx:Panel>
</mx:Application>
```

Table 5.4  Flex offers an expanded range of button components (continued)

<table>
<thead>
<tr>
<th>Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ToggleButtonBar</td>
<td>Nearly identical to a ButtonBar, except it persists the selection—if you press one of the buttons it stays down (and highlighted) until you change to another button. Think of it as a radio button with a label inside.</td>
</tr>
<tr>
<td>PopUpButton</td>
<td>A dual-button that combines the functionality of two buttons—the left side acts like a normal Button; the right side invokes another UI object (anything derived from the UIComponent object). This is typically used to create a multiselection Button that looks as if it is a Button and a ComboBox merged into one.</td>
</tr>
<tr>
<td>PopUpMenuButton</td>
<td>A descendent of the PopUpButton, it is geared to specifically create a dropdown menu.</td>
</tr>
</tbody>
</table>

Licensed to Manning Marketing <mkt@manning.com>
Being an OOP language Flex tends to build upon itself, using some components as a foundation then creating variants on top of those. The Button family demonstrates this strategy quite well.

5.5.1 Bars of Buttons

ButtonBar, LinkBar, and ToggleButtonBar are such close siblings that from a mechanical perspective, aside from minor presentation differences, they’re the same thing (see figure 5.4). More important is their use ofdataProvider, which is a property that accepts array-like data to generate its display (see chapter 4 for a discussion on the Repeater and its dataProvider property).

As with many applications, Flex makes arrays available, but you can also take advantage of its more powerful cousin, ArrayCollection. In the following example, we used ActionScript shorthand to create a hard-coded array on the fly. An alternative method would be to create an ArrayCollection variable, and assign that variable to the dataProvider as follows:

```xml
<mx:Script>
import mx.collections.ArrayCollection;
[Bindable]
public var arData:ArrayCollection = new ArrayCollection(['One','Two','Three']);
</mx:Script>
<mx:ButtonBar id="thisBtnBar" dataProvider="{arData}"/>
```

The benefit of creating an ArrayCollection variable is the ability to decouple the logic it contains from the component that displays it. You can read in configuration data and populate variables accordingly.

NOTE What is the difference between click and itemClick? Not much. Flex applications are event-driven, and triggers cause an event to occur. Different components respond to all kinds of events. click refers to the mouse button being pressed (the trigger) on a Button component; itemClick is used to capture the selection from a list of items.

The previous examples have covered all of the buttons, except two: the PopUpButton and the PopUpMenuButton. As self-explanatory as they appear, in order for you to make proper use of them, some more explanation will be required.

5.5.2 The PopUpButton and PopUpMenuButton

We’re not playing favorites by giving these two controls special attention. They’re considered special cases, and as such, they do need a more thorough explanation.

Listing 5.5 illustrates a PopUpMenuButton in action.
CHAPTER 5  Displaying forms and capturing user input

Notice that this example makes use of both the `click` and `itemClick` events. When you click the right secondary Button (figure 5.5), a drop-down menu will appear, prompting you to select an item. When a selection is made, the `itemClick` event will be triggered. But if you click the left side of the primary Button, the `itemClick` and `click` events will fire, one after the other.

This is useful when you need to differentiate between these two user interactions. For example, let’s assume the drop-down menu defined in listing 5.5 contained a list of credit card brands (for example, Visa, MasterCard, and American Express). When the user changes the selection (`itemClick`) the credit card type is stored to a variable, but when the primary Button (`click`) is clicked, Flex will submit the Form.

Let's move on to the father of `PopupMenuButton`: the `PopupButton`. The `PopupButton` can perform the same tasks as `PopupMenuButton` but possesses broader capabilities, which, naturally takes more code to implement.

`PopupMenuButton` uses an element called `Menu` to generate the drop-down (another standard object with Flex). But `Menu` is the only thing `PopupMenuButton` can display.

`PopupButton` can display a greater variety of elements but it doesn’t default to any one in particular. This means it is considerably more flexible, but you do need to explicitly specify what `PopupButton` is going to show.

To produce the same result as listing 5.5 using `PopupButton`, the code would look like that in listing 5.6.

Listing 5.6  Using `PopupButton` to display an optional menu

