O'REILLY® Short Cuts

ActionScript 3.0 Programming:

Overview, Getting Started, and Examples of New Concepts

By Bill Sanders

ISBN: 978-0-596-52923-9
Release Date: January 19, 2007

The release of ActionScript 3.0 represents the most significant change in ActionScript since Flash was introduced. With each new version of Flash, developers and designers saw incremental changes in ActionScript. ActionScript 1.0 slowly grew with each new version of Flash. Then ActionScript 2.0 introduced user classes stored in separate files and the first inkling of a true object-oriented programming language.

With ActionScript 3.0 is not only a programming language for developing object-oriented applications, but the first major implementation of ECMAScript E4, the current Internet language standard. This means that you're not just learning ActionScript 3.0, but you're also learning all the ECMAScript–based languages to come. Instead of being the language just for Flash 9 and Flex 2, ActionScript 3.0 is the language of the Internet.

ADOBE DEVELOPER LIBRARY

Adobe

Contents

Summary

This Short Cut is designed to get you up and running with ActionScript 3.0 working with Flash 9, currently available in pre-release version to registered Flash 8 users at: http://labs.adobe.com/technologies/flash9as3preview/.

All cases of Example code can be downloaded at: **http://examples.oreilly.com/actionscript3qr.**

The technical editor/reviewer was **Darren Richardson**. Some of his work and insights into the Flash world can be found on his blog at www.playfool.com & www.experiment.org.uk.

You can email him at *darren@actionscripts.co.uk*.

Invitation to ActionScript 3.0

Depending on your orientation to Flash as either a designer or developer, or some combination thereof, most who work with ActionScript create programs in one of three styles.

First, designers tend to select a stage object (such as a button or movie clip) and enter short scripts using the "on" function. For example, a designer might have a button that moves a movie clip when pressed. Or she might have a movie clip change shapes when a mouse moves over it. However, this method is no longer available in ActionScript 3.0; throughout this Short Cut you will see how to effectively create the same effects using other methods in ActionScript 3.0. I urge designers to take a look at the *Display Programming* section (beginning on page 9). The investment in learning something about it will pay dividends far into the future. ActionScript is moving forward at a fast pace, and professional Flash developers need to move ahead as well.

Second, a number of Flash developers have been working with ActionScript as an Internet language almost exclusively. They have become adept at creating scripts using ActionScript 1.0 and 2.0 using the Actions panel in Flash. Scripts are placed in frames and Rich Internet Applications (RIAs) are created for a wide variety of applications. Most of the work is done in a style generally described as "sequential" programming. This group does not have a formal background in programming, but over the years their skills have increased along with the sophistication of ActionScript.

Third, some developers have migrated to ActionScript from other object-oriented programming (OOP) languages such as Java or C#. These developers will find the new ActionScript very similar to what they are used to seeing in these other OOP languages. Because ActionScript 3.0 closely follows the ECMAScript standards, key differences can be found in the way variables are typed, and other features that will differ in syntax but not in purpose from another OOP language. Many of the packages in ActionScript 3.0 are unique to the Flash environment, but the structure is very similar to what programmers have come to expect from other object-oriented languages.

In addition to the three major categories of ActionScript users, I am sure others with JavaScript, VB.NET, PHP and similar backgrounds have found their way to ActionScript. Because ECMAScript was established as a general standard for Internet languages, these other languages may begin a similar migration to meet these standards. As a result, learning ActionScript 3.0 may simply be the step that prepares you for learning the structures to which other languages may evolve. (For a more detailed understanding of ActionScript 3.0, see *Essential ActionScript 3.0*

by Colin Moock and *ActionScript 3.0 Cookbook* by Joey Lott, Darron Schall, and Keith Peters, both from O'Reilly)

Packages and Classes

Probably the most important feature to grasp about ActionScript 3.0 is how to work with *packages* and *classes*. In fact, your entire workflow initially rests on your ability to recognize that you only deal with certain features of ActionScript at any one time. For example, if you want to work with a text field, you have to first import the `TextField` class from the `flash.text` package. Of course, you'll need to know that the `TextField` class is in the `flash.text` package, as well as where other classes are stored. To get started, take a look at the script in Example 1:

Example 1. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;
    import flash.text.TextField;
    public class ShowText extends Sprite
    {
        public function ShowText():void
        {
            var actionText:TextField=new TextField();
            var msg:String="Hello ActionScript 3.ohhhhh";
            actionText.text=msg;
            this.addChild(actionText);
            actionText.width=((msg.length)*12)+4;
            actionText.height=16;
            actionText.x=200;
            actionText.y=150;
        }
    }
}
```

Save the script as `ShowText.as`. To test this script, use the following steps:

1. Open a new Flash document, and in the Document class window in the Properties panel, type in the class name, `ShowText`, as shown in Figure 1.
2. Save the Flash document as `TextSimple.fla` in the same folder as the `ShowText.as` file.
3. Select Control → Test Movie (short cuts: PC = [Cntrl + Enter]; Mac = [Apple + Return]), and in the middle of the page you'll see the message "`ActionScript 3.ohhhhh`", as shown in Figure 1.

Before going on, understanding why you had to import the `TextField` and `Sprite` classes is important. After all, writing a script in the Actions panel requires no such import in previous versions of ActionScript. ActionScript 3.0 has roughly 185 classes organized into 17 packages. In this particular application, only

two of those 185 classes needed to be used, so instead of dragging along 183 unneeded classes, only the two required classes were imported.

To help understand the importance of importing only what you need, consider a trip to Tahiti. When you start packing, you'll take tropical weather clothing, your swimsuit, and maybe a snorkel and diving mask for swimming with the fishes. Just because you also have an Arctic-weather parka, snow shoes, and a portable heater doesn't mean you need to take them along, too. You also might take your laptop computer, but you're not going to try and pack your desktop computer. You just pack what you need and will most efficiently allow you to travel lightly. That's exactly the same thing you're doing with ActionScript 3.0 imports—you're just taking the classes you need. When your program runs, it does so far more efficiently and faster and it won't take up as much memory because you just "packed" what was required—and nothing more.

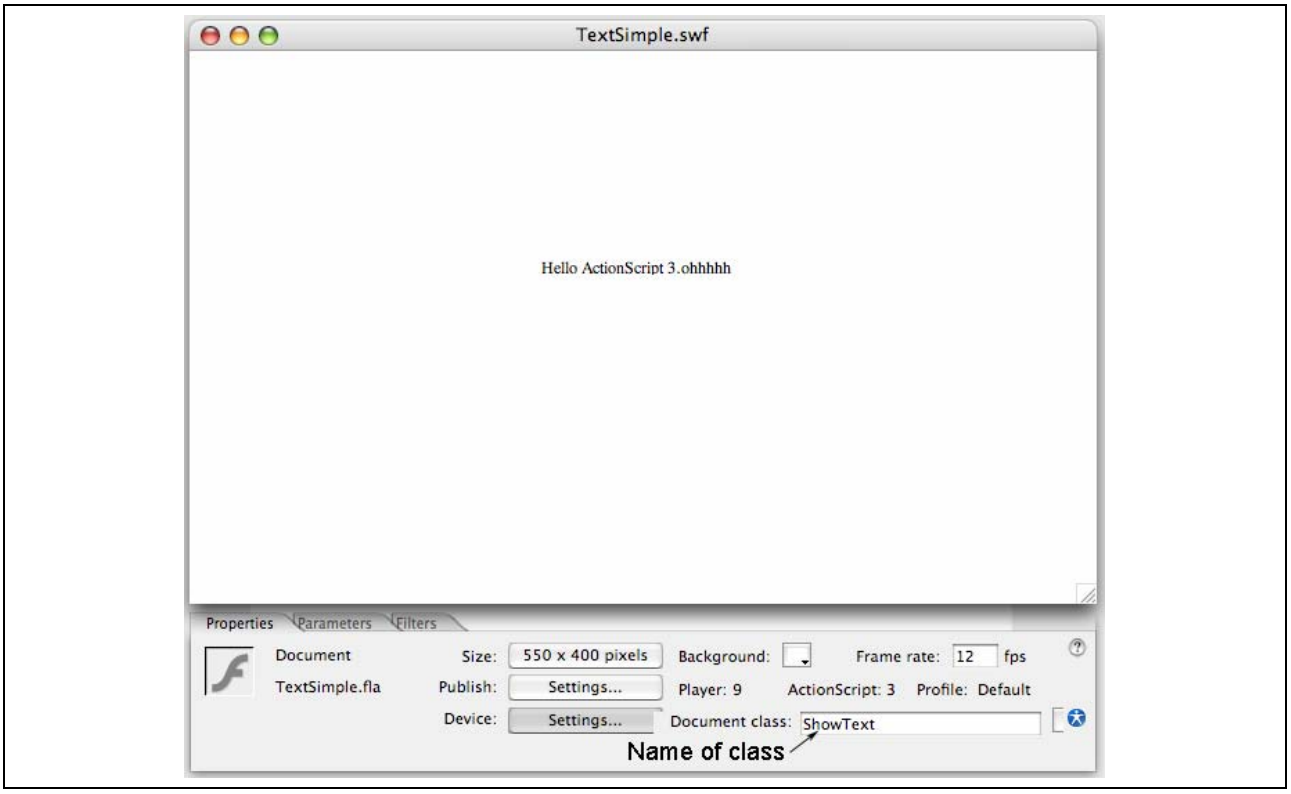

Figure 1. The script is associated with the Flash document through the Document class

Finding the Right Package and Class

In the most general sense you can think of a package as a *class directory*. If you look in the `flash.text` package, for example, you'll find classes like `TextField`, `TextFormat`, and other related classes for doing stuff with text. Table 1 lists the packages where classes are organized along with a brief description of each.

Table 1. ActionScript 3.0 Packages

Package Name	Description
adobe.utils	Used by Flash authoring tool developers
flash.accessibility	Two accessibility classes
flash.display	A key package for visual displays, containing 31 classes
flash.errors	Part of Flash Player API and specific to ActionScript
flash.events	A wide variety of event classes for all events you'll use
flash.external	A single class, `ExternalInterface`
flash.filters	Allows dynamic addition of bitmap filters to display objects
flash.geom	Classes for supporting `BitmapData` class and bitmap caching
flash.media	Video, Sound, Camera, Microphone, and related classes
flash.net	Connection, stream-related and URL request classes
flash.print	Three classes related to printing
flash.profiler	Single `showRedrawRegions` function
flash.system	Classes to access system-level functionality
flash.text	Text, Font, `StyleSheet` and related classes
flash.ui	User interface classes including `ContextMenu`, Keyboard and Mouse
flash.utils	Includes functions, interfaces, and classes for wide range of tasks such as setting and clearing intervals, escaping and unescaping UTF-8 code, getting class names and setting timers
flash.xml	XML-related classes (`XMLSocket` class is in `flash.net` package)

The more you use ActionScript 3.0, the more you'll become familiar with where certain classes are located in the packages. When you fail to include the `import` of a class or function, you'll get an error message.. To see the kind of error an omitted `import` generates, comment out the line that imports the `TextField` class in Example 1; for example:

```
//import flash.text.TextField;
```

Figure 2 shows the compile errors generated when you test the application without importing the `TextField` class first.

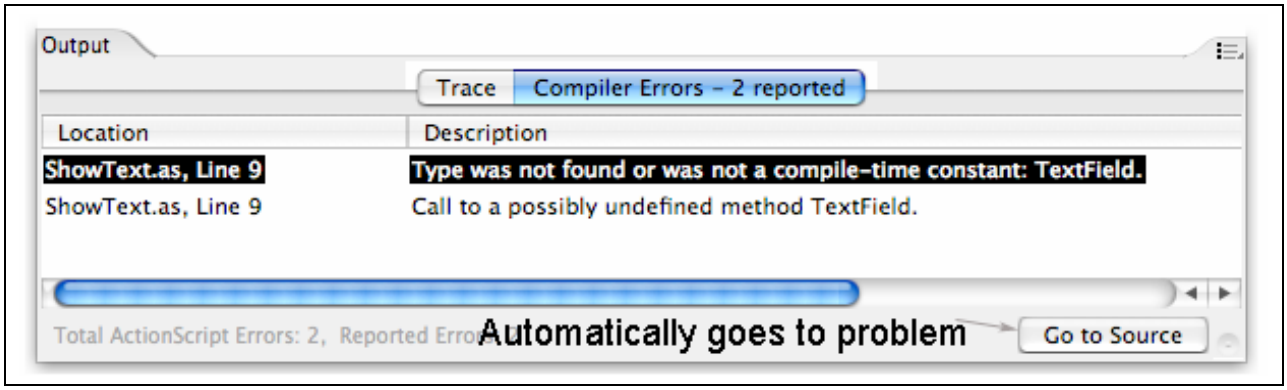

Figure 2. Compile errors due to missing class import from package

Another very nice feature in the Flash 9 Output window is the "Go to Source" button. When clicked, it takes you to the source of the selected error. If the file containing the error is not loaded, it automatically loads it for you.

In the workflow, you will find having the *ActionScript 3.0 Language Reference* open while you work extremely useful. (The *Reference* comes with the preview version of ActionScript 3.0 downloaded from http://labs.adobe.com/technologies/flash9as3preview/) The top left column lists the packages, and the bottom left columns lists all classes. If you click on a class you want to import, the right column shows all the information for that class, including properties, methods, constants and other information about the class. At the top it shows the package path. For example, if you click the `FileReference` class, you will see that the package path is `flash.net`.

Using the wildcard (*) with packages

You'll quickly learn that you can use the wildcard character, an asterisk (*), instead of placing the name of the class in the `import` path. For example, instead of writing the following:

```
import flash.display.Sprite;
```

you can use this to import the whole package instead:

```
import flash.display.*;
```

Doing so, especially with a package like `display`, defeats the purpose of only importing the classes you need because it loads everything in the package.

However, while learning where everything is, you can save some time by using the `*` wildcard. When developing your project, go ahead and use the wildcard, and when you're finished, go back and remove the wildcards and import only those classes your application actually needs.

Another purpose behind specifying the exact classes and functions your script uses is that the list of class `import`s tells you what's going on in your script when you come back to it at a later date. If you want to re-use the class you built, the `import` list gives you a quick overview of your script when you want to use it again. For example, if your script imports a `Sprite` (a Sprite is a basic display list building block—sort of a MovieClip without the timeline), but your new class needs a `MovieClip` class, you can quickly see that you'll just have to change the `Sprite` import to a `MovieClip`. Adding the * wildcard to the import statement lets you import all of the classes in a package, sort of as a catch-all to help speed up your development time. But keep in mind that that wildcard comes with a cost; in this case, the wildcard character just tells you that you're importing all the classes in the display package. Don't forget to go back and change those wildcards to import only the classes your application uses, or else the wildcard will load everything, which could potentially make your application take longer to load and slower to run.

Jumping into Display Programming

For those of you who have worked with ActionScript in the past, chances are you've used it to dynamically manipulate elements on the stage. The term *display programming* refers to ActionScript 3.0's ability to work with different elements that appear on the stage. To effectively do so, you need to know something about ActionScript 3.0's display structure.

The first thing to know is that the `Stage` is the base container of the display structure. As you add further containers to the stage, they are added to a *display*

list. Each container can hold different *display objects* that appear on the screen. To get started, you'll see how to create an object container (`DisplayObjectContainer` object) that's placed on the stage, and then you'll place a `Shape` object into a container.

If you've had previous experience with Flash, you know that you can place objects such as text fields, drawings, and even other movie clips into a `MovieClip` object. However, ActionScript 3.0 has a new `Sprite` class that can hold other objects in a similar way as the `MovieClip` class. Unlike the `MovieClip` class, though, the `Sprite` class has no timeline. (In fact, the `MovieClip` is actually a subclass of the `Sprite` class.) Without a timeline, the `Sprite` class requires less memory and can be used as an ideal container for different objects while retaining many of the advantages of a `MovieClip` object.

Take a look at Figure 3. It shows the script that created the square that appears on the stage below the script. To reproduce this little application and help understand how *display* programming works, implement the following steps:

1. Open a new Flash document and click on the first keyframe.
2. Open the Actions panel and enter the script shown in Figure 3.
3. Save the file and test it.

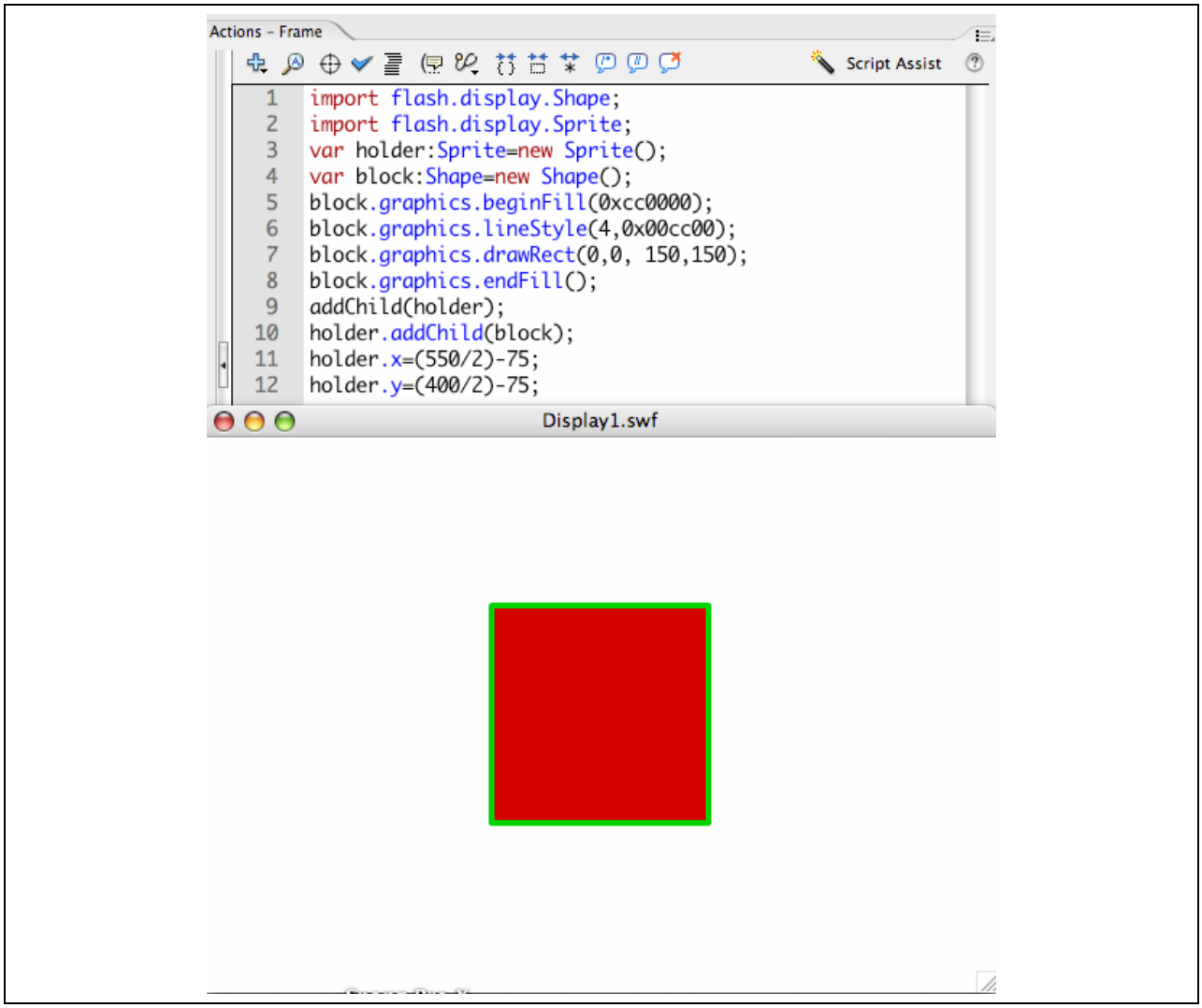

Figure 3. Script to place a shape in a Sprite object on the stage

Now that you have a listing and a visual reference to display programming, you can better understand its structure.

In Line 9, using the `addChild()` method, the `Sprite` instance named `holder` is added to the `Stage` object's base container. Next, the `Shape` instance (named `block`) is added to the `holder` container, again using the `addChild()` method. However, this time, the `holder` instance is the container and not the stage. To specify a container, the `addChild()` method is simply added to the container instance: `holder.addChild(block)`. Using the `x` and `y` coordinates for the center of the stage, the square is repositioned from the `x=0`, `y=0` in the upper-left corner. If you're experienced with ActionScript, you will notice that instead of using `_x` and `_y` for the horizontal and vertical positions, the leading underscores have been removed from the property identifiers. (The stage's dimensions were the default width = 550, height = 400.)

In a nutshell, that's the display architecture. As each new container is added to the stage or another container, it joins the display list. (You will note that a reference to *stage*, with the lowercase first letter, refers to the place where you do your drawings and place objects in different layers. A reference to `Stage`, with the uppercase first letter, refers to the `Stage` class.) Figure 4 shows an abstract model of the application in Figure 3 in terms of the display architecture.

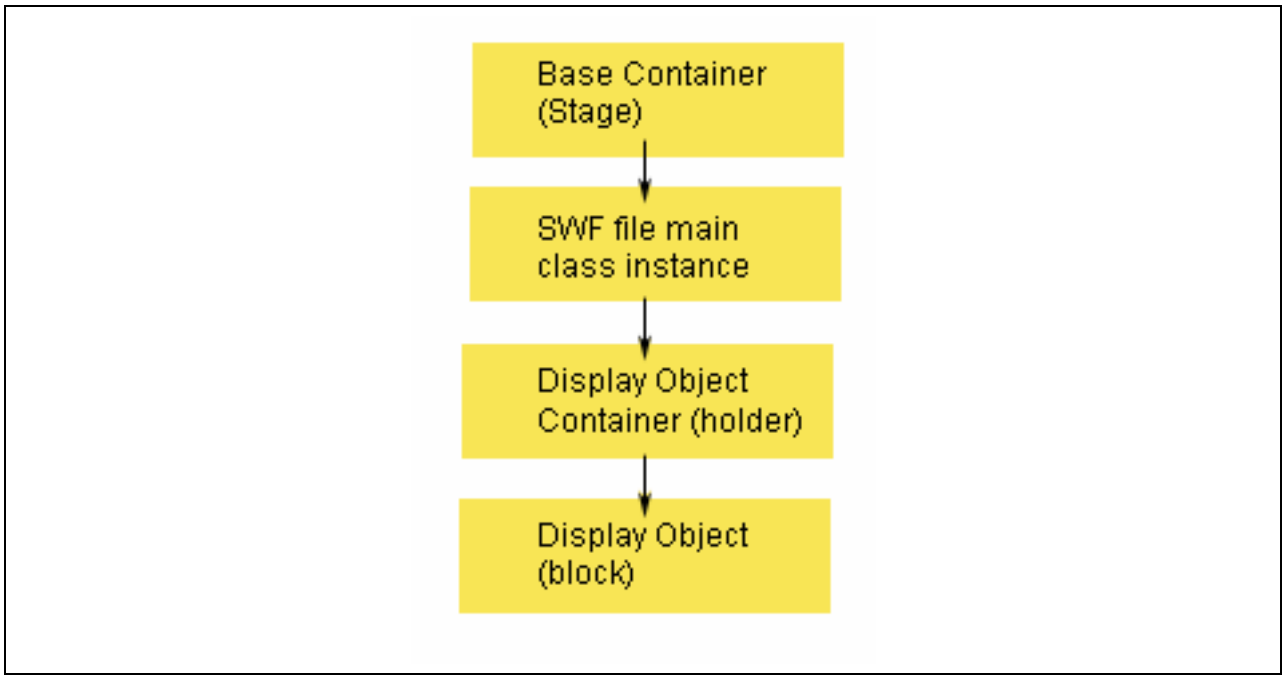

Figure 4. Display architecture is made up of display object containers and the display objects they contain

Now, just to see what happens, comment out the two `import` lines:

```
//import flash.display.Shape;
//import flash.display.Sprite;
```

Test the application again. If everything works fine, you won't see a difference from the first time you ran the application; here's why: When you place script in a keyframe on the timeline using the Actions panel, all of the packages are automatically loaded. Your scripts are nowhere near as efficient because they are hauling the overhead of unused packages and classes.

To employ the `import`s efficiently, you have to put them in a class of your own, and call them from the Document class in the Flash document. (This technique is employed in Example 1.) So, for a good deal of your code work, you'll be using ActionScript files (`.as` files) instead of typing code into the Actions panel. Example 2 shows how your code should look when it is placed in a class. The class name is `ActionSquare`, and the file should be saved as `ActionSquare.as`.

Example 2. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
```

```
{
import flash.display.Shape;
import flash.display.Sprite;
    public class ActionSquare extends Sprite
    {
        public function ActionSquare():void
        {
        var holder:Sprite=new Sprite();
        var block:Shape=new Shape();
        block.graphics.beginFill(0xcc0000);
        block.graphics.lineStyle(4,0x00cc00);
        block.graphics.drawRect(0,0, 150,150);
        block.graphics.endFill();
        addChild(holder);
        holder.addChild(block);
        holder.x=(550/2)-75;
        holder.y=(400/2)-75;
        }
    }
}
```

Remove the ActionScript from the frame, and in the Document class window of the Properties panel, type in `ActionSquare`. Now when you test the script, even though you will see exactly the same thing on the stage, you'll know that it used just the classes it needed. (Using the timeline, the SWF file is 545 bytes, while using a class generates only 451 bytes—a 104-byte difference.)

Loading and Arranging Graphics

Now that you have a general idea of the display architecture and how to get display items on the stage or in a container on the stage, you'll find working with other display objects clearer.

An important new feature of ActionScript 3.0 is the new `Loader` class. This class lets you load external graphic or SWF files. (It replaces the `MovieClipLoader` class from ActionScript 2.0.) For example, suppose you have a graphic for a logo that you use frequently. The `Loader` instance lets you load the image into the instance, and the instance name effectively becomes the object's reference for displaying it on the stage. Using a `load()` method and `URLRequest` instance to target the logo's image file, you can then place it anywhere you want in a container.

Example 3 shows the process for setting up and positioning an image on the stage. You can load GIF, JPEG, and PNG files. If you try to load other kinds of files,

you'll get an error message. This includes PSD (Photoshop) files that you can load directly onto the stage in Flash 9 and are converted to a format that can be displayed. `Loader` object becomes the container, and yet it can be treated as a display object itself. Virtually all of the properties you would apply to a `MovieClip` instance can be used with the `Loader` instance. Open an ActionScript file and save the script shown in Example 3 as `LogoLoader.as`.

Example 3. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;
    import flash.display.Loader;
    import flash.net.URLRequest;
    public class LogoLoader extends Sprite
    {
        public function LogoLoader():void
        {
            var logoLoader:Loader=new Loader();
            var logoup:URLRequest=new URLRequest("logo.gif");
            logoLoader.load(logoup);
            addChild(logoLoader);
            logoLoader.x=40;
            logoLoader.y=20;
        }
    }
}
```

Open a new Flash document, and in the `Document` class, type `LogoLoader`. You can substitute `logo.gif` for any image file you have on hand; be sure to place the image in the same folder where you have saved the `LogoLoader.as` file.

When you test the script, you'll see the image appear on the stage. The `Loader` object can only have a single child display object: the file it loads. While the `Loader` serves as a display object container, you cannot add more children to it as you can a `Sprite`.

However, given the architecture of display programming in ActionScript 3.0, you can add more than one image to the `Stage` or another container, not a `Loader`. Further, you can order the depth of each image. As you add each child to a container, the child is given a 0-based value incremented with each additional child. The lower the value, the lower the image is in the container. So the first image you place in a container is at the very back and the last is at the very top.

To see how this works, Example 4 loads two images, but the second image needs to load beneath the first image. To achieve that, the example uses the `addChildAt(imageRef, depth)` method. Open a new ActionScript file and enter the code shown in Example 4. Save the file as `LogoLoaderAt.as` and add two graphic files to the same folder that represent the `logo.gif` and `star.png` files.

Example 4. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;
    import flash.display.Loader;
    import flash.net.URLRequest;
    public class LogoLoaderAt extends Sprite
    {
        public function LogoLoaderAt():void
        {
            var starLoader:Loader=new Loader();
            var logoLoader:Loader=new Loader();
            var starup:URLRequest=new URLRequest("star.png");
            var logoup:URLRequest=new URLRequest("logo.gif");
            starLoader.load(starup);
            addChild(starLoader);
            logoLoader.load(logoup);
            addChildAt(logoLoader,0);
            logoLoader.x=40;
            logoLoader.y=20;
            starLoader.x=247;
            starLoader.y=75;
        }
    }
}
```

In a Flash document file, type in `LogoLoaderAt` in the `Document` class window and test the application.

Figure 5 shows the results of the code shown in Example 3 (at left) and Example 4 (at right). It also shows what happens if `addChild()` instead of `addChildAt()` were used in Example 4. The figure on the left shows the logo with the star image hidden behind the palm tree because by default the first child added has a lower depth. However, the figure on the right shows the star on top of the palm tree because the script added the second figure, the logo, at a lower depth (0). As such, the star image appears on top of the palm tree.

Figure 5. Loaded images without (left) and with (right) depth control

For those of you who are primarily graphic designers, you should look at ActionScript 3.0 as an opportunity to better control images created in other applications. Also, keep in mind that you can still decorate the stage with vector graphics created in Flash and position other graphics around them using ActionScript 3.0.

Loading and Displaying Text

One of the most important features of ActionScript is its ability to dynamically load text. Flash applications that load all text content at once not only lack flexibility, but they also tend to be bloated and inefficient. Text is relatively "light"; it doesn't require many bytes. However, text within a SWF file can become quite heavy and slow, so learning how to work with dynamically loaded text is something that should be at the top of your list.

ActionScript 3.0 doesn't use the `Loader` class for loading external text files; it uses the `URLLoader` class instead. Moreover, the `URLLoader` is in the `flash.net` package and not the `flash.display` package. One reason for this difference is the content that loads with the loader class is placed into a `DisplayObjectContainer` instance and text is placed into a `TextField` instance. However, the `TextField` object is placed into a display container and becomes part of the display list, the same as non-text objects.

Loading external text is a little more involved than loading graphic and SWF files. For one thing, you need to use an *event listener* to determine when the text file has completed loading, and only after the loading has completed can you place the text into a `TextField` instance. As such, you'll need to use an event listener that signals (indicates with an `Event.COMPLETE` constant) when data are loaded and ready to be placed into a text field.

For this particular example, you'll see how to use a class by referencing it from a script in the timeline using the Actions panel. Then, you'll see how to do the same thing from a special file whose job it is to serve as a link between the class and the Flash document's Document class.

To get started, type in the script in Example 5 and save it as `TextLoader.as`.

