

*Object-Oriented Development with ActionScript 2.0*

# Essential ActionScript 2.0



**O'REILLY®**

*Colin Mook*

# Using Components with ActionScript 2.0

In Chapter 11, we learned how to structure a basic OOP application in ActionScript 2.0. In this chapter, we'll see how to create a GUI application based on that structure. As usual, you can download the sample files discussed in this chapter from <http://moock.org/eas2/examples>.

## Currency Converter Application Overview

Our example GUI application is a simple currency converter. Figure 12-1 shows the components used in our currency converter interface (Button, ComboBox, Label, TextArea, and TextInput). The user must specify an amount in Canadian dollars, select a currency type from the drop-down list, and click the Convert button to determine the equivalent amount in the selected currency. The result is displayed on screen.



Figure 12-1. The currency converter application

We'll place the assets for our application in the following directory structure, which mirrors the structure we used in Chapter 11. Note that the *deploy* and *source* folders are both subdirectories of *CurrencyConverter* and that *org/moock/tools* is a subdirectory of the *source* folder:

```
CurrencyConverter/
  deploy/
```

```
source/  
  org/  
    moock/  
      tools/
```

Our application's main Flash document is named *CurrencyConverter.fla*. It resides in *CurrencyConverter/source*. To create the *CurrencyConverter.fla* file, we'll copy the file *AppName.fla* (which we created in Chapter 11) to the *CurrencyConverter/source* directory, and rename the file to *CurrencyConverter.fla*. That gives *CurrencyConverter.fla* the basic structure it needs, including a class preloader on frames 2 and 3 and a frame labeled *main*, on which we'll start the application.

Our application's only class is *CurrencyConverter*. It is stored in an external *.as* class file named *CurrencyConverter.as*, which resides in *CurrencyConverter/source/org/moock/tools*. Our exported application (a Flash movie) is named *CurrencyConverter.swf*. It resides in *CurrencyConverter/deploy*.

Now let's take a closer look at each of these parts of our application.

## Preparing the Flash Document

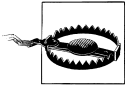
Our *CurrencyConverter* class instantiates all the components needed for our application at runtime. Even though we create instances of components at runtime, Flash requires us to add the components to the *CurrencyConverter.fla*'s Library during authoring. Unfortunately, Flash does not allow us to simply drag the required components from the Flash Components panel to *CurrencyConverter.fla*'s Library. Instead, to add a component to the Library, we must drag an instance of it to the Stage. Although component instances can be left on the Stage of a *.fla* file, that development style is not our current focus. So we'll delete the instances from the Stage; however, Flash leaves a copy of the underlying component in the Library (which was our original goal).

## Adding Components to the Document

Here are the steps to follow to add the Button component to *CurrencyConverter.fla*'s Library. To add the ComboBox, Label, TextArea, and TextInput components to the Library, follow the same steps, choosing the appropriate component instead of Button.

1. With *CurrencyConverter.fla* open in the Flash authoring tool, select Window → Development Panels → Components.
2. In the Components panel, in the folder named UI Components, click and drag the Button component to the Stage. The Button component appears in the Library.
3. Select and delete the Button instance from the Stage.

If we were working in a brand new *.fla* file, our components would now be ready for runtime instantiation. However, recall that we're working with the basic Flash document we created in Chapter 11, which exports its classes at frame 10 and displays a preloading message while those classes load. Because of this preloading structure, our components would not currently work if we attempted to use them! We need to integrate the components into our preloading structure.



When a document's ActionScript 2.0 classes are exported on a frame later than frame 1, components will not work in that document unless they are loaded *after* the class export frame!

To load our components after frame 10, we must set them to not export on frame 1, and then we must place a dummy instance of each component on stage after frame 10. The dummy instance is not used; it merely forces the component to load.

Here are the steps we follow to prevent the Button component from being exported on frame 1. To prevent the remaining components from being exported on frame 1, follow the same steps, choosing the appropriate component instead of Button.

1. Select the Button component in the Library.
2. From the Library's pop-up Options menu in the top-right corner of the panel, select Linkage.
3. In the Linkage Properties dialog box, under Linkage, uncheck the Export in First Frame checkbox.
4. Click OK.

When a component's Export in First Frame option is unchecked, the component is not compiled with the movie unless an instance of the component is placed on the document's timeline. The component loads at the frame on which an instance of it is placed. But the component initialization process requires a movie's ActionScript 2.0 classes to be available *before* the component is exported. Hence, in our *CurrencyConverter.fla* document, we'll place an instance of each of our components on frame 12, two frames after our document's ActionScript 2.0 classes are exported. To store the dummy component instances, we'll create a new timeline layer and keyframe, as follows:

1. Select Insert → Timeline → Layer.
2. Double-click the new layer and rename it **load components**.
3. Select frame 12 in the *load components* layer.
4. Select Insert → Timeline → Keyframe.
5. Select frame 13 in the *load components* layer.
6. Select Insert → Timeline → Keyframe. This second keyframe prevents the dummy component instances from showing up in our application. We need them only for loading; the *CurrencyConverter* class handles the actual creation of component instances in our application.

With our component-loading keyframe prepared, we can now place a dummy instance of each component on our document's timeline, as follows:

1. Select frame 12 in the *load components* layer.
2. From the Library, drag an instance of each component to the Stage.
3. Optionally use the Component Inspector panel (Window → Development Panels → Component Inspector) to add dummy text to each component instance indicating that it is not used, as shown in Figure 12-2. Consult Flash's online help for instructions on setting component parameters via the Component Inspector.

Figure 12-2 shows how our document's timeline and Stage look with frame 12 of the *load components* layer selected.