```xml
<?xml version="1.0"?>
  <mx:Script>
    public function showMsg(msg:String):void {
      mx.controls.Alert.show('You just clicked on ' + msg);
    }
  </mx:Script>
  <mx:Panel width="100" height="100">
    <mx:PopupMenuButton id="menuBtn"
      dataProvider="{["One","Two","Three"]}
      click="showMsg('left side')"
      itemClick="showMsg('right side with ' + event.label)"/>
  </mx:Panel>
</mx:Application>
```

License to Manning Marketing <mkt@manning.com>
import mx.events.*;
import mx.controls.Menu;

public var menuItems:Object = 
[{
label:'One'},
{label:'Two'},
{label:'Three'}];

public var thisMenu:Menu = Menu.createMenu(null,menuItems,false);

public function handleItemClick(event:MenuEvent):void {
    menuBtn.label = event.label;
}

Here are three points to note about listing 5.6:

- In most examples, the variable containing the information to be populated is an array. In the previous example, we used an object. If you recall from chapter 3, there's not much difference between them. In this context you could have declared menuItems an array instead of an object and it would have worked as well.

- We’ve introduced creationComplete. This event instructs PopUpButton to call extra code when its creation is finished.

- This extra code tells our Menu object (thisMenu) to listen for the itemClick event, and if that occurs, to call our event handler (handleItemClick) to manage it.

That seems like a lot more work, so why bother? Taking a look at figure 5.6, the Button and drop-down menu appear nearly the same as those produced by PopUpMenuButton demonstrated earlier. Is it worth the effort? Yes. Unlike with PopUpMenuButton, the label in the Button doesn’t default to the first item in the list.

The main difference between using PopUpButton and PopUpMenuButton is with PopUpButton you’re not restricted to using it for menus, you can make use of any visual element to interact with the user. If all you want is the drop-down menu effect, go with the PopUpMenuButton. If you need more control over the options you want to present the user, leverage the flexibility of PopUpButton.

Along with buttons, the other common user input paradigm (in both desktop and web applications) is a picklist, which provides a selection list from which the user can choose one or more items.
5.6 Picklist controls

Picklists are controls that represent everything else—they present lists of options from which you can choose, and we’ve listed them in table 5.5.

Table 5.5 Flex’s picklist controls

<table>
<thead>
<tr>
<th>Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CheckBox</td>
<td>The generic CheckBox control (identical in behavior to an HTML CheckBox). It is a descendent of the Button, except it is designed specifically for the purpose of being checked and unchecked. The most common use for a CheckBox is to provide a series of options from which the user can check (select) one or more of those options.</td>
</tr>
<tr>
<td>RadioButton</td>
<td>Similar to the CheckBox, except when grouped as a range of choices, the user can select only one. The RadioButton is also a descendent of Button.</td>
</tr>
<tr>
<td>ComboBox</td>
<td>This drop-down menu control displays the currently selected option, but when clicked drops a display list of options from which to choose. ComboBox is the same as a &lt;select&gt; box in HTML, except unlike its HTML equivalent, you can allow the user to free-form edit the selected text. Although it appears to be based on Button, it is not a descendent.</td>
</tr>
<tr>
<td>ColorPicker</td>
<td>A sibling to ComboBox. It drops down a color menu allowing the user to select from a color palette. You could use this to allow users to customize their desktop, or as part of the application workflow (for example, a color selector with which visitors to an automobile manufacturer’s website can view a vehicle’s color options in real time).</td>
</tr>
</tbody>
</table>

To get an idea of what these look like in action we assembled a program fashioned on the concept of a user profile form (listing 5.7).

Listing 5.7 Examples of picklist controls