Example 5. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;
    import flash.events.DataEvent;
    import flash.events.Event;
    import flash.net.URLLoader;
    import flash.net.URLRequest;
    import flash.text.TextField;
    import flash.text.TextFieldType;

    public class TextLoader extends Sprite
    {
        public function TextLoader(tfile:String):void
        {
            var txtLoader:URLLoader = new URLLoader();
            txtLoader.addEventListener(Event.COMPLETE,nowLoaded);
            var fileNow:URLRequest = new URLRequest(tfile);
            txtLoader.load(fileNow);
        }
        private function nowLoaded(event:Event):void
        {
            var loadCheck:URLLoader = URLLoader(event.target);
            var showText:TextField=new TextField();
            showText.width=300;
            showText.height=300;
            showText.type=TextFieldType.DYNAMIC;
            showText.multiline=true;
            showText.wordWrap=true;
            showText.text=loadCheck.data;
            this.addChild(showText);
        }
    }
}
```

The script consists of two key parts: the `txtLoader` function and the `nowLoaded` function. The class constructor (`TextLoader` function) has a single parameter, a string where the user places the name (URL) of the text field to be loaded. Using an `URLRequest` instance, the text file is loaded. Using an event listener, the `URLLoader` instance awaits the `COMPLETE` event, which is a `DataEvent` constant. When the load is completed, the private function (one that is visible only to references in the same class) fires the script that creates a text field and places the loaded text (`URLLoader` instance `data` property) into the text field. The `addChild()` function wraps up the process by displaying the text field.

This class has a level of abstraction that allows for greater flexibility in its implementation. Because the constructor class provides a parameter, it's easy to use the script to add any text file you want. This class is accessed directly from the Flash document using code placed on the timeline. Place the following script in the timeline of a Flash document by selecting the first keyframe and adding the code to the Actions panel:

```
var textLoad:TextLoader=new TextLoader("Schopenhauer.txt");
this.addChild(textLoad);
textLoad.x=100;
textLoad.y=100;
```

Name this file as `TextLoader.fla` and save it in the same folder as the `TextLoader.as` file. Also, in the same folder, add a text file with any text you'd like to add. For example, this script uses a reference to the following quote saved in a text file:

Quote

The discovery of truth is prevented more effectively, not by the false appearance things present and which mislead into error, not directly by weakness of the reasoning powers, but by preconceived opinion, by prejudice.

—Arthur Schopenhauer

You can use any text file you want, but make sure you replace the string, `Schopenhauer.txt` with the name of your text file.

In looking at the way in which the `TextField` instance is constructed the following lines, describe the nature of the text field:

```
showText.type=TextFieldType.DYNAMIC;
showText.multiline=true;
showText.wordWrap=true;
```

The `TextField.type` property is assigned one of two `TextFieldType` class constants: `DYNAMIC` or `INPUT`. The class isn't constructed as most are, but instead acts like a property value. Also, when adding text files, you'll want to set the Boolean values to `true` for both the `multiline` and `wordWrap` properties of the `TextField` class. The default `TextField` values are for a single line and do not include automatically wrapping words to the next line.

Instead of using code generated in the timeline, you can avoid the unnecessary package loading by simply having a reference to an implementation script. Often saved as `Main.as`, this script can be used to implement a class you have built. Essentially, the "main" class instantiates the elements of another class. Open a new ActionScript file and add the following script:

```
package
{
    import flash.display.Sprite;
    public class Main extends Sprite
    {
        public function Main():void
        {
            var textLoad:TextLoader=new TextLoader("Schopenhauer.txt");
            this.addChild(textLoad);
            textLoad.x=100;
            textLoad.y=100;
        }
    }
}
```

Save the script as `Main.as` and, in a new Flash document file, type `Main` in the Document class window. When you test it, you'll see exactly the same text message as you did when you placed the script in timeline and wrote the code in the Actions panel. In the example, we used the quote from Schopenhauer, and if you used the Schopenhauer text in the first test, you'll see it again. All that has changed between the first and second examples is the method used to run the test.

Formatting text

When working with text fields, you'll often want something other than the default values for the `TextField` properties. For example, the `wordWrap` property is set to `false` as is the `multiline` property. You may want to change those values to `true` as well as other properties in that class. ActionScript 3.0 has a `TextFormat` class very similar to that found in earlier versions of ActionScript. However, some new wrinkles have been introduced to text formatting and are worth examining.

The basics of formatting text in a text field center around the `TextFormat` class and the `StyleSheet` class. This section examines the `TextFormat` class; the next section examines HTML text, which is where you'll see how the `StyleSheet` class is used for formatting text.

The `TextFormat` class contains a number of properties to which you assign values. Then, you assign the `TextFormat` instance to the `TextField` instance to apply the format to all of the text in the `TextField` instance. Use the `TextField.defaultTextFormat` property to assign a `TextFormat` instance using the following format:

```
myTextField.defaultTextFormat=myTextFormat;
```

Example 6 shows a simple example of how to use the `TextFormat` class with the `TextField` class. Also note the inclusion of the new `TextFieldAutoSize` class. This new class is used to automatically set the size of the text field to

accommodate the amount of text used. Open a new ActionScript file and enter the code in Example 6. Save the file as `Lady.as`, and in a Flash document, type in `Lady` as the Document class. When you test it, you should see the Dorothy Parker quote in a red, 16-point Comic Sans MS font. The quotation dash in front of "Dorothy Parker" is generated from the Unicode, `\u2015`. Using the same format, you can add any Unicode symbols you need.

Example 6. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;
    import flash.text.TextField;
    import flash.text.TextFormat;
    import flash.text.TextFieldAutoSize;

    public class Lady extends Sprite
    {
        public function Lady():void
        {
            var fourMore:String="My land is bare of chattering folk;\n";
            fourMore+="the clouds are low along the ridges\n";
            fourMore+="and sweet's the air with curly smoke,\n";
            fourMore+="from all my burning bridges.\n\n";
            fourMore+="\u2015Dorothy Parker";

            var ladyFormat:TextFormat=new TextFormat();
            ladyFormat.font="Comic Sans MS";
            ladyFormat.color=0xcc0000;
            ladyFormat.size=16;

            var ladyText:TextField=new TextField();
            ladyText.autoSize=TextFieldAutoSize.LEFT;
            ladyText.defaultTextFormat=ladyFormat;
            ladyText.text=fourMore;
            this.addChild(ladyText);
            ladyText.x=100;
            ladyText.y=90;
        }
    }
}
```

Unlike previous versions of ActionScript, the `autoSize` property is assigned a `TextFieldAutoSize` constant. Also, the `newline` statement is no longer used in ActionScript 3.0. Instead, to add a new line, use the `\n` escape character embedded in the text string.

HTML text and style sheets

ActionScript 3.0 provides a subset of the Cascading Style Sheet (CSS) properties used with setting styles in HTML. While it's possible to apply CSS to HTML text, only 13 CSS properties are available for use in ActionScript 3.0. Table 2 provides a list of the usable CSS properties supported by ActionScript 3.0.

Creating a `StyleSheet` instance in ActionScript 3.0 is different from the way it was done in ActionScript 2.0. Different styles within a style sheet are created using an `Object` instance, and each CSS property is assigned as a property of the `Object`. For example, the following shows the process for setting up a CSS style in ActionScript 3.0:

```
var poem:Object = new Object();
poem.textAlign = “center”;
poem.color = “#9900aa”;
poem.marginLeft=20;
```

The preceding code is equivalent to the following in CSS:

```
.poem {
       text-align: center;
       color: #9900aa;
       margin-left: 20pt;
}
```

Once you have set up one or more styles, you “load” the style sheet using the `StyleSheet.setStyle()` method, as shown here:

```
var myStyleSheet:StyleSheet = new Style;
myStyle,setStyle(“.poem”, poem);
```

Finally, you apply the style sheet similar to the way you would when applying CSS to an HTML page. Example 7 shows how to create all the different aspects of ActionScript 3.0 CSS and apply them in a script.

Table 2. Available CSS Properties for ActionScript 3.0

ActionScript CSS Property	Possible Values
`Color`	Hexadecimal only in the format #RRGGBB (e.g., `#00aa33`); color names are not acceptable
`display`	`inline`, `block`, or `none`
`fontFamily`	Name or comma-separated names of font (Arial Bold, Verdana)
`fontSize`	Number only, without unit specification (e.g., `pt`, `px`)
`fontStyle`	`normal` or `italic`
`fontWeight`	`normal` or `bold`
`kerning`	`true` or `false` (very limited, available only for window-developed SWF files)
`letterSpacing`	Number only, without unit specification
`marginLeft`	Number only, without unit specification (e.g., `pt`, `px`)
`marginRight`	Number only, without unit specification (e.g., `pt`, `px`)
`textAlign`	`left`, `center`, `right`, or `justify`
`textDecoration`	`none` or `underline`
`textIndent`	Number only, without unit specification (e.g., `em`)

Open a new ActionScript file and add the script in Example 7. Save it as `Stylish.as`. Next, open a new Flash document and save it as `Stylish.fla` in the same folder. Type in `Stylish` in the Document class and test the script.

Example 7. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;
    import flash.text.StyleSheet
    import flash.text.TextField;
    import flash.text.TextFieldAutoSize;

    public class Stylish extends Sprite
    {
        public function Stylish():void
        {
             var selfReliance:StyleSheet=new StyleSheet();
```

```
            var quote:Object=new Object();
            quote.color="#999999";
            quote.marginLeft=20;
            quote.fontSize=16;
            quote.textAlign="left";

            var signature:Object=new Object();
            signature.fontStyle="italic";
            signature.color ="#cc0000";
            signature.fontSize=16;
            signature.textIndent=30;

            var qStr:String="<body><p class='quote'>A foolish consistency ";
            qStr+="is the hobgoblin of little minds,\n";
            qStr+="adored by little statesmen and philosophers ";
            qStr+="and divines.\n";
            qStr+="With consistency a great soul has simply ";
            qStr+="nothing to do.</p>\n\n";
            qStr+="<p class='signature'>Ralph Waldo Emerson</p></body>"

            selfReliance.setStyle(".quote",quote);
            selfReliance.setStyle(".signature",signature);