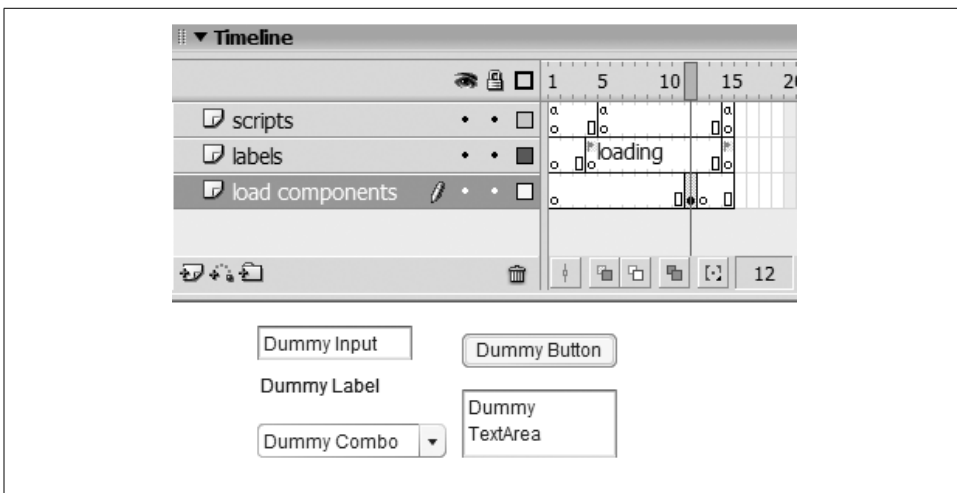


Figure 12-2. *CurrencyConverter.fla*'s timeline and Stage

## Starting the Application

In Chapter 11, we saw how to start an OOP application by invoking the *main()* method of the application's primary class (i.e., the class and method that we design to be the program launch point). We'll start our currency converter by invoking *main()* on our application's primary (indeed, only) class, *CurrencyConverter*. The *main()* call comes on frame 15 of the *scripts* layer in *CurrencyConverter.fla*'s timeline, after our classes and components have been preloaded. Here's the code:

```
import org.mock.tools.CurrencyConverter;
CurrencyConverter.main(this, 0, 150, 100);
```

Notice that the *import* statement allows us, in the future, to refer to the *CurrencyConverter* class by its name without typing its fully qualified package path. As implied by the preceding code, our class's *main()* method expects four

parameters: the movie clip to hold the currency converter (this) and the depth, horizontal position, and vertical position—0, 150 and 100, respectively—at which to display the converter within the clip.

Our *CurrencyConverter.fla* file is now ready. We can now turn our attention to the class that creates and manages the currency converter application itself, *CurrencyConverter*.

## The CurrencyConverter Class

The *CurrencyConverter* class's three main duties are to:

- Provide a method that starts the application (*main()*)
- Create the application interface
- Respond to user input

Before we examine the specific code required to perform those duties, you should skim the entire class listing, presented in Example 12-1. For now, you don't need to read the code too carefully; we'll study it in detail over the remainder of this chapter.

*Example 12-1. The CurrencyConverter class*

```
// Import the package containing the Flash UI components we're using.
import mx.controls.*;

// Define the CurrencyConverter class, and include the package path.
class org.moock.tools.CurrencyConverter {
    // Hardcode the exchange rates for this example.
    private static var rateUS:Number = 1.3205; // Rate for US dollar
    private static var rateUK:Number = 2.1996; // Rate for pound sterling
    private static var rateEU:Number = 1.5600; // Rate for euro

    // The container for all UI elements
    private var converter_mc:MovieClip;

    // The user interface components
    private var input:TextInput; // Text field for original amount
    private var currencyPicker:ComboBox; // Currency selection menu
    private var result:TextArea; // Text field for conversion output

    /**
     * CurrencyConverter Constructor
     *
     * @param target The movie clip to which
     *               converter_mc will be attached.
     * @param depth The depth, in target, on which to
     *               attach converter_mc.
     * @param x The horizontal position of converter_mc.
     * @param y The vertical position of converter_mc.
     */
    public function CurrencyConverter (target:MovieClip, depth:Number,
```

Example 12-1. The `CurrencyConverter` class (continued)

```

        x:Number, y:Number) {
    buildConverter(target, depth, x, y);
}

/**
 * Creates the user interface for the currency converter
 * and defines the events for that interface.
 */
public function buildConverter (target:MovieClip, depth:Number,
        x:Number, y:Number):Void {
    // Store a reference to the current object for use by nested functions.
    var thisConverter:CurrencyConverter = this;

    // Make a container movie clip to hold the converter's UI.
    converter_mc = target.createEmptyMovieClip("converter", depth);
    converter_mc._x = x;
    converter_mc._y = y;

    // Create the title.
    var title:Label = converter_mc.createClassObject(Label, "title", 0);
    title.autoSize = "left";
    title.text = "Canadian Currency Converter";
    title.setStyle("color", 0x770000);
    title.setStyle("fontSize", 16);

    // Create the instructions.
    var instructions:Label = converter_mc.createClassObject(Label,
        "instructions",
        1);

    instructions.autoSize = "left";
    instructions.text = "Enter Amount in Canadian Dollars";
    instructions.move(instructions.x, title.y + title.height + 5);

    // Create an input text field to receive the amount to convert.
    input = converter_mc.createClassObject(TextInput, "input", 2);
    input.setSize(200, 25);
    input.move(input.x, instructions.y + instructions.height);
    input.restrict = "0-9.";
    // Handle this component's enter event using a generic listener object.
    var enterHandler:Object = new Object();
    enterHandler.enter = function (e:Object):Void {
        thisConverter.convert();
    }
    input.addEventListener("enter", enterHandler);

    // Create the currency selector ComboBox.
    currencyPicker = converter_mc.createClassObject(ComboBox, "picker", 3);
    currencyPicker.setSize(200, currencyPicker.height);
    currencyPicker.move(currencyPicker.x, input.y + input.height + 10);
    currencyPicker.dataProvider = [
        {label:"Select Target Currency", data:null},
        {label:"Canadian to U.S. Dollar", data:"US"},
    ]
}
```