```xml
<?xml version="1.0"?>
    <mx:Script>
        public function showMsg(msg:String):void
        {
            mx.controls.Alert.show(msg);
        }
    </mx:Script>
    <mx:Panel width="400" height="150" title="Profile">
        <mx:HBox>
            <mx:Label text="Your hobbies:"/>
            <mx:CheckBox id="cbVideoGames" label="Video Games" click="showMsg('Video Games is ' + cbVideoGames.selected)="/n"
            <mx:CheckBox id="cbFishing" label="Fishing" click="showMsg('Fishing is ' + cbFishing.selected)="/n"
        </mx:HBox>
    </mx:Panel>
</mx:Application>
```
We present two CheckBoxes and display their value when clicked 1. In our next step 2 we create two grouped RadioButtons, which triggers two pop-up windows. First click on the RadioButton will fire, followed by itemClick. RadioButtonGroup is optional—use it when you need to call a single function regardless of RadioButton.

After compiling and running the application you’ll see something similar to that of figure 5.7 which contains various questions using these picklist controls.

In comparing these components to their HTML forms equivalents, there’s not much of a departure in their general behavior—although the color picker stands apart as something completely different.

![Figure 5.7](image_url)
NOTE  The term picklist isn’t an official term associated with Flex, it is something we use as a category for these particular controls.

Now having covered all of the user input components (controls) along with usage examples, we need to take a deeper look at what your options are when interacting with the user, and accessing any values they have input.

5.7 **Accessing the control’s value**

As with any programming language, there are many ways to carry out the same task. In the previous examples, the most straightforward techniques were used to make it easy to see what’s going on in the background. Be aware that for each of these examples, there are other (and perhaps better) ways to execute them.

5.7.1 **Passing values to a function**

The previous examples contain many instances of passing values to a function. Although my intention was to keep the examples streamlined and easy to understand, passing values to independent functions is a good technique to employ, as you relieve the logic that processes data from needing to know where the data came from.

This will allow you to reuse that function and to change the implementation—for example, switching from a RadioButton to a CheckBox) doesn’t faze the function at all. This is what you observed with the showMsg function:

```java
public function showMsg(msg:String):void
{
    mx.controls.Alert.show(msg);
}
```

When we call a function to which we pass the value, it is in response to a trigger that caused the event to occur. Each event has an event object to go along with it, which, by means of the `currentTarget` property, contains all kinds of goodies, including where the event came from.

This is what we’re doing in the following code as we use `currentTarget` (in this case a reference to the RadioButtonGroup) to extract the `selectedValue` of the RadioButtonGroup.

```xml
<mx:RadioButtonGroup
   itemClick="showMsg(event.currentTarget.selectedValue)"/>
```

But if we assign that control an `id`, we can reference it using that instead:

```xml
<mx:RadioButtonGroup id="Spam"
   itemClick="showMsg(Spam.selectedValue.toString())"/>
```

Passing simple values is common in any language, but Flex, which is event–based, provides an opportunity to tap into the event itself.

5.7.2 **Passing events to a function**

An inverse approach is to put more responsibility on the function by increasing its awareness of the data’s origin.
To do that, we can pass the entire event object as shown in listing 5.8.

**Listing 5.8 When the user clicks a control, you can pass the entire event object**