            var waldo:TextField=new TextField();
            waldo.autoSize=TextFieldAutoSize.LEFT;
            waldo.styleSheet=selfReliance;
            waldo.htmlText=qStr;
            this.addChild(waldo);
            waldo.x=100;
            waldo.y=80;
        }
    }
}
```

When testing the script, you'll see everything formatted as planned. Figure 6 shows how the CSS formatted the text. Were it not for the fact that you've seen the code, you probably couldn't tell the difference between using a `StyleSheet` and HTML text and using a `TextFormat` instance with regular text.

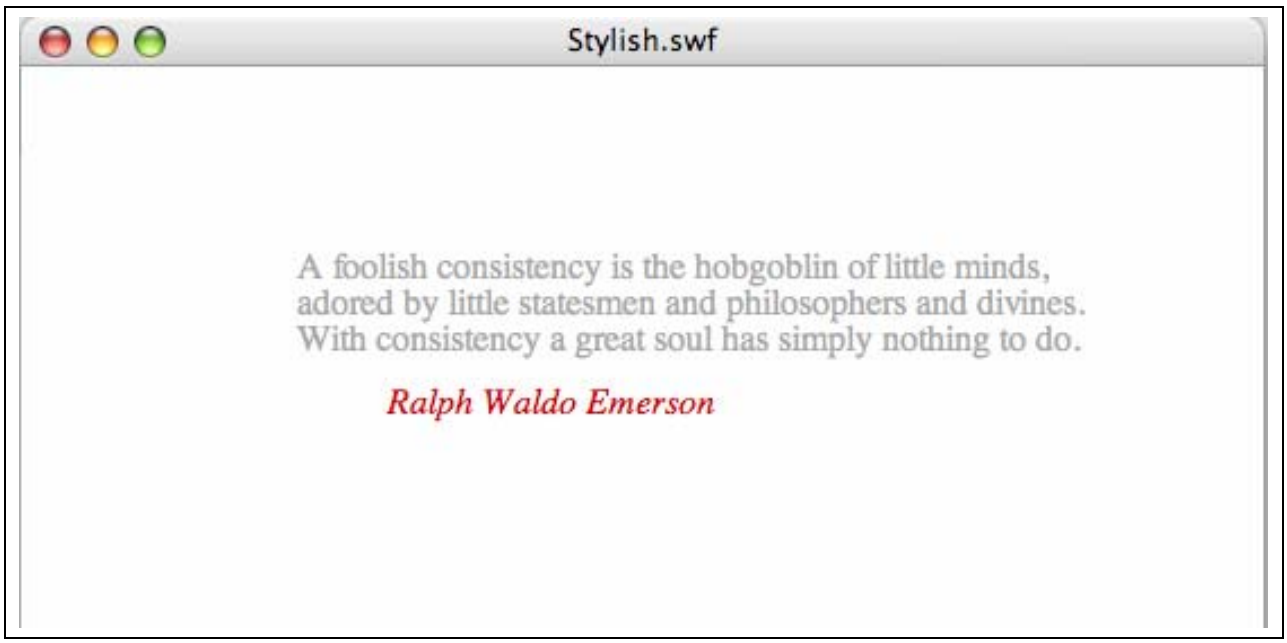

Figure 6. Text displayed using a StyleSheet and HTML text

Working with Movie Clips and Buttons

This section examines two familiar symbols in Flash: movie clips and buttons. As you will see when working with ActionScript 3.0, you also have an option to work with `Sprite` objects, of which `MovieClip` objects are a subclass. Likewise, you will need to know something about a new class called `SimpleButton` to work effectively with buttons. Both movie clips and buttons inherit a number of their classes from the `DisplayObject` class, and in looking at the properties of both classes, the inherited classes are listed as well.

MovieClip Objects

To get started, you have to re-orient your thinking about how movie clips created on the stage are connected to ActionScript. Essentially, when you convert a drawing into a `MovieClip` or `Button`, you are creating a class. The class name is whatever you give for the `Symbol Name`.

When you create a movie clip using the drawing tools (instead of creating it using ActionScript 3.0), the Symbol dialog box provides an "Export for ActionScript" checkbox. When you enable that option, the `Symbol Name` appears in the Class window, and the Base class shows the path to the class base. For example, Figure 8 shows what you would see if you name the symbol, *Truck*, and enable the "Export for ActionScript" checkbox.

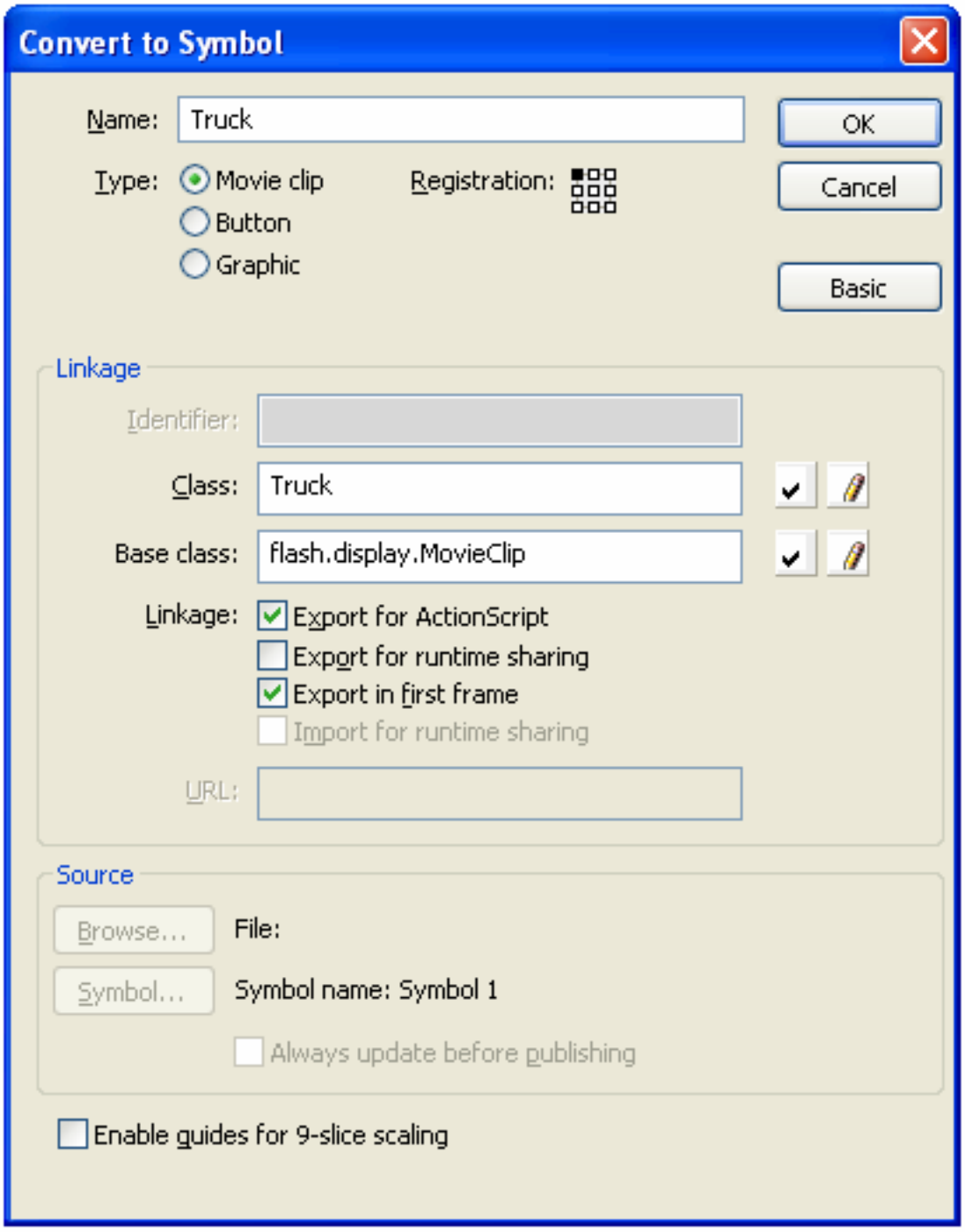

Figure 8. Symbol names become class names for Movie clips and buttons

To work with movie clips created on the stage with drawing tools and ActionScript 3.0, let's go through an example beginning with a drawing of a truck. After completing the drawing and adding the necessary other parts, you'll see how to convert the `MovieClip` object into a class. To get started, follow these steps:

1. Open a new Flash document and save it as `Truck.fla`.

Using Figure 9 as a guide, draw the cab and the cargo box of the truck.

Select the image and press the F8 key to transform the drawing into a Symbol. Fill out the Symbol dialog box as shown in Figure 8.

Double-click the image to enter the Symbol editor and add two layers, giving your project three layers in total. Name the layer with the truck image, *Truck*, another *Wheel*, and the third one *Text*. Order the layers from top to bottom, *Text*, *Truck*, *Wheel*. Lock the *Truck* layer.

Click the *Wheel* layer and draw a wheel. Select the wheel and press F8 to convert it to a Movie clip Symbol, but *do not* select any of the Linkage checkboxes.

In the Name window, type in *Wheel* and click OK. Place the two wheel objects beneath the truck as shown in Figure 9. Provide the instance name `wheel1_mc` for the front wheel and `wheel2_mc` for the rear wheel.

Click on the *Text* layer and add a dynamic text field over the cargo area of the truck, as shown in Figure 9. Provide it with the instance name `truck_txt` in the Properties panel. Lock the layer.

Double-click on one of the *Wheels* to enter the Symbol edit mode. Add a layer and name it *Rotation*, and name the original layer *Wheel*. Lock the *Wheel* layer.

Draw four yellow lines over the tire as shown in Figure 9. Select the lines and press F8 to convert it to a Symbol. In the Symbol dialog box, name the symbol *Rotate* and select Movie clip as the Behavior. Click OK. Give the *Rotate* object the instance name, `rotate_mc`.

In the *Wheel* edit mode, add 30 frames to both layers. In the *Rotation* layer, add keyframes to frames 7, 15, 22, and 30. Using the Free Transform tool, rotate the image counter-clockwise (CCW) -90° in Frame 7, -180° in Frame 15, -270° in Frame 22, and back to 0 degrees in Frame 30. Add Motion tweens to Frames 0, 7, 15, and 22.

Return to the main stage and test the movie clip. The wheels should appear to rotate. Remove the movie clip from the stage so that when you add the class instance of the truck using ActionScript, you won't have another truck object already on the stage. The stage is now empty, but you should see three Movie Clip symbols in the Library, as shown in Figure 9.

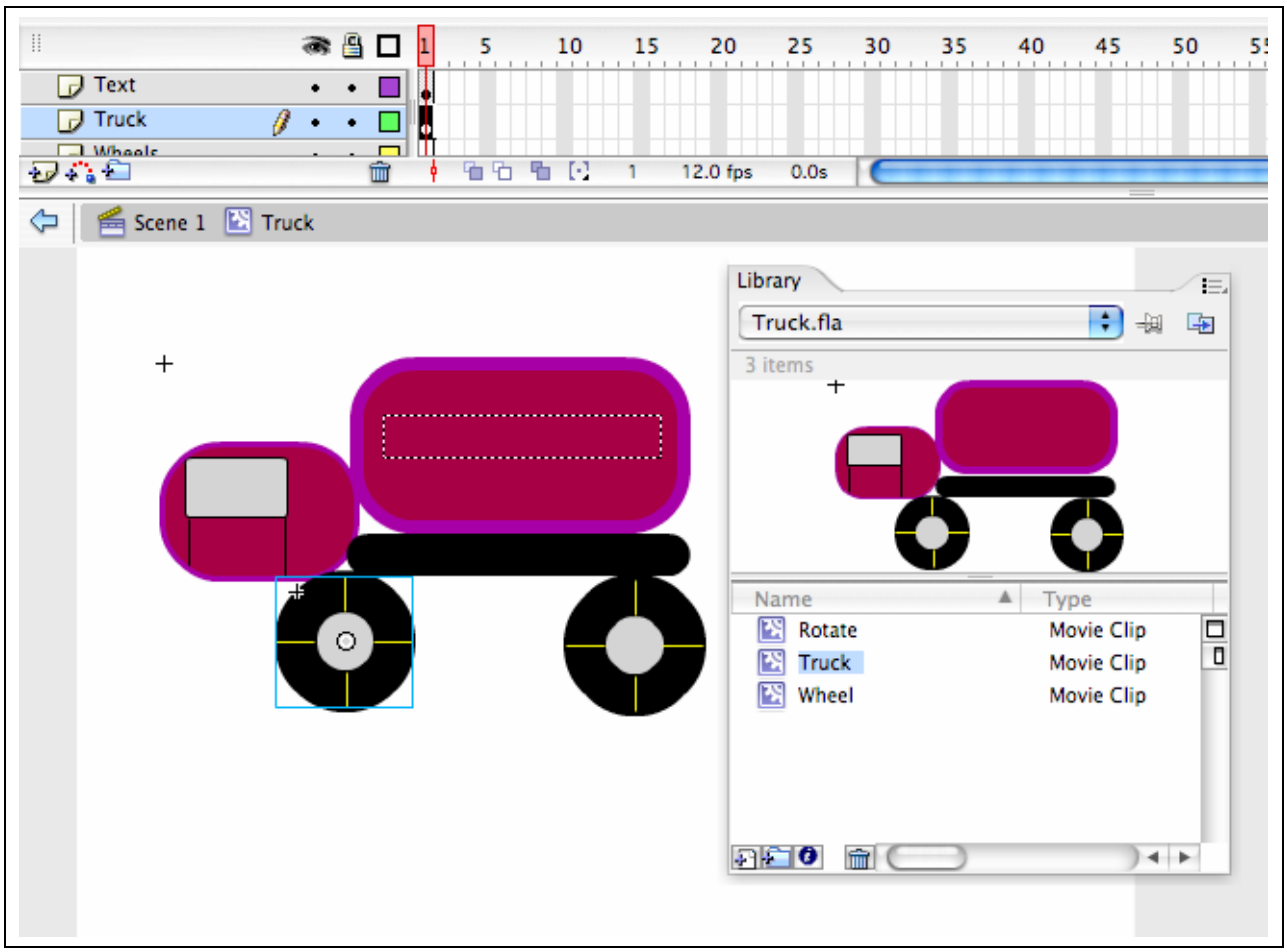

Figure 9. Compound movie clip on stage (before removal) and movie clips in Library

Before going on to look at some code, let's consider the objects on the stage in terms of coded elements. The actual Truck movie clip is really just a graphic—a Sprite. The timeline is not used, so why make the class a `MovieClip` class and load the extra code to deal with a timeline for truck? Essentially, the movie clip is a container where we have placed a text field and a couple of other movie clips. So, as a class, the movie clip really doesn't need to be anything other than a Sprite.

You cannot create a `Sprite` on the stage. So you need to convert a `MovieClip` into a `Sprite` by other means. Here's how to do that:

1. Select the Truck icon in the Library panel.
2. Click the Information icon in the Library panel.
3. In the Base class window, change `flash.display.MovieClip` to `flash.display.Sprite`, as shown in Figure 10.
4. Click OK.

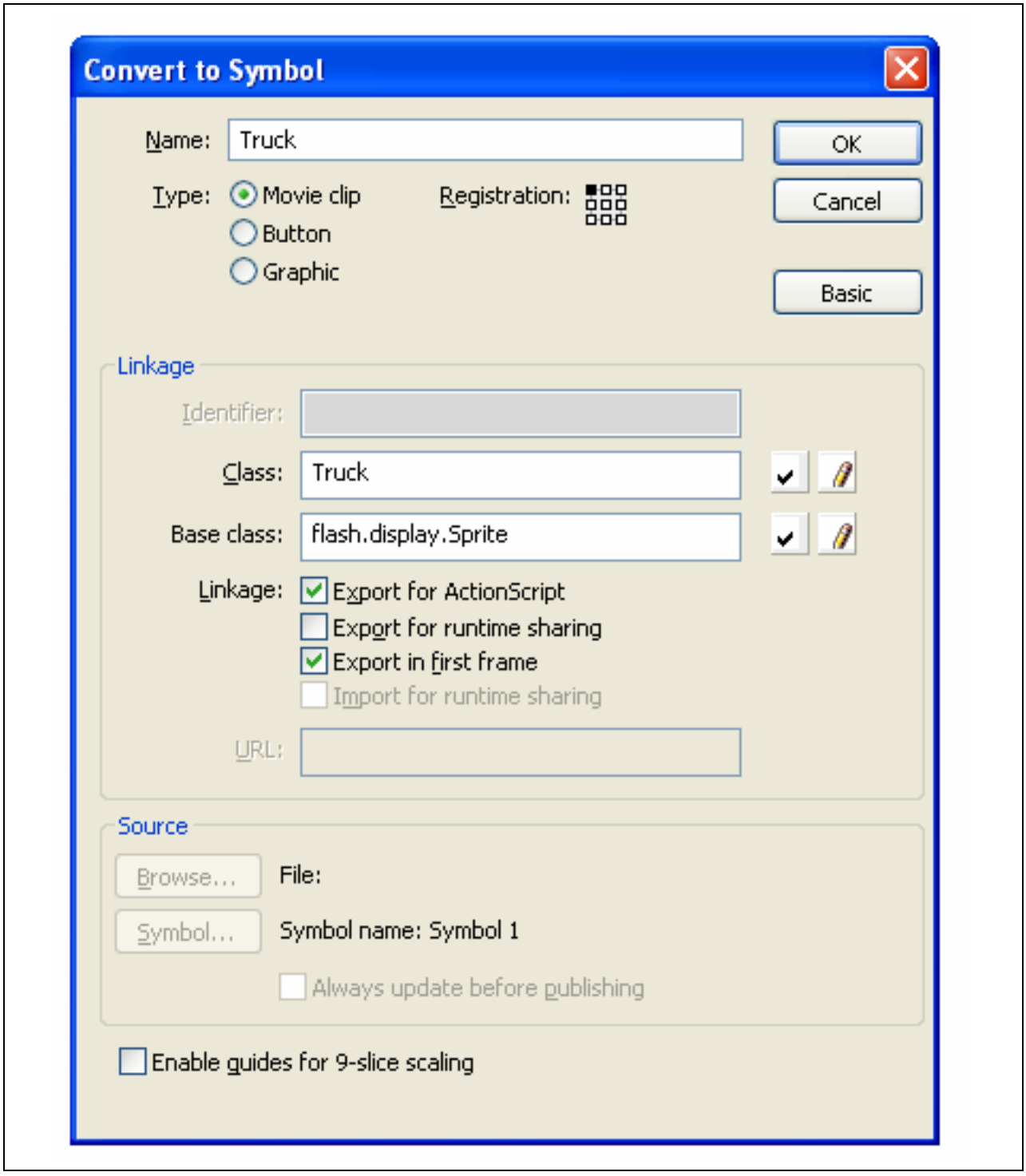

Figure 10. Converting a MovieClip to a Sprite

That's all there is to it. The Truck movie clip is now a Sprite object. So, when it comes time to writing a script that involves your `Truck` class, just know that it's a Sprite. To see how this works, open up a new ActionScript file and save it as `PlayTruck.as`. Next, in the Document class of the `Truck.fla` file, type in `PlayTruck`. Now type in the code shown in Example 8.

Example 8. **(download this code sample at http://examples.oreilly.com/actionscript3qr)**

```
package
{
    import flash.display.Sprite;
    public class PlayTruck extends Sprite
    {
        public function PlayTruck():void
        {
            var javaTruck:Truck=new Truck();
            this.addChild(javaTruck);
            javaTruck.truck_txt.text="Drink Java!";
            //javaTruck.wheel1_mc.rotate_mc.visible=false;
            //javaTruck.wheel2_mc.stop();
            javaTruck.x=(stage.stageWidth/2)-(javaTruck.width/2);
            javaTruck.y=(stage.stageHeight/2)-(javaTruck.height/2);
        }
    }
}
```

Test the script. You should see the truck figure in the center of the stage with the wheels spinning with the message, “Drink Java!,” displayed on the side of the truck.

The important aspect of this example is that the `Truck` class is treated just like any other class, and the text area is addressable as a property of a `Truck` class instance. Keeping in mind that the original Truck movie clip was an extension of the `MovieClip` class but changed to a `Spite` extension, you can see that everything works as expected.

By adding the commented-out lines to the script, you will see that you can control the `Wheel` movie clip’s properties and methods as well. Remove the comment operators (`//`) from the following two lines:

```
//javaTruck.wheel1_mc.rotate_mc.visible=false;
//javaTruck.wheel2_mc.stop();
```

Now when you test the movie, the rotation lines are invisible on the front wheel, and the rear wheel rotation lines are visible, but have stopped moving.

As you can see, you can create all of the display items you want on the stage, nesting them as deep as you want. Then, by assigning the top-level symbol a class status, you can access all of the properties and methods contained in that class. You can assign a base class as either a `MovieClip` (default) or as a `Sprite`.

Of course, you can code your own movie clip without using the Flash IDE tools. In the context of using ActionScript, that’s pretty easy. You can dynamically add text or graphics by coding shapes and text fields right into a class that extends the `MovieClip` class. However, unless you need the timeline, you’re better off coding text fields and shapes in a Sprite object, as you saw earlier in Figure 3.

When working with `MovieClip` properties, you have to work with `MovieClip` properties as well as the properties it inherits. Because many of the names have changed in the property names, take a look at Table 3 for an overview of the `MovieClip`'s properties and those properties it has inherited.

Table 3. MovieClip Properties and Inherited Properties

Property	Data Type	Function
`accessibilityProperties`	AccessibilityProperties	Display object's current accessibility options
`alpha`	Number	Opaque level (obversely, transparency level)
`blendMode`	String	BlendMode class value expressed as a constant
`buttonMode`	Boolean	Button mode on or off
`cacheAsBitmap`	Boolean	Flash Player caches an internal bitmap representation of the display object if set to true
`constructor`	Object	Reference for class if object is class instance; otherwise, the reference for constructor function for a given object instance.
`contextMenu`	ContextMenu	Objects' associated context menu
`currentFrame`	int	Playhead's current frame number (read-only)

`currentLabel`	String	Playhead's current frame label (read-only)
`currentLabels`	Array	Scene's label array. (read-only)
`currentScene`	Scene	`MovieClip` instance's current scene (read-only)
`doubleClickEnabled`	Boolean	Object receives `doubleClick` event
`dropTarget`	DisplayObject	Target `DisplayObject` where sprite is dropped or dragged
`enabled`	Boolean	`MovieClip` instance is enabled if `true`
`filters`	Array	Filter object's indexed array, currently associated with the display object
`focusRect`	Object	Null value indicates object obeys `stageFocusRect` property
`framesLoaded`	int	Current number of frames loaded in SWF file (read-only)
`graphics`	Graphics	Associated with `Graphics` object that, belonging to this Sprite, allows vector drawing to take place

`height`	Number	`DisplayObjec`'s height
`hitArea`	Sprite	Specify another Sprite as the hit area for Sprite
`loaderInfo`	LoaderInfo	Obtains information about display object's file being loaded
`mask`	DisplayObject	Request to use the called object as a mask to calling object
`mouseChildren`	Boolean	Returns `true` if children of the object are mouse enabled
`mouseEnabled`	Boolean	Returns `true` if this object receives mouse messages
`mouseX`	Number	Mouse `x` coordinate
`mouseY`	Number	Mouse `y` coordinate
`name`	String	`DisplayObjec`'s instance name
`numChildren`	int	Object's number of children
`opaqueBackground`	Object	Indicates opaque display and background color
`parent`	DisplayObjectContainer	Reference to container object containing display object

`prototype`	Object	Class or function object's prototype reference
`root`	DisplayObject	Root of loaded SWF file owning `DisplayObject`
`rotation`	Number	Rotation value (0-360)
`scale9Grid`	Rectangle	Scaling grid currently in effect
`scaleX`	Number	Horizontal percent of `DisplayObject`
`scaleY`	Number	Vertical percent of `DisplayObject`
`scenes`	Array	Scene array with name, total number of frames, and scene and frame names for `MovieClip`'s scene (read-only)
`scrollRect`	Rectangle	Boundary of a display object's scroll rectangle
`soundTransform`	SoundTransform	Sound controls in this Sprite
`stage`	Stage	Display object's stage
`tabChildren`	Boolean	`true` if display object's children are tab enabled
`tabEnabled`	Boolean	`true` if object is tab enabled

`tabIndex`	int	Tab order in SWF file
`textSnapshot`	TextSnapshot	`TextSnapshot` object of current `DisplayObjectContainer` instance
`totalFrames`	int	Total number of frames in the movie clip instance (read-only)
`trackAsMenu`	Boolean	`true` means that `SimpleButton` or `MovieClip` objects are able to get mouse release events
`transform`	Transform	Object properties with display object's matrix, color transform, and pixel bounds
`useHandCursor`	Boolean	`true` enables hand cursor appearance when the mouse rolls over a sprite with `buttonMode` set to `true`
`visible`	Boolean	`false` hides object
`width`	Number	`DisplayObject`'s width
`x`	Number	`DisplayObject`'s *x* coordinate
`y`	Number	`DisplayObject`'s *y* coordinate

Scripting sprite classes

You can dynamically create movie clips using ActionScript, but because you cannot dynamically script in frames and keyframes, most of what you will be doing with objects created with ActionScript 3.0 will be with Sprites. If there's no reason for the overhead in a timeline, then use a `Sprite` instead of a `MovieClip`.

One of the more important features of Flash is the ability to drag `MovieClip` instances. However, the same thing can now be done with `Sprite` objects. A common use of the `Sprite` object is for creating a slider control. By sliding a movie clip and tracking the horizontal or vertical position, it's possible to change the properties of other objects, such as the volume level in a sound application. To accomplish this, you'll need to do the following:

1. Create a "groove" line. This is a shape that indicates where the slider may slide. This is easy because all you have to do is generate a line. With ActionScript 3.0, this can be done with a `Shape` object and graphic drawing methods found in the `flash.display.Graphics` package.
2. Create a "lever" `Sprite` object. This requires that the script draw the lever; a rounded rectangle created with the `drawRoundRect()` graphic drawing method will take care of that.
3. Finally, an event handler needs to be added to the lever object so it can be moved along the groove line. The `startDrag()` method has changed from earlier versions of ActionScript, so you'll need a `Rectangle` instance from the `flash.geom` package instead. The new `startDrag()` has the format,

   ```
   startDrag(lockCenter:Boolean, bounds:Rectangle);
   ```

   To specify the bounds, create a `Rectangle` instance defining the allowable area where the lever can be dragged. Since the groove line is a 1-pixel high "rectangle," you'll need to define the rectangle as one with the width and height of the groove line.

Everything you need is in Example 9. Note that the script imports four different classes. The use of both private variables and functions exposes those variables and functions only to the class they are in.

Example 9. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;
    import flash.display.Shape;
    import flash.events.MouseEvent;
    import flash.geom.Rectangle;

    public class Slider extends Sprite
```

```
    {
        private var posx:uint=200;
        private var posy:uint=100;

        public function Slider():void
        {
            doGroove();
            doLever();
        }

        private function doGroove():void
        {
            var groove:Shape =new Shape();
            groove.graphics.lineStyle(1,0x000000);
            groove.graphics.moveTo(10,100);
            groove.graphics.lineTo(210,100);
            addChild(groove);
            groove.x=posx;
            groove.y=posy;
        }
        private function doLever():void
        {
            var lever:Sprite =new Sprite();
            lever.graphics.beginFill(0xcccccc);
            lever.graphics.lineStyle(2,0xaaaaaa);
            lever.graphics.drawRoundRect(10,80,10,40,6);
            lever.graphics.endFill();
            addChild(lever);
            lever.x=posx;
            lever.y=posy;

            function startLever(event:MouseEvent):void
            {
                var dragRec:Rectangle=new Rectangle(posx,posy,200,0);
                lever.startDrag(false,dragRec);
                trace(Math.round((lever.x-posx)/2));
            }
            lever.addEventListener(MouseEvent.MOUSE_DOWN,startLever);

            function stopLever(event:MouseEvent):void
            {
                lever.stopDrag();
                trace(Math.round((lever.x-posx)/2));
            }
            lever.addEventListener(MouseEvent.MOUSE_UP,stopLever);
        }
    }
}
```

The following steps show what you need to do:

1. Get started by opening a new Flash document and saving it as `Slider.fla`.

   Next, open an ActionScript file and save it as `Slider.as` in the same folder as `Slider.fla`.

   Type in the code in Example 9, and save it as `Slider.as`.

Finally, type in `Slider` as the Document class in the `Slider.fla` Properties panel.

When you test the application, you should see values from 0-100 in the Output window as you move the slider from left to right. You can change that range to whatever you want by working with the base value of 200, the number of pixels that the sliding lever can traverse. Figure 12 shows what you will see when you test the application.

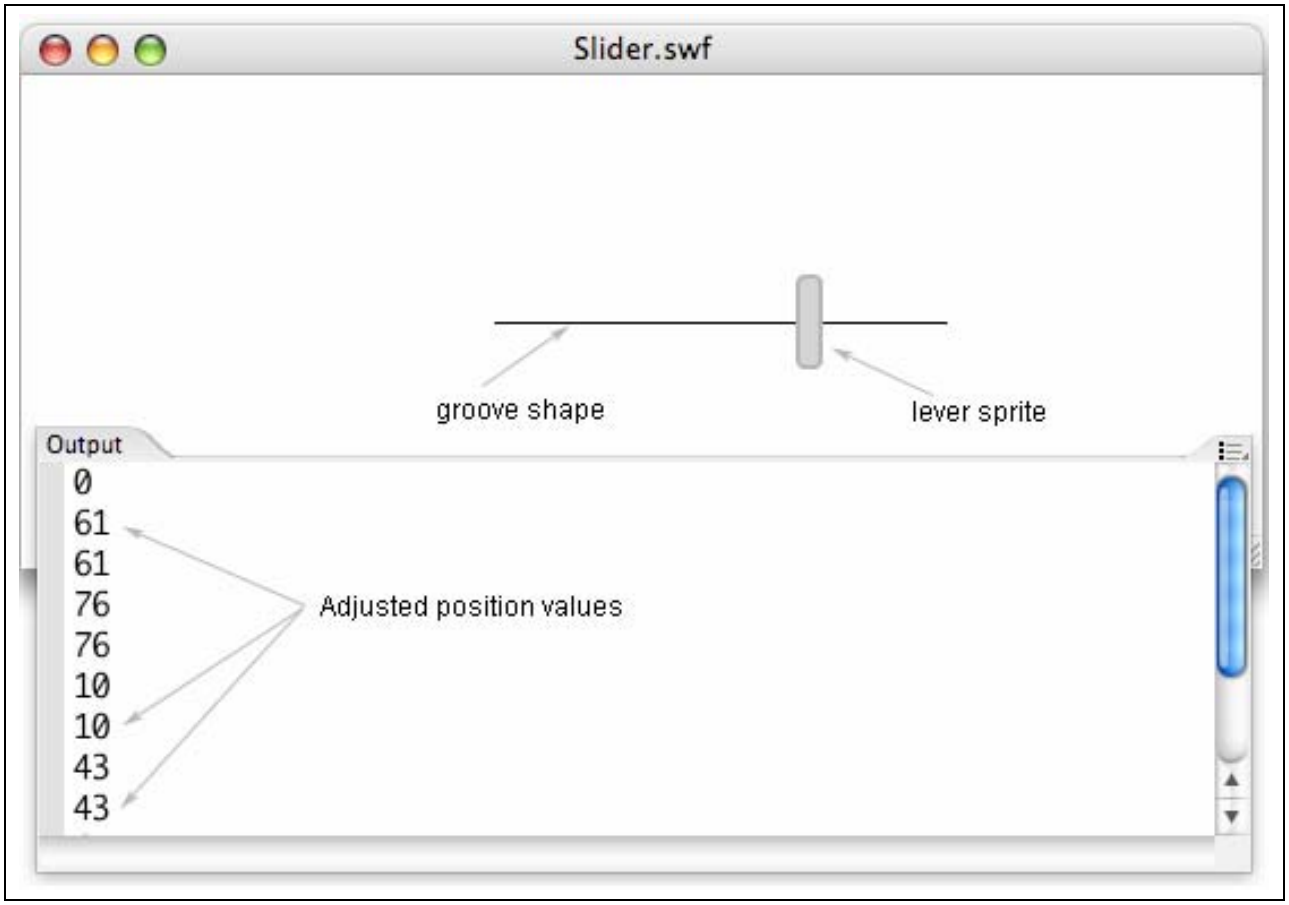

Figure 12. Slider application made of Sprite and Shape instances generating position values

Buttons

Like `MovieClip` and `Sprite` objects, you can draw buttons on the stage and convert them to symbols and then reference them as a class. The base class for a button object is the `SimpleButton` class. If you're familiar with the buttons you create on the stage, you will be somewhat familiar with this new class. However, using ActionScript 3.0, you can do far more in terms of creating your own buttons. To get started, though, create a button on the stage:

1. Open a new Flash document and save it as `GoButton.fla`.

2. Using the Oval tool, draw a 36-pixel circle using two different colors for the fill and stroke; set the stroke width to 2.25 pixels.
3. Select the circle and press the F8 key to open the Convert to Symbol dialog box. Type in `GoButton` for the Name and select Button as the type, as shown in Figure 11.
4. Enable the "Export for ActionScript" checkbox in the Linkage group. The class name `GoButton` appears and the Base class is listed as `flash.display.SimpleButton`, as shown in Figure 11.
5. Click OK.
6. Remove the button from the stage. Select the button in the Library panel and right-click (Control-click on the Mac) the icon. Select Edit from the Context menu.
7. Select the Over, Down, and Hit frames in turn and press F6 to insert a keyframe for each. Click the Over frame and reverse the fill and stroke colors. Click the Down frame and change the fill color to a color other than the current one. Click the Hit frame and draw an oval or rounded rectangle to cover the button and about 50 pixels to the right of the button. (This gives the button an extended hit area to the right.) Click the Scene 1 icon to exit the Symbol Edit mode.

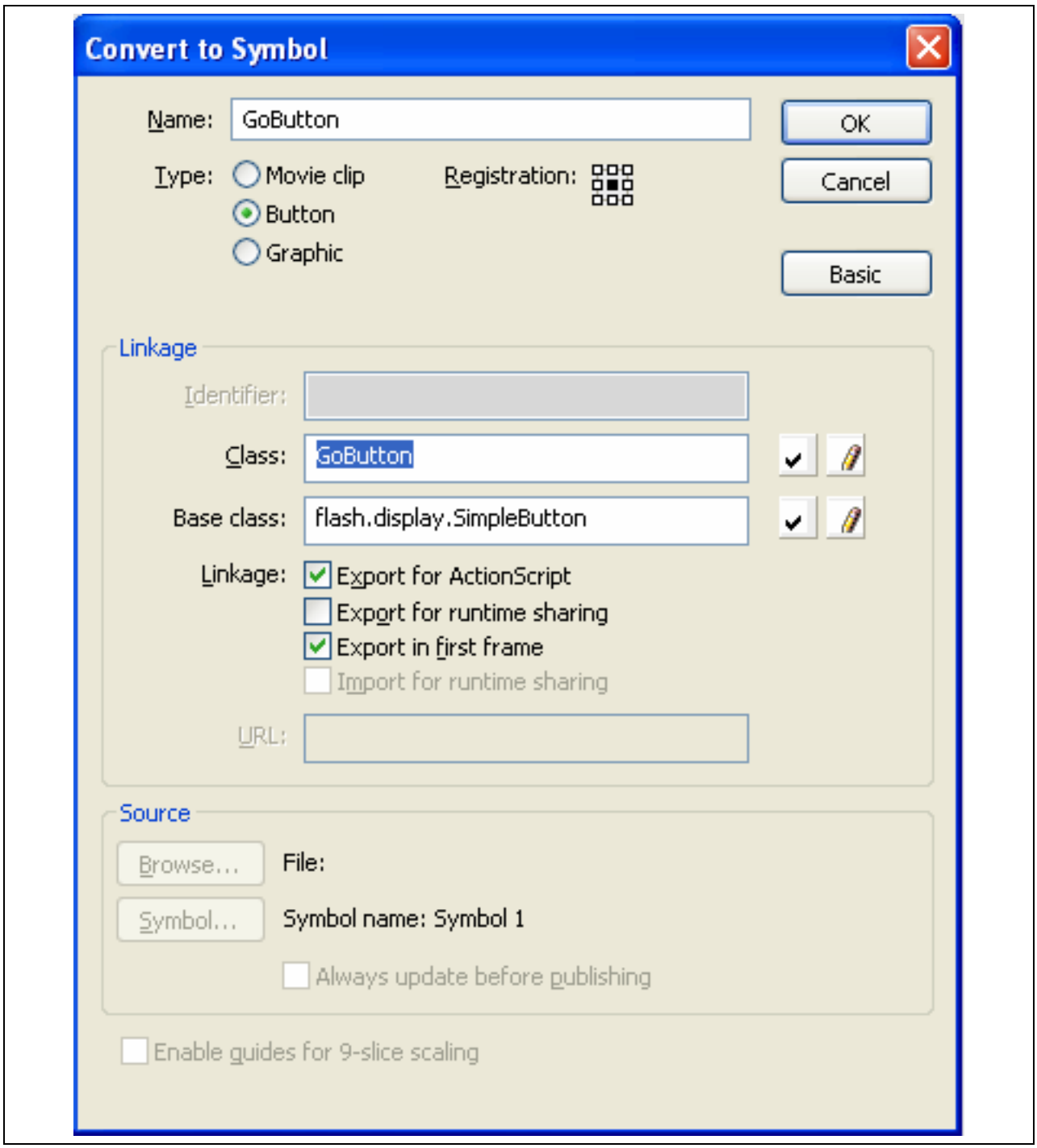

Figure 11. Creating a SimpleButton class

8. Open the Properties panel and type in `ButtonWork`, which is the name of the class that places the buttons on the stage.
9. Open a new ActionScript file and save it as `ButtonWork.as` in the same folder as the Flash document file, `GoButton.fla`.
10. Add the script shown in Example 10 and save the file.