Example 12-1. The CurrencyConverter class (continued)

```
        {label:"Canadian to UK Pound Sterling", data:"UK"},
        {label:"Canadian to EURO", data:"EU"}]];

// Create the Convert button.
var convertButton:Button = converter_mc.createClassObject(Button,
    "convertButton",
    4);
convertButton.move(currencyPicker.x + currencyPicker.width + 5,
    currencyPicker.y);
convertButton.label = "Convert!";
// Handle this component's events using a handler function.
// As discussed later under "Handling Component Events," this technique
// is discouraged by Macromedia.
convertButton.clickHandler = function (e:Object):Void {
    thisConverter.convert();
};

// Create the result output field.
result = converter_mc.createClassObject(TextArea, "result", 5);
result.setSize(200, 25);
result.move(result.x, currencyPicker.y + currencyPicker.height + 10);
result.editable = false;
}

/**
 * Converts a user-supplied quantity from Canadian dollars to
 * the selected currency.
 */
public function convert ():Void {
    var convertedAmount:Number;
    var origAmount:Number = parseFloat(input.text);
    if (!isNaN(origAmount)) {
        if (currencyPicker.selectedItem.data != null) {
            switch (currencyPicker.selectedItem.data) {
                case "US":
                    convertedAmount = origAmount / CurrencyConverter.rateUS;
                    break;
                case "UK":
                    convertedAmount = origAmount / CurrencyConverter.rateUK;
                    break;
                case "EU":
                    convertedAmount = origAmount / CurrencyConverter.rateEU;
                    break;
            }
            result.text = "Result: " + convertedAmount;
        } else {
            result.text = "Please select a currency.";
        }
    } else {
        result.text = "Original amount is not valid.";
    }
}
```

Example 12-1. The *CurrencyConverter* class (continued)

```
// Program point of entry
public static function main (target:MovieClip, depth:Number,
                             x:Number, y:Number):Void {
    var converter:CurrencyConverter = new CurrencyConverter(target, depth,
                                                            x, y);
}
}
```

## Importing the Components' Package

Our *CurrencyConverter* class makes many references to various component classes (*Button*, *ComboBox*, *Label*, etc.). The component classes we need reside in the package *mx.controls*. Throughout our *CurrencyConverter* class, we could refer to the components by their fully qualified names, such as *mx.controls.Button* or *mx.controls.ComboBox*. That obviously gets tedious, so prior to defining our *CurrencyConverter* class, we import the entire *mx.controls* package, as follows:

```
import mx.controls.*;
```

Once the *mx.controls* package is imported, we can refer to a component class such as *Button* without specifying its full package name. Note that not all component classes reside in the *mx.controls* package. For example, the classes for containers such as *Window* and *ScrollPane* reside in the *mx.containers* package. To determine the package for a component class, consult its entry in Flash's built-in Components Dictionary (Help → Using Components → Components Dictionary).

For complete details on packages and the *import* statement, see Chapter 9.

## CurrencyConverter Properties

Our *CurrencyConverter* class defines two general categories of properties: class properties that specify the exchange rates for various currencies and instance properties that store references to some of the components in our user interface:

```
// The exchange rate properties
private static var rateUS:Number = 1.3205; // Rate for US dollar
private static var rateUK:Number = 2.1996; // Rate for Pound Sterling
private static var rateEU:Number = 1.5600; // Rate for euro

// The container for all UI elements
private var converter_mc:MovieClip;

// The user interface components
private var input:TextInput; // Text field for original amount
private var currencyPicker:ComboBox; // Currency selection menu
private var result:TextArea; // Text field for conversion output
```

For the sake of this example, the exchange rates in our application are permanently set in class properties. In a real currency converter, they'd most likely be retrieved at runtime from a dynamic, server-side source, as described in *Flash Remoting: The Definitive Guide* by Tom Muck (O'Reilly).

Notice that not all of our user interface components are stored in instance properties. After creation, some components (e.g., the Label components) need not be used again and, hence, are not stored in instance properties. When creating a component, we store a reference to it only if another method in the class needs to manipulate it later in the application.

## The `main()` method

The currency converter application's startup method is `CurrencyConverter.main()`:

```
public static function main (target:MovieClip, depth:Number,  
                             x:Number, y:Number):Void {  
    var converter:CurrencyConverter = new CurrencyConverter(target, depth,  
                                                            x, y);  
}
```

The `main()` method is a class method (i.e., declared *static*) because it is invoked once for the entire application and is not associated with a particular instance. The `main()` method uses a common Java convention to set things in motion: it creates an instance of the application's primary class, `CurrencyConverter`, and stores that instance in a local variable, `converter`. The instance manages the remainder of the application either by itself or, in larger applications, by creating and interacting with other classes. Notice that the application's primary class, `CurrencyConverter`, is also the class that defines the `main()` method. In fact, the `CurrencyConverter` class creates an instance of itself! This structure is both legitimate and common.

As we saw previously, `CurrencyConverter.main()` is invoked from the frame labeled `main` following the preloader. When the `main()` method exits, the local `converter` variable goes out of scope (i.e., no longer exists), and the reference to the `CurrencyConverter` instance in it is automatically deleted. However, the application continues to run because the movie clip and other components created by `CurrencyConverter` continue to exist on stage, even after `main()` exits. Physical assets on stage (e.g., movie clips and text fields) that are created at runtime exist until they are explicitly removed (or the clip in which they reside is removed), even if they are not stored in variables or properties.

As we'll see later, the `CurrencyConverter` instance created by `main()` is kept alive in the scope chain of the `buildConverter()` method. The movie clip and components created by `buildConverter()` retain a reference back to the `CurrencyConverter` instance via the scope chain. Without that reference, our application wouldn't be able to respond to component events.

## The Class Constructor

The *CurrencyConverter* class constructor is straightforward. It simply invokes *buildConverter()*, which creates our application's user interface:

```
public function CurrencyConverter (target:MovieClip, depth:Number,  
                                   x:Number, y:Number) {  
    buildConverter(target, depth, x, y);  
}
```

Notice that, like the *main()* method, the constructor passes on its arguments to another section of the program—in this case the *buildConverter()* method, which uses the arguments to create the application's interface.