```xml
<mx:Script>
    import mx.events.ItemClickEvent;
    public function showMsg(anEvent:ItemClickEvent):void
    {
        mx.controls.Alert.show(anEvent.currentTarget.selectedValue);
    }
</mx:Script>
<mx:RadioButtonGroup id="Spam" itemClick="showMsg(event)"/>
```

As you can see, it is more involved, but it does simplify matters for the control that invokes the function. It only needs to pass the event object; the event handler function can independently process the event in whatever fashion is required. The key aspect to note is you’ll need to know the type of event that’s being passed and execute an import of the class 1.

You can determine this by displaying the component or control’s class reference in the Flex API Reference (see chapter 2).

**TIP** The quickest way to access the Flex API class reference (in this case to show the definition of the control) is to click the control in question then press Shift+F2.

For example, in listing 5.8, we can determine the event was an `ItemClickEvent` object by following these steps:

1. In your code, click RadioButtonGroup then press Shift+F2. The API Reference for this component will display.
2. We’re using events, so scroll down to the events section.
3. Select `ItemClick` from the list (see figure 5.8).
4. The API Reference for the event itself will appear. The class will be listed under the event object type area (see figure 5.9). In this example it shows we need the `mx.events.ItemClickEvent` class.

Figure 5.8 To learn more about a particular event, such as which class to import, select the event from the events list in the API Reference for a given component.

Figure 5.9 When looking at an event in the API Reference, look at the object type to determine which class to import.
Initially, this approach takes the most amount of work to implement, but if you wanted a really fast route, your function can directly access properties of the components to retrieve the values.

5.7.3 **Accessing properties directly**

Finally, the most tightly coupled approach. This requires your function to have very specific knowledge of what it is accessing. This, of course, limits your function’s reusability. Worse case, if the property it is accessing changes, the code could break down.

On the flip side, it is a very convenient approach that doesn’t require passing any data (listing 5.9).

**Listing 5.9** This function accesses the value of the component directly

```xml
<mx:Script>
  public function showMsg():void
  {
    mx.controls.Alert.show(Spam.selectedValue.toString());
  }
</mx:Script>
<mx:RadioButtonGroup id="Spam" itemClick="showMsg()"/>

In this scenario, you can respond to an event on the input control if you want, but since you’re accessing the value of the control directly you can access it at any time, such as when a submit Button is pressed.

You can also use this technique in combination with any of the others previously mentioned. In listing 5.10 an event handler is used to validate the input value using a highly decoupled function that doesn’t know where the value came from, at the same time the submit Button is clicked on another function used to explicitly access a particular instance of the TextInput.

**Listing 5.10** Combining techniques to access values of a control

```xml
<?xml version="1.0"?>
  <mx:Script>
    <![CDATA[
      import mx.controls.Alert;
      public function checkValue(inputValue:String):void
      {
        if(inputValue.length < 5)
          Alert.show("Are you sure there's not that much new?");
      }
    ]]>
    public function submitClicked():void
    {
      Alert.show("User says:" + whatsnew.text + " is new.");
    }
  </mx:Script>
  <mx:Panel title="Profile" width="400" height="400">
    <mx:Label text="What's new?"/>
  </mx:Panel>
</mx:Application>
```
TIP Notice that <![CDATA[ ... ]]> was used inside the <mx:Script> block. This is because we’re using a less than (<) character in ActionScript. In order to keep the examples short we’ve been omitting this declaration, but it is always a good idea to declare it in your ActionScript blocks (refer to chapter 3 for a refresher).

Clearly, there’s quite a bit of flexibility, but the question is which technique is the way to go?

5.7.4 Which approach to use?

Although purists may be adamant about how to implement functions that need to work with the values of controls, there’s no hard and fast rule as to which approach works best. And as with any other programming language, the more time you invest upfront making your code reusable and easier to maintain, the less it will cost you in the long run.

In the end, there is no wrong way. It is a matter of evaluating factors, such as the size of the application, its life expectancy, and other issues. For example, if the goal of the application is to be a quick, one-off project, or proof of concept, it may not warrant investing a lot of time concerning yourself with reusability issues.

5.8 Summary

Flex comes with a number of interface components known as controls. Some controls have equivalents in HTML, and many go far beyond their HTML equivalents.

For example, the ColorPicker, Sliders, and sheer variety of Buttons offer tremendous freedom to be creative when building interfaces and forms.

Keep in mind, unlike HTML, where a Form encapsulates a collection of form inputs, in Flex a form does nothing more than help lay out components. Those components don’t necessarily need to be inside a <mx:Form> container.

There are a number of ways you can retrieve the value of user selections; you can do it in real time by leveraging events, or you can access the value as needed by using the id property of the component.

Capturing user input is one thing, but what about validating it? For instance, you may need to build an application that requires a user to enter a password made up of at least five characters. In the next chapter we’ll take a look at how Flex validators can be used to undertake this, and much more.