Example 10. **(download this code sample at http://examples.oreilly.com/actionscript3qr)**

```
package
{
    import flash.display.Sprite;
    public class ButtonWork extends Sprite
    {
        public function ButtonWork():void
        {
            var navBtn1:GoButton=new GoButton();
            var navBtn2:GoButton=new GoButton();
            var navBtn3:GoButton=new GoButton();

            this.addChild(navBtn1);
            this.addChild(navBtn2);
            this.addChild(navBtn3);

            navBtn1.x=100;
            navBtn2.x=100;
            navBtn3.x=100;

            navBtn1.y=100;
            navBtn2.y=150;
            navBtn3.y=200;

        }
    }
}
```

When you save the file and test it, you'll see three buttons on the stage lined up vertically. Now you know how to create a button using the Flash tools and use ActionScript 3.0 to place multiple instances of it on the stage. To make the button work, though, you'll need to add event listeners so the button will do something when it's clicked or rolled over. Keep the button class you just created because you'll need to see how to connect it to a mouse event in the next section.

Connecting events to your buttons

To understand events in ActionScript 3.0, assume as little as possible about using events in previous versions of ActionScript. Not surprisingly, you have a package that contains the different events you'll be using, `flash.events`. So whenever you use an event (which is often), be sure to remember to import the necessary `flash.events` package and the specific class. ActionScript 3.0 has 21 different event classes, and most of these classes contain event constants that reference a particular event. When using *any class with an event*, the following shows the general format:

```
objectInstance.addEventListener(EventClass.EVENT_CONSTANT, eventHandler);
```

For example, the following shows how a button can be linked to a mouse event:

```
myButton.addEventListener(MouseEvent.CLICK, moveMC);
```

The button instance named `myButton` adds the mouse `CLICK` listener to itself. A function named `moveMC` does something when the mouse is clicked while over the button instance. (Effectively, this works as a button click.) To see how this works, let's make some changes to the script shown in Example 10. Example 11 shows the changes; just add the new code to the `ButtonWorks.as` file, then save and test the code. (The added code is in bold.)

Example 11. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.events.MouseEvent;
    import flash.display.Sprite;
    public class ButtonWork extends Sprite
    {
        public function ButtonWork():void
        {
            var navBtn1:GoButton=new GoButton();
            var navBtn2:GoButton=new GoButton();
            var navBtn3:GoButton=new GoButton();

            this.addChild(navBtn1);
            this.addChild(navBtn2);
            this.addChild(navBtn3);

            navBtn1.x=100;
            navBtn2.x=100;
            navBtn3.x=100;

            navBtn1.y=100;
            navBtn2.y=150;
            navBtn3.y=200;

            navBtn1.addEventListener(MouseEvent.CLICK,jump);
            navBtn3.addEventListener(MouseEvent.CLICK,back);
            function jump():void
            {
                navBtn2.x+= 5;
            }
            function back():void
            {
                navBtn2.x+= -5;
            }

        }
    }
}
```

When you make the changes, be sure to add the line that imports the `MouseEvent` class to the file. This script moves the middle button to the left and right, but the click-handling function can do anything you want it to.

Aside from reading the `CLICK` event with a mouse, several other constants can be associated with the `MouseEvent` class. Table 4 lists the events that can be read by an object with an event listener associated with the `MouseEvent` class.

Table 4. MouseEvent class Constants

Constant Name	Event
`CLICK`	Mouse button is pressed and released (the left button on two-button mouse)
`DOUBLE_CLICK`	Mouse button is pressed twice in quick succession
`MOUSE_DOWN`	Mouse button is held down
`MOUSE_MOVE`	Mouse is currently changing positions
`MOUSE_OUT`	Mouse has moved the over position to a position no longer over the object
`MOUSE_OVER`	Mouse position is over the target, including the entire hit area of a button
`MOUSE_UP`	Mouse button has been down and this event is after it returns to the up position
`MOUSE_WHEEL`	Mouse wheel is rotated over target
`ROLL_OUT`	Moving mouse leaves target
`ROLL_OVER`	Moving mouse is over target

To get an idea of how the different mouse events work, simply change the event constants in Example 11. For example, change `CLICK` to `ROLL_OVER` on one of the buttons and notice the result you get.

Scripting a simplebutton

Unlike the `MovieClip` class, where you cannot add frames and keyframes, the new `SimpleButton` class can be used to create fairly robust buttons. As you will see, such buttons can have their own dynamic text and reaction to the mouse. In developing buttons using ActionScript, you need to consider the following elements:

1. What is the shape of your button? Will it be round, oval, rectangular, or some other shape? What `Shape` or `Sprite` properties will you need?
2. Will your button require a text label? Will the text be dynamic or static? Should the user have the option of using the same class with different text?
3. What kind of events will be used with the button?
4. Which states will you use? (See Table 5 for the `SimpleButton` properties.)

You may wonder, why bother with scripting a button when you can easily draw one on the stage? The answer is that you can control more aspects of a button when it's written with code. Furthermore, as a class, once you've spent time creating the

button, you can reuse the class all you want, changing details of the code to change characteristics of the button.

Before looking at an example for creating and using a scripted button, review Table 5 and take a look at all of the properties you can control directly in ActionScript 3.0.

Table 5. SimpleButton Properties

Property	Characteristics
`downstate`	Button is pressed down
`enabled`	Button is ready to use if `true`
`hitTestState`	Display object used as "live" area of button. May be a different size and shape than visible aspects of button.
`overState`	State when mouse is over button hit area
`soundTransform`	`SoundTransform` object connected to button
`trackAsMenu`	Boolean indicating display object can receive mouse release events
`upState`	State when mouse has been held down and then released or simply is not held down
`useHandCursor`	A Boolean value of `true` displays the hand cursor; `false` displays the arrow cursor

Example 12 shows a fairly robust button. However, the button is created without any events connected to it—concretely or abstractly. In part this simplifies matters, but more importantly, as you will see in Example 13, it allows greater flexibility when using the button class. Example 12 shows you how to build the button as an entity you can place in your application; Example 13 shows how you can flexibly place the button instances, uniquely label each instance, and set up an event for each instance.

Open a new ActionScript file and save it as `FlashBtn.as`. Then enter and save the script shown in Example 12.

Example 12. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;
    import flash.display.SimpleButton;
    import flash.display.Shape;
    import flash.text.TextFormat;
    import flash.text.TextField;
    import flash.text.TextFieldAutoSize;
```

```
    public class FlashBtn extends SimpleButton
    {
        public function FlashBtn(txt:String)
        {
            upState = new BtnState(0xfff56d, txt);
            downState = new BtnState(0xfff222, txt);
            overState= new BtnState (0xfff000,txt);
            hitTestState=upState;
            useHandCursor=true;
        }
    }
    class BtnState extends Sprite
    {
        private var btnLabel:TextField;
        public function BtnState(color:uint,btnLabelText:String):void
        {
            //Text field for button
            btnLabel = new TextField();
            btnLabel.text=btnLabelText;
            btnLabel.autoSize=TextFieldAutoSize.LEFT;
            //Text format for text field
            var format:TextFormat = new TextFormat("Verdana");
            format.size=12;
            btnLabel.setTextFormat(format);
            var btnWidth:Number=btnLabel.textWidth + 10;
            //Shape for button background
            var bkground:Shape = new Shape();
            bkground.graphics.beginFill(color);
            bkground.graphics.lineStyle(2,0xf3c716);
            bkground.graphics.drawRect(0,0,btnWidth,18);
            addChild(bkground);
            addChild(btnLabel);
        }
    }
}
```

As usual, your first order of business is to make sure that you import all of the packages and classes you need. Note that the script does not import any `flash.events` classes because the script that is building the button is not using them. In subsequent scripts that employ the button, `flash.events` will be employed and the import will occur in those scripts. However, be sure to get the necessary classes for the text objects and drawings you will be creating along with the `SimpleButton` class so you can use its properties in the script.

Note that the `FlashBtn` class extends `SimpleButton`. Therefore, you can assign values to the different `SimpleButton` properties (shown in Table 5) without having to declare them. In fact, the constructor for the class simply assigns five of the `SimpleButton` property values.

The second class (which can be treated as a private class, `BtnState`), is a `Sprite` extension that's made up of the display characteristics for the button. First, a text field and text format is added, and then a `Shape` draws a rectangle that defines the button's shape. The background image and text are then assigned as

values to the different states in the constructor function. The constructor function, used to instantiate class objects, can be identified by having the same as the class it's in. (In this case, the constructor function is `FlashBtn`.) Color values are inserted as arguments, and so you can change them to any color you want for the different button states.

The background is made up of the rectangle and the text field is added to the stage using the `addChild()` function. The order of the two `addChild()` functions is important. The text must be visible on top of the graphic, so it is placed second in the order—at a higher level. Otherwise, the graphic would block the text in the text field. (Switch the order of the `addChild()` functions and you won't see any text when you test the button script using the code in Example 13.)

Example 13simply implements the buttons with unique labels and shows how the buttons can be connected to a mouse event. Open a new ActionScript file and save it as `ShowBtn.as` in the same folder as the `FlashBtn.as` file. Then open a new Flash document file, enter `ShowBtn` in the Document class window in the Properties panel and save it as `ShowBtn.fla` (again, in the same folder as both the `FlashBtn.as` and `ShowBtn.as` files). Finally, in the `ShowBtn.as` file, enter and save the code in Example 13.

Example 13. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;
    import flash.events.MouseEvent;

    public class ShowBtn extends Sprite
    {
        public function ShowBtn():void
        {
            var btnProducts:FlashBtn=new FlashBtn("Products");
            var btnServices:FlashBtn=new FlashBtn("Services");
            var btnContact:FlashBtn=new FlashBtn("Contact Us");

            this.addChild(btnProducts);
            this.addChild(btnServices);
            this.addChild(btnContact);

            btnProducts.x=100;
            btnProducts.y=100;
```

```
            btnServices.x=100;
            btnServices.y=150;

            btnContact.x=100;
            btnContact.y=200;

            //Add Event to a button
            function prods():void
            {
                trace("Products selected");
            }
            btnProducts.addEventListener(MouseEvent.CLICK,prods);
        }
    }
}
```

When you test the application, you'll see three buttons, each with a different label and a width relative to the amount of text in each. When you click the Products button, the Output window shows the message, "Products selected," as shown in Figure 13.

Figure 13. Custom buttons created using ActionScript 3.0

To help you better understand the relationship between the buttons and events, try adding event handlers to the other two buttons. Better yet, change the labels to ones you may be using in your own applications.

Basic Structures

Now that you've had a crash course in display programming, you'll be happy to know that the basic structures in ActionScript 3.0 are very similar to earlier versions of ActionScript. Classes are handled differently, but if you're familiar with the way classes are used in ActionScript 2.0, you won't find too many differences. These structures are reviewed in this section along with a discussion of the different attributes associated with classes, variables and functions. Understanding these relationships is important for working successfully in object-oriented programming (OOP), the topic of the next section.

To get you started, we'll review data types and then classes and their attributes, along with how the different attributes work in different contexts. Let's start by reviewing data types.

Data Types

To successfully work with ActionScript 3.0, you need to use data types extensively and correctly. Data types are declared in ActionScript 3.0 by placing a colon before the data type and after the variable or constant name, as shown here:

```
var total:Number;
const WARNING:String=”Please enter your email address”;
var strSize: uint = mystring.length;
```

With functions, the same rule applies except that the colon follows the parenthesis after a function’s name. The data type refers to the `return` type, if any `return` statement is used. If no `return` statement is used, the data type is `void`. The following illustrates how to use data types with functions:

```
function getNumber() : Number
{
    return total + tax + shipping;
}
function startPlay(): void
{
    trace(“I’m playing”);
}
```

Most data types are capitalized, such as `String`, `Object`, and `Array` as well as class names. However, ActionScript 3.0 has some lowercase data types. Both signed and unsigned integers data types, `int` and `uint`, are lowercase, as is `void`.

> If you’re accustomed to ActionScript 2.0 where `Void` is capitalized, you have to remember this important change; otherwise you’ll get an error message. In ActionScript 3.0, it’s `void` not `Void`.

Making Classes

At this point, you’ve seen several different user classes constructed in the examples, so by now, they’re not a total mystery. However, none of their attributes were discussed in any detail and the process was shown more than explained. So, here we’ll start by going over the basics.

The first step in creating a class is *planning*. An old saying of “Measure twice, cut once,” certainly holds true. That summarizes the whole idea of planning a class. The planning process involves what you want your class to achieve and which parts will make up your class. So what do you plan for? The following list shows what you must consider before you start pounding away on the keyboard to create an application:

Task to achieve

Without knowing exactly what you want your class to achieve, you cannot even begin to plan. So the first thing you want to ask yourself is “What is this class going to accomplish?” It can be anything from creating a button for use in multiple applications, or a very specific task for a single application, such as sending text to a text field inside a movie clip. Write the task down on a piece

of paper. It's also easier to scribble down ideas and re-arrange them until you have your task clearly defined.

Component parts required

Whether you're building a house or baking a cake, you plan for the tools and the ingredients. The tools (hammer, saw, mixer, whisk) are something like the methods and the ingredients (wood, dry wall, flour, baking soda) are analogous to the properties. If you're working on a video application class, you're going to have to think about methods for turning the video on and off, and properties such as placing the video object on the stage and sizing it correctly. So grab a pen or pencil and some note paperWrite down the parts list.

Package and class list

Once you've decided on the parts you're going to need, prepare a list of packages you will need to import. Generally, this begins with the `package` keyword followed by a series of `import` statements that serve to gather up all of your packages and classes. All of this is automatic using older versions of ActionScript with your script in the timeline associated with a keyframe. However, that practice is extremely wasteful and needlessly overloads your code with packages you'll never use. So when you prepare your list, you are not only creating more efficient code and ultimately a better application, you're also organizing your class elements.

Declare your class and extension, if any

For display programming, your class needs to extend the `Sprite` class (at a minimum) for display on the stage. Other classes, including those that may be supporting classes, may not need to extend any other class.

Build your constructor

To implement your class, you need a class constructor, created using a public function with the same name as your class.

Add your methods and properties

These are the tools and parts (or ingredients). Just remember that methods are functions and properties are variables or objects.

These steps and considerations represent everything you'll need to create a simple class, but they also serve as the bedrock of what you will be doing with ActionScript 3.0 and classes. A reference to a "user" class simply distinguishes it from a "built-in" class that is part of a package.

Basic class

To see the basics of how to create a class, let's make a simple one that sends a message to the screen using a text field. As you will see, this example is different from Example 1, in which a script placed a text message on the screen. It is an example of OOP, which is discussed in the next section. The primary feature of the

class built in Example 14 (`SetGet`) is that it is *abstract*. The abstract quality stems from the encapsulated character of the references. For example, in the constructor function (`SetGet`) the `setWord(myword)` function is encapsulated in that it is made up of hidden details. The `setWord()` function is a reference to a private static function in a different part of the program outside of the constructor function. The `myword` parameter is passed from an attribute in the constructor, but none of the details are visible. Also, you will notice that the `private` and `static` attributes are used in the `SetGet` class, which are explained along with Example 14 and Example 15. (Example 15 simply provides content to, and employs, the `SetGet` class.)

Open up two ActionScript files and save one as `SetGet.as` and the other as `DoSetGet.as` in the same folder. Then open a new Flash document file and save it as `DoSetGet.fla` in the same folder as the two ActionScript files. Enter the text and save the script in Example 14 to the `SetGet.as` file.

Example 14. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;
    import flash.text.TextField;

    public class SetGet extends Sprite
    {
        private static var _msg:String;
        public const CONMSG:String="CONSTANT MESSAGE";
        public function SetGet(myword:String):void
        {
            setWord(myword);
            var showStuff:TextField=new TextField();
            showStuff.text=getWord();
            this.addChild(showStuff);
        }

        private static function setWord(saywhat:String):void
        {
            _msg=saywhat;
        }

        private static function getWord():String
        {
            return _msg;
        }
        public function justTrace():void
        {
            trace(CONMSG);
        }
    }
}
```

Type in the script from Example 15 to `DoSetGet.as`. In the Document class window of the `DoSetGet.fla` file, type in `DoSetGet` and save them both in the same folder as `SetGet.as`.