## Creating the User Interface

As we've just learned, the currency converter interface is created by the *buildConverter()* method. To create the interface, *buildConverter()* instantiates user interface components and defines event handlers that dictate the interface's behavior.

You should already have skimmed the full listing for *buildConverter()* earlier in Example 12-1. Now let's study the method line by line.

The first line of code in *buildConverter()* may be unfamiliar to some programmers—it stores a reference to the current object in a local variable named *thisConverter*:

```
var thisConverter:CurrencyConverter = this;
```

Storing the current object in a local variable adds it to the *buildConverter()* method's scope chain. This not only allows the current object to be accessed by nested functions, but it keeps the current object alive for as long as those nested functions exist. As we'll see shortly, the event handlers in our application are implemented as nested functions; they use the *thisConverter* variable to access the current *CurrencyConverter* instance. For general information on this technique, see "Nesting Functions in Methods" in Chapter 4 and "Handling Component Events" later in this chapter.

### The interface container

Now we can move on to creating interface elements. First, we must create a container movie clip to which all our components are placed. We give the clip an instance name of *converter* and place it inside the specified target clip at the specified depth. Recall that the *target* and *depth* are supplied to the *main()* method in *CurrencyConverter.fla*, which passes them on to the *CurrencyConverter* constructor, which in turn passes them to *buildConverter()*.

```
converter_mc = target.createEmptyMovieClip("converter", depth);
```

Placing our application’s components in a single movie clip container makes them easy to manipulate as a group. For example, to move the entire group of components, we can simply set the `_x` and `_y` properties of the container clip:

```
converter_mc._x = x;  
converter_mc._y = y;
```

Again, the values `x` and `y` were originally supplied to the `main()` method on frame 15 of `CurrencyConverter.fla`.

Notice that our main container clip is stored in the instance property `converter_mc`. Storing the clip in a property allows it to be accessed outside the `buildConverter()` clip, for the purpose of, say, repositioning it or deleting it. In our case, we do not reposition or delete the main container, so we could, in theory, store it in a local variable instead of a property. In this case, however, we store it in the instance property `converter_mc` simply to make the class easier to enhance in the future.

### The title Label component

Now that our container movie clip is prepared, we can put our components into it. We start by giving our application a title using a Label component, which is used to display a single line of text on screen.

The following code creates a Label component named `title` inside `converter_mc` and places it on depth 0:

```
var title:Label = converter_mc.createClassObject(Label, "title", 0);
```

All components in our application are created with the `UIObject` class’s `createClassObject()` method. Its three parameters specify:

- The class of component to create
- The instance name of the component
- The depth of the component inside its parent movie clip or component

The `createClassObject()` method returns a reference to the newly created component. In the case of our `title` instance, we store that reference in a local variable named `title`. We use a local variable in this case because we have no need to access the `title` instance later. If, elsewhere in our application, we needed access to the `title` component, we’d store it in an instance property instead of a local variable.

Note that the instance name, “`title`” is used by convention for clarity, but we never use that name in our code. In our application, we refer to components via variables and properties only. In this case, we refer to the `title` instance by the variable name `title`—not by its instance name (which happens to be the same as the variable name). By convention, most of our component instance names match the variable or property name in which they are stored. However, nothing requires the instance name and the variable or property name to match; only the variable or property name matters to our application (the instance name is ignored).

Now take a closer look at the code we use to create the title instance:

```
converter_mc.createClassObject(Label, "title", 0);
```

Notice that we invoke `createClassObject()` on `converter_mc`, which is a `MovieClip` instance, despite the fact that the `MovieClip` class does not define a `createClassObject()` method! We can use `createClassObject()` on `MovieClip` instances because that method is added to the `MovieClip` class at runtime by the v2 component architecture (along with various other methods). Adding a new method or new property to a class at runtime works only when the recipient class is declared *dynamic*, as is the built-in `MovieClip` class.

Because we're accessing `createClassObject()` through `MovieClip`, we also don't need to worry that it returns an object of type `UIObject`, not type `Label` as required by our `title` variable. In theory, the following line of code should generate a type mismatch error because `createClassObject()` returns a `UIObject`, but `title`'s datatype is `Label`:

```
var title:Label = converter_mc.createClassObject(Label, "title", 0);
```

However, no error occurs because type checking is not performed on methods that are added to a class dynamically at runtime. When invoking `createClassObject()` on a component, to prevent a compiler error, you *must* cast the return value to the type of object you are creating. For example, here we cast the return of `createClassObject()` to the `Label` type, as is required if `converter_mc` holds a component instance instead of a `MovieClip` instance:

```
var title:Label = Label(converter_mc.createClassObject(Label, "title", 0));
```

Now that our application's title instance has been created, we can adjust its display properties, as follows:

```
title.autoSize = "left";           // Make the label resize automatically
title.text = "Canadian Currency Converter"; // Set the text to display
title.setStyle("color", 0x770000); // Set the text color
title.setStyle("fontSize", 16);    // Set the text size
```

For more information on setting the style of a single component or all the components in an application, see [Help → Using Components → Customizing Components → Using Styles to Customize Component Color and Text](#).

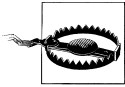
## The instructions Label component

To create the instructions for our application, we'll use another `Label` component similar to the application title created earlier. We store this `Label` instance in a local variable named `instructions`:

```
var instructions:Label = converter_mc.createClassObject(Label,
                                                         "instructions", 1);

instructions.autoSize = "left";
instructions.text = "Enter Amount in Canadian Dollars";
instructions.move(instructions.x, title.y + title.height + 5);
```

The last line of the preceding code places the instructions Label 5 pixels below the title Label.



Components can be positioned only with the `move()` method. Attempts to set a component's read-only `x` and `y` properties fail silently. Note that components support the properties `x`, `y`, `width`, and `height` (without an underscore). However, *MovieClip* instances support the properties `_x` and `_y`, `_width`, and `_height` (with an underscore).

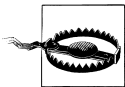
The `move()` method expects a new `x` coordinate as its first parameter and a new `y` coordinate as its second parameter. To leave the instructions instance's horizontal location unchanged, we specify its existing horizontal location (`instructions.x`) as the first argument to `move()`. The instructions instance's vertical location is calculated as the vertical position of the title instance, plus the title instance's height, plus a 5-pixel buffer.

### The input TextInput component

With the title and instructions completed, we can move on to components that accept user input. Our first input component is a `TextInput` instance, which lets a user enter a single line of text (in this case, an amount in Canadian dollars):

```
input = converter_mc.createClassObject(TextInput, "input", 2);
input.setSize(200, 25);
input.move(input.x, instructions.y + instructions.height);
input.restrict = "0-9.";
```

We use `setSize()` to make our `TextInput` component 200 x 25 pixels.



Components can be resized only with `setSize()`. Attempts to set a component's read-only `width` and `height` properties fail silently.

We use the `restrict` property to prevent the user from entering anything other than numbers or a decimal point character. Negative values and dollar signs are not allowed.

Next, we define what happens when the Enter key (or Return key) is pressed while the `TextInput` component instance has keyboard focus. When Enter is pressed, we want to convert the user-supplied value to the chosen currency. Conversion is performed by the `CurrencyConverter.convert()` method. To invoke that method when Enter is pressed, we register an *event listener object* to receive events from the `TextInput` instance. In this case, the event listener object is simply an instance of the generic *Object* class:

```
var enterHandler:Object = new Object();
```

On the generic *Object* instance, we define an *enter()* method that is invoked automatically by *input* (our *TextInput* instance) when the Enter key is pressed:

```
enterHandler.enter = function (e:Object):Void {
    thisConverter.convert();
}
```

Notice that the *enter()* method accesses the current *CurrencyConverter* instance via the *thisConverter* variable we defined earlier. That variable is accessible to the nested *enterHandler.enter()* method for as long as the *input* instance exists.

Finally, we register the generic *Object* instance to receive events from *input*, specifying "enter" as the type of event the instance is interested in receiving:

```
input.addEventListener("enter", enterHandler);
```

Although it's perfectly legitimate to use a generic *Object* instance to handle events, in more complex applications, an event listener object may well be an instance of a separate class. However, when a single event from a single user interface component generates a single response (as in our present case), it is also possible to handle the event with an *event handler function* rather than a generic *Object* instance. Later, we'll use an event handler function to handle events from our application's Convert button. We'll also learn why Macromedia discourages the use of event handler functions. In the specific case of *TextInput.enter()*, however, we're forced to use a listener object due to a bug in the *TextInput* class that prevents its event handler functions from working properly.

## The currencyPicker ComboBox component

Our application already has a place for the user to enter a dollar amount. Next, we add the drop-down menu (a *ComboBox* component) that lets the user pick a currency to which to convert. The *ComboBox* instance is stored in the *currencyPicker* property and is created, sized, and positioned exactly like previous components in the application:

```
currencyPicker = converter_mc.createClassObject(ComboBox, "picker", 3);
currencyPicker.setSize(200, currencyPicker.height);
currencyPicker.move(currencyPicker.x, input.y + input.height + 10);
```

To populate our *currencyPicker* component with choices, we set its *dataProvider* property to an array whose elements are generic objects with *label* and *data* properties. The value of each object's *label* property is displayed on screen as an option in the drop-down list. The value of each object's *data* property is used later by the *convert()* method to determine which currency was selected by the user.

```
currencyPicker.dataProvider = [
    {label:"Select Target Currency", data:null},
    {label:"Canadian to U.S. Dollar", data:"US"},
    {label:"Canadian to UK Pound Sterling", data:"UK"},
    {label:"Canadian to EURO", data:"EU"}];
```

We chose this implementation to separate the application's display layer from its data layer (which might retrieve conversion rates from a server-side application). An alternative, less flexible implementation might embed the currency conversion rates directly in the data property of each ComboBox item.

### The `convertButton` Button component

We've already seen that the user can convert a value simply by pressing the Enter key while typing in the input text field. We'll now add an explicit Convert button to the application for users who prefer not to use the Enter key.

The `convertButton` component is an instance of the `mx.controls.Button` component class, not to be confused with the native `Button` class that represents instances of `Button` symbols in a movie. Because we imported the `mx.controls.*` package earlier, the compiler knows to treat unqualified references to `Button` as references to the `mx.controls.Button` class.

To create and position the `convertButton` component, we use now-familiar techniques:

```
// Create the Convert button.
var convertButton:Button = converter_mc.createClassObject(Button,
    "convertButton",
    4);
convertButton.move(currencyPicker.x + currencyPicker.width + 5,
    currencyPicker.y);
```

To specify the text on `convertButton`, we assign a string to its `label` property:

```
convertButton.label = "Convert!";
```

Finally, we must define what happens when `convertButton` is clicked. To do so, we assign an event handler function to `convertButton`'s `clickHandler` property, as follows:

```
convertButton.clickHandler = function (e:Object):Void {
    // Invoke convert() on the current CurrencyConverter instance
    // when convertButton is clicked.
    thisConverter.convert();
};
```

The preceding code demonstrates one way to handle component events. It defines an anonymous function that is executed automatically when `convertButton` is clicked. The property name, `clickHandler`, specifies the event to handle, in this case the `click` event. To define an event handler for a different event, we'd define an `eventHandler` property where `event` is the name of the event. For example, to handle the List component's `change` event or `scroll` events, we'd assign a function to the `changeHandler` or `scrollHandler` property.

Because the anonymous function is nested inside the `buildConverter()` method, it has access to `buildConverter()`'s local variables via the scope chain. Hence, `convertButton` can reference the current `CurrencyConverter` instance via the local variable `thisConverter`. We use that reference to invoke the `convert()` method.

We've now seen two different ways to handle events from components. For guidance on which to use in various situations, see "Handling Component Events," later in this chapter.

### The result `TextArea` component

Our interface is almost complete. We have a `TextInput` component that accepts a dollar amount from the user, a `ComboBox` component that lets the user pick a currency, and a button that triggers the conversion. All we need now is a `TextArea` component in which to display currency conversion results. `TextArea` components are used to display one or more lines of text on screen within a bordered, adjustable rectangle.