Example 15. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;

    public class DoSetGet extends Sprite
    {
        public function DoSetGet():void
        {
            var showOff:SetGet=new SetGet("Look at this!");
            this.addChild(showOff);
            showOff.x=100;
            showOff.y=100;
            showOff.justTrace();
        }
    }
}
```

By breaking down Example 14 into its different parts, you'll find the key elements of creating a class along with its methods and properties.

1. *Package import.* After the `package` keyword and open curly brace (`{`), which must be included in the class declaration, the packages are imported. This script only uses two classes, so that's all that is imported:

   ```
   import flash.display.Sprite;
   import flash.text.TextField;
   ```

2. *Class declaration*. Here, a `public` class is declared along with the class name. Since the class is an extension of the `Sprite` class, it inherits all `Sprite` class properties and methods.

   ```
   public class SetGet extends Sprite
   {
   ```

   Classes have four different attributes:

   - *internal*: This is the default attribute. If no attribute is specified, the class is created as `internal`. If a class is `internal`, it is visible to reference to the class in the package of the current file. It operates something like a private class.
   - *public*: A `public` declaration means that the class is visible to references everywhere. As you have seen in earlier examples, most of the declarations use the `public` attribute.
   - *dynamic*: A `dynamic` declaration opens up the class for adding properties and methods to the class at run time.

- *final*: Classes with a `final` attribute cannot be extended by another class.

3. *Class constructor*. The class constructor is a function that uses the class name as the function's name. This function runs whenever the class is instantiated. Because it includes a parameter and because it is `public`, it's possible to pass values when invoking the class:

```
public function SetGet(myword:String) :void
{
    setWord(myword);
    var showStuff:TextField=new TextField();
    showStuff.text=getWord();
    this.addChild(showStuff);
}
```

In the same way that classes have attributes, so do class properties and methods. Since the class constructor is actually a function, let's take this opportunity to look at the different attributes associated with the variables, constants, and functions within a class:

- *public*: The function, variable, or constant is visible to references everywhere. You can see this in the constructor function and constant declaration, shown here:

```
public function SetGet(myword:String)….
public const CONMSG:String="CONSTANT MESSAGE";
```

All Caps for CONSTANTS

Another convention you will find both in the built-in and user constants is the use of all caps as labels for constants. In general, you want to avoid all caps in your code because it's more difficult to read. However, using all caps for constant labels is an easy way to differentiate them from variables and it's one of the few exceptions to the "avoid all caps" rule.

- *private*: The function, variable, or constant is visible only to references of the same class. One convention used to identify `private` variables or constants is to begin the property label with an underscore. In Example 14, you can see where the `private` attribute has been used. For instance, the following is a `private` variable:

```
private static var _msg:String;
```

- *static*: The function, variable, or constant is not available in a subclass; they're not inherited. If you don't want a property or method to be inherited by a subclass, use the `static` attribute. For example, the following method isn't available to subclasses of the `SetGet` class:

```
private static function setWord(saywhat:String):void
```

- *protected*: The function, variable, or constant is available only to its own class and subclasses. For instance, the following is a protected variable:

```
protected var exclusive:String="Only for our class and subclasses."
```

- *internal*: This is the default attribute and if no attribute is placed in a function, variable, or constant, it is `internal`. This makes the property or method visible only to its own package. Both of the following examples are identical because `internal` is the default:

```
function haveFun():void
internal function haveFun():void
```

4. *Properties and methods*. Example 14 has a setter/getter pair of `private` static functions, a `private` property, and a single `public` method. The `private` static elements are used in the class constructor function and are executed as soon as the class instance is instantiated. However, the `public` method, `justTrace()` needs to be invoked by attaching it to the instance as shown in Example 15.

```
showOff.justTrace();
```

   You can add as many properties as you want to a class. The planning process, discussed earlier, will help guide you as to how many properties and what type you need.

Decision-Making: Conditional Structures

The conditional structures you'll find in ActionScript 3.0 are like the ones in previous versions of ActionScript. This section simply reviews the basic elements of the different kinds of conditionals you'll find with ActionScript 3.0.

The If condition

The `if` condition sets a single outcome based on whether the Boolean condition is `true` or `false`. For example, suppose you're building a site where you have a single shipping cost for your product. However, if the customer purchases more than $100 worth of products, the shipping is free; otherwise, $12.44 is added to the total. Example 16 shows a class that uses a simple conditional statement to determine whether shipping costs are added. Open a new ActionScript file and copy the script in Example 16 and save the file as `Condition.as`.

Example 16. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    public class Condition
    {
        private var total:Number;
        private const SHIPCOST:Number=12.44;
        public function Condition(amount:Number):void
        {
```

```
            total=amount;
            if(total < 100) {
                total += SHIPCOST;
                trace(total);
            }
        }
    }
}
```

To test the `Condition` class, place the script in Example 17 in an ActionScript file and save it as `ConTester.as` in the same folder as `Condition.as`. (The value 55, placed as a parameter value, is an arbitrary one that's less than 100, which is used to test the conditional.)

Example 17. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;
    public class ConTester extends Sprite
    {
        public function ConTester():void
        {
            var test:Condition=new Condition(55);
        }
    }
}
```

Open a new Flash document file, and in the `Document` class window enter `ConTester` and test the file. Because 55 is less than 100, the shipping costs are added and the output shows 67.44. If you enter 100 or more in the parameter, you won't get an output value.

The if…else and else if conditions

In testing Examples 16 and 17, you saw that if you have values greater than 100, no output is displayed. By adding an `else` clause to Example 16, it's possible to have a response no matter what amount is purchased. Change Example 16 as shown in Example 18.

Example 18. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    public class Condition
    {
        private var total:Number;
        private const SHIPCOST:Number=12.44;
        public function Condition(amount:Number):void
        {
            total=amount;
            if(total < 100) {
                total += SHIPCOST;
                trace(total);
            }
            else
```

```
            {
                trace(total);
            }
        }
    }
}
```

Next, change the parameter value in Example 17 from 55 to 255 and test it. This time, the value in the Output window is 255. That's because no shipping costs were added. The conditional statement determined the value to be no less than 100, so the script turns to the `else` clause to output the total without adding the shipping costs.

In situations where you have more than one condition you want to test, you can use the `else if` statement. For example, suppose you create a video player application and the user chooses from different FLV files to play. Depending on which video the user selects, different choices need to be made. Example 19 shows a class with multiple conditions. Enter the code in an ActionScript file and save it as `MulCondition.as`.

Example 19. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    public class MulCondition
    {
        private var flvnow:String;
        public function MulCondition(flv:String):void
        {
            flvnow=flv;
            if (flvnow == "One")
            {
                trace("Play One.flv");
            } else if (flvnow == "Two")
            {
                trace("Play Two.flv");
            } else if (flvnow == "Three")
            {
                trace("Play Three.flv");
            } else
            {
                trace("Choice not recognized");
            }
        }
    }
}
```

Note that the first conditional is a simple `if` statement and that the last `else` clause does not contain the `else if` statement. Basically, if the first condition isn't met, it brings in different conditions to test for. If none are recognized, the last `else` clause executes. Example 20 shows an implementation in use. Save the code as `MulConTester.as` and then open a new Flash document and save it as `MulConTester.fla` in the same folder as `MulCondition.as`. Next, type in

`MulConTester` in the Document class window of the `MulConTester.fla` and test the file.

Example 20. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;
    public class MulConTester extends Sprite
    {
        public function MulConTester():void
        {
            var test:MulCondition=new MulCondition("Three");
        }
    }
}
```

When you test the file, you'll see that the Output window displays, "Play Three.flv." Try entering the name, "Gone with the Silicon" in the FLV parameter and see what happens. You will see that any entry other than "One", "Two", or "Three" will result in the "Choice Not Recognized".

Switch/Case structure

The final conditional structure is the `switch`/`case` statement. As an alternative to using the `else if` condition, the `switch`/`case` statement is favored when several different conditions must be handled, and in cases where you need to add more conditions. Using the same example as was employed for the `else if` statement, you can see the relative simplicity of the `switch`/`case` structure. Examples 21 and 22 provide the script for creating a `switch`/`case` application. Save both examples in the same folder, with Example 21 as `SwitchCase.as` and Example 22 as `SwitchCaseTester.as`. Then in a Flash Document class window, enter `SwitchCaseTester` and test the application.

Example 21. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    public class SwitchCase
    {
        private var flvnow:String;
        public function SwitchCase(flv:String):void
        {
            flvnow=flv;
            switch (flvnow)
            {
            case "One" :
                trace("Play One.flv");
                break;

            case "Two" :
                trace("Play Two.flv");
                break;
```

```
            case "Three" :
                trace("Play Three.flv");
                break;

            default :
                trace("Choice not recognized");
            }
        }
    }
}
```

Example 22. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;
    public class SwitchCaseTester extends Sprite
    {
        public function SwitchCaseTester():void
        {
            var test:SwitchCase=new SwitchCase("Two");
        }
    }
}
```

When applying the `SwitchCaseTester` class in Example 22, you can see that other than changing the name of the class reference and its selected string (“Two’ instead of “”), this example is identical to the script shown in Example 20.

Loops

Like the conditional statements, loop syntaxes are similar to earlier versions of ActionScript. Generally, loops are the workhorses for repeating tasks. For example, loops are used for parsing arrays and other objects where either a known or unknown number of elements must be sought.

To look at the format and structure of the different loops, the following examples are named for the loop type. The examples are set up in pairs or as standalone examples that can be run by entering the class name in the Document class window of a Flash document. To make the process simple, create a single Flash document and save it as `Loop.fla`. Then enter the test class name for each of the examples and test them. Save all of the following loop examples in the same folder as the `Loop.fla` file, using the class name as the filename.

The for loop

The `for` loop is used to iterate through a known and finite number of repeated operations. The operations for the loop begin at the opening curly brace and end before the closing curly brace. The general format consists of a counter variable, an end condition, and an increment/decrement variable separated by semi-colons as shown in the following pseudocode:

```
for(counter=startValue; terminate condition; increment) {
```

The following example, made up of two files, uses the length of an array as an end condition:

for loop (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    public class ForLoop
    {
        private var loopsize:uint;
        private var counter:uint;
        public function ForLoop(gang:Array):void
        {
            loopsize=gang.length;
            for (counter=0; counter < loopsize; counter++)
            {
                trace(gang[counter]);
            }
        }
    }
}
```

for loop test (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;
    public class ForLoopTester extends Sprite
    {
        private var guys:Array=new Array("Larry","Mo","Curly");
        public function ForLoopTester():void
        {
            var test:ForLoop=new ForLoop(guys);
        }
    }
}
```

The for...in loop

The `for...in` loop is used to iterate through objects, including arrays. The structure is made up of a variable to store the elements in a named object:

```
for(storeVar in someObject) {…
```

The loop is zero-based, in that the first element is the zero-property or zero-element of the object. The following example iterates through an array. Note how the abstract structure in the initial class is implemented concretely in the test class.

for...in (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    public class ForInLoop
    {
        public function ForInLoop(prop:Object):void
        {
            for (var p:Object in prop)
            {
                trace((Number(p)+1) +". "+ prop[p]);
```

```
            }
        }
    }
}
```

for..in test (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;
    public class ForInLoopTester extends Sprite
    {
        private var mypack:Array=new Array("Delia","WillDe","Bill");
        public function ForInLoopTester():void
        {
            var test:ForInLoop=new ForInLoop(mypack);
        }
    }
}
```

The for each...in loop

In addition to what the `for...in` loop does, the for `each...in` loop iterates through collections, including property values and `XML` and `XMLList` object properties. It uses the following format:

```
for each(someObject in otherObject) {...
```

The following example pair is set up to use any kind of object, illustrating its actual use with an array.

for each in (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    public class ForEachInLoop
    {
        public function ForEachInLoop(group:Object) :void
        {
            for each (var thingy:Object in group)
            {
                trace(thingy);
            }
        }
    }
}
```

for each in test (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;
    public class ForEachInLoopTester extends Sprite
    {
        private var internet:Array=new Array("PHP","Perl","C#","ColdFusion");

        public function ForEachInLoopTester():void
        {
            var test:ForEachInLoop=new ForEachInLoop(internet);
        }
```

```
    }
}
```

The while loop

The `while` loop has a test condition at the beginning of the structure followed by the operations after the first curly brace. It uses the following format:

```
while(condition) {…
```

If the condition is `false` in the first iteration, no operation is run. This kind of loop is good in situations where you don't want anything to happen if the loop condition is `false` at the outset. In the following example, you can directly enter the class name in the Document class window without having to create a test class:

while loop (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;
    public class WhileLoop extends Sprite
    {
        private static var counter:Number=10;
        public function WhileLoop():void
        {
            while (counter > 0)
            {
                trace(counter);
                counter--;
            }
            trace("Blast off!");
        }
    }
}
```

The do...while loop

The key difference between the `while` and `do...while` loop is that the test condition comes after the first operation. So, even if the test condition is initially `false`, the `do...while` loop executes at least one operation. It has the following format:

```
do { operation } while(testCondition)
```

The next example can be tested by entering the class name in the Document class window of the test Flash document:

do...while loop (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;

    public class DoWhile extends Sprite
    {
        private var parseme:String="Albert Einstein";
        private var counter:uint=parseme.length;
```

```
        private const WHOLE:uint=parseme.length;
        private var tempstr:String;
        private var albert:String="";

        public function DoWhile():void
        {
            do
            {
                tempstr=parseme.charAt(WHOLE-counter);
                albert+= tempstr;
                trace(tempstr);
                counter--;
            } while (tempstr != " ");
            trace(albert);
        }
    }
}
```

Object-Oriented Programming

One of my favorite books on object-oriented programming (OOP) books is Alexander Nakhimovsky's and Tom Myers' *JavaScript Objects: Object Use and Data Manipulation with JavaScript* (Wrox, 1998). I like it because the authors are smart, and they're not trying to prove how smart they are because they know OOP. What's more, they show how OOP can be applied to JavaScript, a weakly typed, interpreted language. Coming from a couple of Colgate University computer science professors accustomed to programming in languages like C++ and Java, this perspective and stance caught my attention more than the same such issued by someone who just happens to like JavaScript. The clear implication is that you don't need an OOP language to apply OOP principles. In fact, OOP is as much an approach to programming as it is the built-in structures of a language.

Another point made by Nakhimovsky and Myers was that in the future (post-1998), more non-programmers would be writing programs. In other words, more people without a computer science or computer engineering degree would be hacking code—literally. The prediction back in 1998 accurately portrays most ActionScript programmers; starting with just a few statements, operators, and commands, ActionScript has grown to a full-fledged OOP language in ActionScript 3.0. Still, though, most who write ActionScript programs aren't programmers with CS degrees. Nakhimovsky and Myers were not lamenting the fact that non-programmers would be programming, but rather pointing out that if you're going to program, you can do a better job and enjoy it more by using the principles of OOP.

Why OOP?

If you root around in the origins of OOP, you will find that, among other reasons, OOP was developed as an approach to programming because it provides *real-world analogies*. That is, the developers (those who write code) of OOP wanted to make programming easier. For those who think of OOP as convoluted and a more difficult way of programming, consider the fact that OOP's creators were running into difficulties that were extremely hard to solve using *procedural* programming methods. (Procedural programming is a method of writing one procedure after another in a linear manner.) OOP's developers came up with the idea of *objects* (or the object analogy), which have different properties and methods (characteristics and actions). For example, I have a dog (an object) who has a personality (a property), a tail (another property) and he barks (an action) at the various critters that pass through our yard in the middle of the night. All of his doggy parts work in concert to produce the appearance and behavior he exhibits.

By the same token, I have a Flash application that I use for playing (method) selected videos (property). It also records (method), appends (method), and pauses (method) the video (property). It has a text window (object) where the user types in what she wants to see and hear (property). Instructions (property) tell the user what information she has to enter.