Here's the code that creates, sizes, and positions our `TextArea` component:

```
result = converter_mc.createClassObject(TextArea, "result", 5);
result.setSize(200, 25);
result.move(result.x, currencyPicker.y + currencyPicker.height + 10);
```

To prevent the user from changing the contents of the result `TextArea` component, we set its `editable` property to `false` (by default, `TextArea` component instances are `editable`):

```
result.editable = false;
```

And that concludes the `buildConverter()` method. Let's move on to the `convert()` method, which performs currency conversion.

## Converting Currency Based on User Input

In the preceding section, we created the user interface for our currency converter application. We specified that when the `Convert` button is clicked or when the user presses the `Enter` key while entering a number, the `convert()` method should be invoked. The `convert()` method converts a user-supplied value to a specific currency and displays the result on screen. It also displays error messages. The code in `convert()` provides a good example of how components are accessed and manipulated in a typical `ActionScript 2.0` OOP application. Here's a reminder of the full code listing for `convert()`:

```
public function convert ():Void {
    var convertedAmount:Number;
    var origAmount:Number = parseFloat(input.text);
    if (!isNaN(origAmount)) {
        if (currencyPicker.selectedItem.data != null) {
            switch (currencyPicker.selectedItem.data) {
                case "US":
                    convertedAmount = origAmount / CurrencyConverter.rateUS;
                    break;
                case "UK":
                    convertedAmount = origAmount / CurrencyConverter.rateUK;
                    break;
                case "EU":
```

```

        convertedAmount = origAmount / CurrencyConverter.rateEU;
        break;
    }
    result.text = "Result: " + convertedAmount;
} else {
    result.text = "Please select a currency.";
}
} else {
    result.text = "Original amount is not valid.";
}
}
}

```

Our first task in `convert()` is to create two local variables: `convertedAmount`, which stores the postconversion value, and `origAmount`, which stores the user-supplied value. The value of `origAmount` is retrieved from the input component's text property, which stores the user input as a *String*. We convert that *String* to a *Number* using the built-in `parseFloat()` function:

```

var convertedAmount:Number;
var origAmount:Number = parseFloat(input.text);

```

But the `input.text` property might be empty, or it might not be valid (that is, it might contain a nonmathematical value such as "...4..34"). Hence, our next task is to check whether the conversion from a *String* to a *Number* succeeded. If conversion did not succeed, the value of `origAmount` will be NaN (not-a-number). Hence, we can say that the value is valid when it is *not* NaN:

```

if (!isNaN(origAmount)) {

```

If the value is valid, we check which currency is selected in the drop-down list using `currencyPicker.selectedItem.data`. The `selectedItem` property stores a reference to the currently selected item in the `ComboBox`, which is one of the objects in the `dataProvider` array we created earlier. To determine which object is selected, we consult that object's `data` property, which will be one of: `null`, "US," "UK," or "EU." If the `data` property is `null`, no item is selected and we should not attempt to convert the currency:

```

    if (currencyPicker.selectedItem.data != null) {

```

If, on the other hand, `data` is not `null`, we use a *switch* statement to determine whether `data` is "US," "UK," or "EU." Within the *case* block for each of those possibilities, we perform the conversion to the selected currency:

```

    switch (currencyPicker.selectedItem.data) {
        case "US":
            convertedAmount = origAmount / CurrencyConverter.rateUS;
            break;
        case "UK":
            convertedAmount = origAmount / CurrencyConverter.rateUK;
            break;
        case "EU":
            convertedAmount = origAmount / CurrencyConverter.rateEU;
            break;
    }
}

```

Once the converted amount has been calculated, we display it in the result TextArea component, as follows:

```
result.text = "Result: " + convertedAmount;
```

When no currency has been selected, we display a warning in the result component:

```
result.text = "Please select a currency.";
```

Similarly, when the amount entered in the input component is not valid, we display a warning in the result component:

```
result.text = "Original amount is not valid.";
```

## Exporting the Final Application

Our currency converter application is ready for testing and deployment. To specify the name of the movie to create (*CurrencyConverter.swf*), follow these steps:

1. With *CurrencyConverter.fla* open, choose File → Publish Settings → Formats.
2. Under the File heading, for Flash (.swf), enter **../deploy/CurrencyConverter.swf**.
3. Click OK.
4. To test our application in the Flash authoring tool's Test Movie mode, select Control → Test Movie.

For our application, we'll export to Flash Player 7 format (the default in Flash MX 2004), but you could also export to Flash Player 6 format if you expect your visitors to be using that version of the Flash Player. The v2 components officially require Flash Player 6.0.79.0 or higher, but in my tests the currency converter application worked happily in Flash Player 6.0.40.0 as well.

## Handling Component Events

In this chapter, we handled component events in two different ways:

- With a generic listener object (in the case of the TextInput component):

```
var enterHandler:Object = new Object();
enterHandler.enter = function (e:Object):Void {
    thisConverter.convert();
}
input.addEventListener("enter", enterHandler);
```

- With an event handler function (in the case of the Button component):

```
convertButton.clickHandler = function (e:Object):Void {
    thisConverter.convert();
};
```

Handling component events with generic listener objects in ActionScript 2.0 is somewhat analogous to handling Java Swing component events with anonymous inner classes. In Swing, an anonymous instance of an anonymous inner class is created

simply to define a method that responds to a component event. In ActionScript 2.0, an instance of the generic *Object* class is created for the same reason (to define a component-event-handling method). But in ActionScript 2.0, the anonymous class is not required because new methods can legally be added dynamically to instances of the *Object* class at runtime.

In general, the listener object approach is favored over the event handler function approach, primarily because multiple listener objects can receive events from the same component, whereas only one event handler function can be defined for a component at a time. This makes listener objects more flexible and scalable than event handler functions. Hence, Macromedia formally discourages use of event handler functions. However, you'll definitely see both approaches thriving in the wild. The older v1 components that shipped with Flash MX did not support listener objects, so all older v1 code uses event handler functions. The v2 components support event handler functions for backward compatibility. Moving forward, you should use listener objects rather than event handler functions. That said, even if you're not working with components, you'll still encounter event handler functions when working with the Flash Player's built-in library of classes. Many of the built-in classes, including *MovieClip*, *Sound*, *XML*, and *XMLSocket* use event handler functions as their only means of broadcasting events.

As an alternative to defining an event handler function on a component instance, you can also use a so-called *listener function*, which logically lies somewhere between an event handler function and a listener object. A listener function is a standalone function (i.e., a function not defined on any object) registered to handle a component event. For example, our earlier event handler function for the Convert Button component looked like this:

```
convertButton.clickHandler = function (e:Object):Void {
    thisConverter.convert();
}
```

The analogous listener function would be:

```
function convertClickHandler (e:Object):Void {
    thisConverter.convert();
};
convertButton.addEventListener("click", convertClickHandler);
```

Listener functions are preferred over event handler functions because multiple listener functions can be registered to handle events for the same component. However, when using listener functions, you should be careful to delete the function once it is no longer in use. Or, to avoid cleanup work, you might simply pass a function literal to the *addEventListener()* method of the component in question, as follows:

```
convertButton.addEventListener("click", function (e:Object):Void {
    thisConverter.convert();
});
```

However, using this function literal approach prevents you from ever removing the listener function from the component's listener list. Hence, when registering for an event that you may later want to stop receiving, you should not use the preceding function literal approach.

Whether you're using a listener object, an event handler function, or a listener function, the fundamental goal is the same: to map an event from a component to a method call on an object. For example, in our Convert button example, we want to map the button's click event to our *CurrencyConverter* object's *convert()* method. Yet another way to make that mapping would be to define a *click()* method on the *CurrencyConverter* class and register the *CurrencyConverter* instance to handle button click events. Here's the *click()* method definition:

```
public function click (e:Object):Void {
    convert();
}
```

And here's the code that would register the *CurrencyConverter* instance to receive click events from the Convert button:

```
convertButton.addEventListener("click", this);
```

In the preceding approach, because the *click()* method is called on the *CurrencyConverter* instance, the *click()* method can invoke *convert()* directly, without the need for the *thisConverter* local variable that was required earlier. However, problems arise when the *CurrencyConverter* instance needs to respond to more than one Button component's click event. To differentiate between our Convert button and, say, a Reset button, we'd have to add cumbersome *if* or *switch* statements to our *click()* method, as shown in the following code. For this example, assume that the instance properties *convertButton* and *resetButton* have been added to the *CurrencyConverter* class.

```
public function click (e:Object):Void {
    if (e.target == convertButton) {
        convert();
    } else if (e.target == resetButton) {
        reset();
    }
}
```

Rather than forcing our *CurrencyConverter* class to handle multiple like-named events from various components, we're better off reverting to our earlier generic listener object system, in which each generic object could happily forward events to the appropriate methods on *CurrencyConverter*. For example:

```
// Convert button handler
var convertClickHandler:Object = new Object();
convertClickHandler.click = function (e:Object):Void {
    thisConverter.convert();
}
convertButton.addEventListener("click", convertClickHandler);
```

```

// Reset button handler
var resetClickHandler:Object = new Object();
resetClickHandler.click = function (e:Object):Void {
    thisConverter.reset();
}
resetButton.addEventListener("click", resetClickHandler);

```

To reduce the labor required to create generic listener objects that map component events to object method calls, Mike Chambers from Macromedia created a utility class, *EventProxy*. Using Mike's *EventProxy* class, the preceding code could be reduced to:

```

convertButton.addEventListener("click", new EventProxy(this, "convert"));
resetButton.addEventListener("click", new EventProxy(this, "reset"));

```

The *EventProxy* class, shown in Example 12-2, does a good, clean job of mapping a component event to an object method call. However, the convenience of *EventProxy* comes at a price: reduced type checking. For example, in the following line, even if the current object (*this*) does not define the method *convert()*, the compiler does not generate a type error:

```

convertButton.addEventListener("click", new EventProxy(this, "convert"));

```

Hence, wherever you use the *EventProxy* class, remember to carefully check your code for potential datatype errors. For more information on *EventProxy*, see: <http://www.markme.com/mesh/archives/004286.cfm>.

#### Example 12-2. The *EventProxy* class

```

class EventProxy {
    private var receiverObj:Object;
    private var funcName:String;

    /**
     * receiverObj The object on which funcName will be called.
     * funcName The function name to be called in response to the event.
     */
    function EventProxy(receiverObj:Object, funcName:String) {
        this.receiverObj = receiverObj;
        this.funcName = funcName;
    }

    /**
     * Invoked before the registered event is broadcast by the component.
     * Proxies the event call out to the receiverObj object's method.
     */
    private function handleEvent(eventObj:Object):Void {
        // If no function name has been defined...
        if (funcName == undefined) {
            // ...pass the call to the event name method
            receiverObj[eventObj.type](eventObj);
        } else {

```

Example 12-2. The *EventProxy* class (continued)

```
        // ...otherwise, pass the call to the specified method name
        receiverObj[funcName](eventObj);
    }
}
}
```

To avoid the type checking problem presented by the *EventProxy* class, you can use the rewritten version of Mike Chambers' original class, shown in Example 12-3. The rewritten version uses a function reference instead of a string to access the method to which an event is mapped. Hence, to use the rewritten *EventProxy* class, we pass a method instead of a string as the second constructor argument, like this:

```
    // No quotation marks around convert! It's a reference, not a string!
    convertButton.addEventListener("click", new EventProxy(this, convert));
```

Because the *convert()* method is accessed by reference, the compiler generates a helpful error if the method doesn't exist.

Example 12-3. The Rewritten *EventProxy* class

```
class EventProxy {
    private var receiverObj:Object;
    private var funcRef:Function;

    /**
     * receiverObj The object on which funcRef will be called.
     * funcName   A reference to the function to call in response
     *            to the event.
     */
    function EventProxy(receiverObj:Object, funcRef:Function) {
        this.receiverObj = receiverObj;
        this.funcRef = funcRef;
    }

    /**
     * Invoked before the registered event is broadcast by the component.
     * Proxies the event call out to the receiverObj object's method.
     */
    private function handleEvent(eventObj:Object):Void {
        // If no function name has been defined...
        if (funcRef == undefined) {
            // ...pass the call to the event name method
            receiverObj[eventObj.type](eventObj);
        } else {
            // ...otherwise, pass the call to the specified method using
            // Function.call().
            funcRef.call(receiverObj, eventObj);
        }
    }
}
```

As evidenced by the sheer number of event-handling techniques just discussed, the v2 component-event-handling architecture is very flexible. But it also suffers from a general weakness: it allows type errors to go undetected in two specific ways.

First, any component's events can be handled by any object of any class. The compiler does not (indeed cannot) check whether an event-consuming object defines the method(s) required to handle the event(s) for which it has registered. In the following code, if the `convertClickHandler` object does not define the required `click()` method, no error occurs at compile time:

```
var convertClickHandler:Object = new Object();
// Oops! Forgot the second "c" in "click," but no compiler error occurs!
convertClickHandler.click = function (e:Object):Void {
    thisConverter.convert();
}
convertButton.addEventListener("click", convertClickHandler);
```

In other words, in the v2 component architecture there's no well-known manifest of the events a component broadcasts and no contract between the event source and the event consumer to guarantee that the consumer actually defines the events broadcast by the source.

Second, event objects themselves are not represented by individual classes. All event objects are instances of the generic *Object* class. Hence, if you misuse an event object within an event-handling method, the compiler, again, does not generate type errors. For example, in the following code (which, so far, contains no type errors), we disable a clicked button by setting the button's `enabled` property to `false` via an event object. We access the button through the event object's `target` property, which always stores a reference to the event source:

```
convertClickHandler.click = function (e:Object):Void {
    thisConverter.convert();
    e.target.enabled = false;
}
```

But if the programmer specifies the wrong property name for `target` (perhaps due to a typographical error or a mistaken assumption), the compiler does not generate a type error:

```
convertClickHandler.click = function (e:Object):Void {
    thisConverter.convert();
    e.source.enabled = false; // Wrong property name! But no compiler error!
    e.trget.enabled = false; // Oops! A typo, but no compiler error!
}
```

In addition to suppressing potential compiler errors, the lack of typed event objects in the v2 component architecture effectively hides the information those objects contain. If the architecture used formal event classes, such as, say, *Event* or *ButtonEvent*, the programmer could quickly determine what information is available for an event simply by examining the v2 component class library. As things stand, such information can be found only in the documentation (which may be incomplete) or in an event-broadcasting component's raw source code (which is laborious to read).

One way to help make an application's handling of v2 component events more obvious is to define specific classes for event-consumer objects rather than using generic objects. For example, to handle events for the Convert button, an instance of the Button component, in our *CurrencyConverter* application, we might create a custom *ConvertButtonHandler* class as follows:

```
import org.moock.tools.CurrencyConverter;

class org.moock.tools.ConvertButtonHandler {
    private var converter:CurrencyConverter;

    public function ConvertButtonHandler (converter:CurrencyConverter) {
        this.converter = converter;
    }

    public function click (e:Object):Void {
        converter.convert();
    }
}
```

Then, to handle the Button component events for the Convert button, we'd use:

```
convertButton.addEventListener("click", new ConvertButtonHandler(this));
```

By encapsulating the button-event-handling code in a separate class, we make the overall structure of the application more outwardly apparent. We also isolate the button-handling code, making it easier to change and maintain. However, in simple applications, using a separate class can require more work than it's worth. And it doesn't alleviate the other event type checking problems discussed earlier (namely, the compiler's inability to type check event-consumer objects and event objects).

In Java, every aspect of the Swing component event architecture includes type checking. Event consumers in Java must implement the appropriate event listener interface, and event objects belong to custom event classes. In simple Flash applications, Java's additional component event architecture would be cumbersome and hinder rapid, lightweight development. However, for more complex situations, Java's strictness would be welcome. Therefore, we'll see how to implement Java-style events in ActionScript 2.0 classes in Chapter 19. And we'll learn more about generating and handling custom user interface events in Chapter 18.

For reference, Table 12-1 summarizes the various component-event-handling techniques discussed in this chapter.

Table 12-1. Component-event-handling techniques

Technique	Example	Notes
Generic listener object	<pre>var convertClickHandler:Object = new Object(); convertClickHandler.click = function (e:Object):Void {     thisConverter.convert(); } convertButton.addEventListener("click", convertClickHandler);</pre>	Generally, the preferred means of handling component events
Typed listener object	<pre>convertButton.addEventListener("click", new ConvertButtonHandler(this));</pre>	Same as generic listener object, but exposes event-handling code more explicitly
EventProxy class	<pre>convertButton.addEventListener("click", new EventProxy(this, "convert")); or convertButton.addEventListener("click", new EventProxy(this, convert));</pre>	Functionally the same as generic listener object, but more convenient and easier to read
Listener function	<pre>convertButton.addEventListener("click", function (e:Object):Void {     thisConverter.convert(); });</pre>	The lesser evil of the two function-only event-handling mechanisms
Event handler function	<pre>convertButton.clickHandler = function (e:Object):Void {     thisConverter.convert(); }</pre>	The least-desirable means of handling component events; discouraged by Macromedia

## Components Complete

We've come to the end of our look at a component-based ActionScript 2.0 application. If you want to see the currency converter in action, you can download all the files discussed in this chapter from <http://moock.org/eas2/examples>. For a lot more information on both using and authoring components, see Flash's online Help (Help → Using Components) and the components section of Macromedia's Flash Developer Center at <http://www.macromedia.com/devnet/mx/flash/components.html>.

In the next chapter, we'll continue our exploration of controlling and creating visual assets by studying *MovieClip* subclasses.