In addition to making programming easier, OOP was developed to handle more complex programs, especially those with interaction between the program and other elements, such as users, other program elements, and data. Internally, OOP's organization is easier for the compiler to handle and the program runs more efficiently.

So the answer to the question of "Why OOP?" is that your programming gets easier as you develop more complex structures, and your application will run more efficiently and effectively. Thus, you will enjoy programming more. What's not to like? Figure 14 shows a simplified OOP structure.

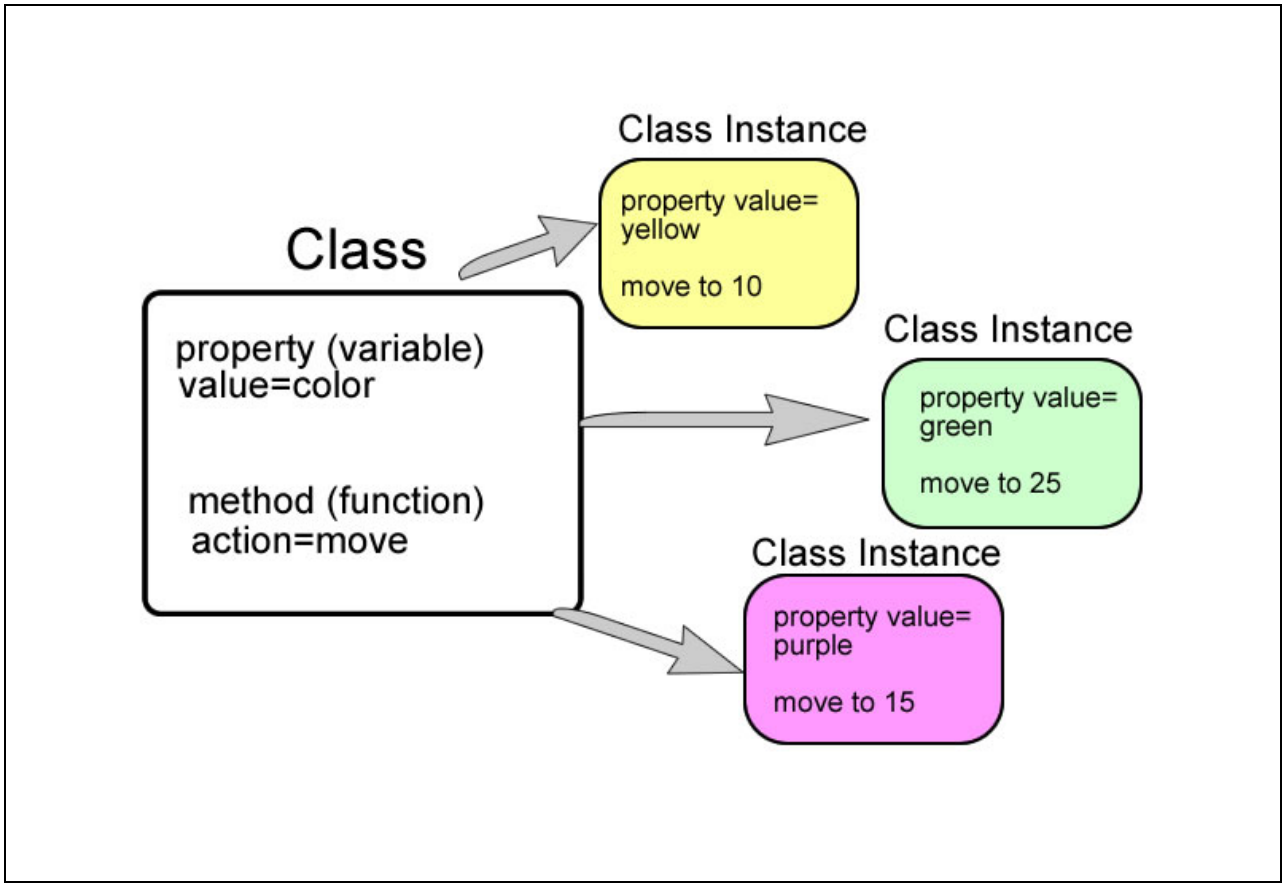

Figure 14. Classes are Abstract Representations of Objects (Instances)

ActionScript 3.0 and OOP

Probably the most important features of ActionScript and OOP are not visible. ActionScript 2.0 (and earlier versions) could be organized and executed using OOP principles; however, the actual compilation and execution of the code did not take advantage of OOP structures. ActionScript 3.0 does.

OOP Fundamentals

This next section steps through the basic elements of object-oriented programming. Everything is quite basic and is meant as a starting point for further exploration. The examples highlight how ActionScript 3.0 relates to these different OOP elements. For OOP novices, this start is meant as an invitation to explore object-oriented programming and become at least a nodding acquaintance. Experienced OOP developers will get a glimpse of how ActionScript 3.0 is similar to and different from other OOP languages they may have used previously.

Class organization

Whether you're seasoned at OOP or a relative ActionScript newbie, chances are that you've worked with classes, even if you didn't realize it. Classes in ActionScript are abstract diagrams of a set of properties and methods. In this document, you've seen several examples of both built-in and user-generated classes and their properties and methods. For example, the following lines display an instance of the `String` class (`Pedro`), a property of the `String` class (`length`), and a `String` class method (`slice`).

```
var Pedro:String="Vote for Pedro";
var perimetro:uint = Pedro.length;
var cordonnuevo:String = Pedro.slice(0,perimetro-10);
trace(cordonnuevo);
```

To create your own OOP class, you'll need an abstract plan for what you want the class to do. For instance, you might want a plan for how to generate messages entered by users. To hold that message, you'll need a variable, which is one of the classe's *properties*. To deliver the message, you'll need a function, which is one of the classe's *methods*. Like the property, the method is abstract. It doesn't place the message into a text field or pass it to another variable, it simply returns the message. Example 23 creates a class that satisfies the plan's criteria to create and deliver messages. It includes commented elements of a complete OOP class:

Example 23. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    public class Sample
    {
        // This is a property
        private var goodProperty:String;
        // This is a constructor method
        public function Sample():void
        {
        }
        // This is a method
        public function goodShow():String
        {
            return goodProperty;
        }
        // This is a method
        public function setProp(msg:String):void
        {
            goodProperty=msg;
        }
    }
}
```

The `Sample` class is an abstraction of a simple plan to return a text string. `Sample` has a single string property (`goodProperty`), a constructor method, a method to set the property value (`setProp`), and a method to return the string property (`goodShow`). Remember, the constructor method has the same name as

the class and is used to instantiate the class. In this example, the constructor class doesn't have any content, so nothing happens when an instance (object) of the class is created.

To create an instance of the abstract class, another class (often named `Main`), is used to assign content for the abstract class and display the contents assigned to the properties, as shown in Example 24.

Example 24. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;
    import flash.text.TextField;
    public class TestSample extends Sprite
    {
        public function TestSample() {
            var textUp:TextField=new TextField();
            var test:Sample=new Sample();
            test.setProp("Hi Class");
            textUp.text=test.goodShow();
            this.addChild(textUp);
            textUp.x=100;
            textUp.y=100;
        }
    }
}
```

As you can see, the first aspect of writing a good OOP program is to use both built-in and user-created classes that contain properties and methods that map out what your plan needs.

They're All Properties

When reading books and articles about OOP, you will find that the word *properties* often refers to both properties and methods. While properties are variables, and methods are functions, grouping them as "properties" is a bit less awkward when discussing objects in general. After all, an object is made up of parts, which is the important feature, and to refer to both content features (properties) and action features (methods) in the same vein doesn't really distract from that fact.

Inheritance

In the context of OOP, *inheritance* refers to a class's ability to pass on its properties and methods to another class. When one class extends another class (called *subclassing*), the subclass inherits all of the class's non-private properties. Example 25 extends the `Sample` class created in Example 23. In Example 23, the `Sample` class contained one property, `goodProperty`, and two methods, `setProp` and `goodShow`. By extending the `Sample` class to `ExtendedSample` (Example 25), the `ExtendedSample` *inherits* `setProp`,

`goodProperty` and `goodShow`. It can also add more methods and properties of its own. Example 25 adds a numeric property and a function that returns the numeric property.

Example 25. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    public class ExtendSample extends Sample
    {
        //This is an added property
        public var numProperty:Number;
        //This is a constructor method
        public function ExtendSample():void
        {
        }
        //This is an added method
        public function goodNum():Number
        {
            return numProperty;
        }
    }
}
```

To see inheritance at work, Example 26 instantiates the `ExtendSample` class. Both methods and the property inherited from `Sample` and those created in the `ExtendSample` class are assigned values and targets and are output on-screen.

Example 26. (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;
    import flash.text.TextField;
    public class TestExtendedSample extends Sprite
    {
        public function TestExtendedSample()
        {
            var textUp:TextField=new TextField();
            textUp.width=150;
            textUp.height=15;
            var test:ExtendSample=new ExtendSample();
            test.goodProperty="Half price = only $";
            test.numProperty=42;
            test.numProperty/= 2;
            textUp.text=test.goodShow() + test.goodNum();
            this.addChild(textUp);
            textUp.x=100;
            textUp.y=100;
        }
    }
}
```

As with inheritance in family trees, you'll find more complexities than meet the eye. However, the main point here is that characteristics can be inherited from one class to another.

Abstraction and encapsulation

While different concepts, discussing abstraction and encapsulation together, helps to understand both.

Abstraction refers to defining characteristics of an object without specifying the contents of that object. Example 23 is a class that has abstract characteristics in the form of the `goodProperty` property, the `setProp`, and `goodShow` methods:

- All we know about `goodProperty` is that it's a string, but we don't know the content of that string.
- Likewise, we know that `goodShow` returns a string, but how and where that string goes is up to its implementation.
- The `setProp` method allows some string to be set, but it leaves that up to the implementation.

Imagine an abstraction in the same way that a window is an abstraction. We all understand the abstract concept of a window, and we also know that a window can be anything from a large plate glass window in a department store to a stained glass window in a cathedral; even a porthole on a ship. Likewise, we can use the abstract term *dog* to refer to implementations that range from a Chihuahua to a Greater Swiss Mountain Dog. However, the abstraction is clear enough that we can differentiate one abstraction from another. For example, our abstraction of *window* should differentiate it from the abstraction for a *door*. Likewise, our abstraction of *dog* should be different from an abstraction of a *cat*. So while abstractions don't provide the particular content, they are specific enough to distinguish one abstraction from another.

To appreciate *encapsulation*, let's go back to why OOP was developed in the first place. Using procedural methods often caused one property and its values to get tangled up with others. As programs became larger, this problem increased exponentially. By encapsulating properties, changes can be made to that property without accidentally making changes where you don't want them to occur. Both abstraction and encapsulation are techniques used to take a big problem and break it down into manageable parts. At the same time, they afford greater flexibility to your code so that if you want to expand or change your program, you can do so without having to start all over again. What's more, you can reuse the parts (classes) in wholly different programs. For example, once you program an extension of the `SimpleButton` class, you don't want to rewrite the code every time you create a new application that needs a button. Thanks to abstraction, however, you can implement the button with the specific characteristics you want for your new application. At the same time, the encapsulated elements help to keep

the new implementation clean and clear by not allowing changes in the properties to cause unwanted changes in the new implementation.

Getters/Accessors

One way to enforce encapsulation is to use *getters* (also called *accessors*). A getter is a method that gets information about an object (it provides access to that information). Getters can be restricted by using private methods. If a method is private, it can only be accessed by the class itself, so there's no way for an outside object to get to it. In Example 23, the getter method is `goodShow`, but it is a `public` method, so it isn't as restrictive. However, if you look at Example 14, you'll see that the `getWord` method is `private` and is only implemented within the class. As a result, the only way to get to the property in Example 14 is through the class constructor. That isn't very flexible or helpful. (Example 14's purpose is to illustrate a class and its internal processes.)

Generally, though, getters are `public` methods, which allow changes to be made, as shown in Example 24. In the implementation of a class, `private` getters certainly enforce encapsulation, but they don't allow the flexibility often needed with getters.

Setters/Mutators

Besides getting information about an object's properties, a property's values often have to be changed. To change an object's information while enforcing encapsulation, *setters* (also called *mutators*) are used. Public methods that do nothing more than assign values are setters, as with the `setProp` method shown in Example 23. When the class is implemented in Example 24, you can see how the `setProp` method is used to assign a concrete value. Note, though, that it assigns a value to a `private` property encapsulated in the class definition.

Getters and Setters with `get` and `set`

You can also use the ActionScript 3.0 `get` and `set` statements to create your getters and setters. The methods resulting from using these statements allow you to call information very much like a property—without using parentheses. For example, if you establish your getters and setters as:

```
public function get flv():String {
          return flvTitle;
}
public function set flv(fileName:String):void {
      flvTitle= fileName;
}
```

You would access them using the format:

```
someInstance.flv= "flyByWire .flv " //Setter
trace(someInstance.flv); //This is how the getter works
```

As you can see, they look more like properties than methods, but they are actually methods. The `get` and `set` statements were not used in the examples because they may have been mistaken for properties. However, they are methods, and once you get used to their format, you may prefer their use in creating your getters and setters.

Polymorphism

In the most general sense, *polymorphism* refers to the different implementations of an abstract class. For example, consider again the concept of a window as an abstraction. You could subclass `Windows` to department store window or cathedral window. Both classes implement methods from the `Windows` class, but they do it in different ways. Instead of having a single (mono) implementation, you have several (poly) implementations (or forms) that the initial concept of window can take. The following pseudo-code illustrates this behavior:

```
public class Windows {
    public function Windows() {}
    public function putInFrame() {
    // Do frame stuff }
    public function putInGlass()
    // Do glass stuff }
}
```

By subclassing the `Windows` class into `DepartmentStore` and `Cathedral` classes, the same function names are inherited from the `Window` class, but they will have different functionality. In the `DepartmentStore` implementation, the `putInFrame` method will have one big window frame to put in; the department store and the `Cathedral` implementation will have lots of little frames for the stained glass windows. Likewise, the `putInGlass` method is different in both as

well. One implementation has one big piece of clear glass, and the other has lots of little pieces of colored glass.

The concept of polymorphism is an important one for object-oriented programming, and it is one best appreciated when you run into it in one of your own applications. Personally, I found the concept clearer when using interfaces, which is the next topic of discussion.

Interfaces

An *interface* is something like an abstract class. Basically, an interface (not to be confused with a graphical user interface, which is something altogether different) is a set of methods differentiated by name and data type, but which don't contain anything. Interfaces are often likened to contracts, where everyone is in agreement that they'll stick with the terms set up in an interface. The terms of the "contracts" are an agreement that the same data types and arguments in the interface will be used in the implementation of the interface.

Keeping with the window analogy, two methods were used to build a window: `putInFrame` and `putInGlass`. Here's what an ActionScript 3.0 interface would look like:

Interface (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    interface Windows
    {
        function putInFrame():void;
        function putInGlass():void;
    }
}
```

Basically, the contract says, "If you're going to implement this interface, use these two functions with no returns (`void`)." This is a new level of abstraction, but with this, you can create windows for anything from a shed to the Empire State Building.

To use the interface, you need to implement it. You'll need two interfaces: one for the department store and another for the cathedral, as shown here:

Department Store (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    class DepartmentStore implements Windows
    {
        public function DepartmentStore()
        {
            trace("==Department Store==");
        }
        public function putInFrame():void
        {
```

```
            trace("Put in one big frame.");
        }
        public function putInGlass():void
        {
            trace("Put in one big clear piece of glass.");
        }
    }
}
```

The constructor function simply adds a message to the Output window using a `trace()` statement so you can tell it's the department store implementation. Then, both of the methods in the interfaces are written with `trace()` statements to indicate a unique implementation of the method. Compare that with the cathedral implementation, shown here:

Cathedral (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    class Cathedral implements Windows
    {
        public function Cathedral()
        {
            trace("==Cathedral==");
        }
        public function putInFrame():void
        {
            trace("Put in lots of little frames made of leaded glass.");
        }
        public function putInGlass():void
        {
            trace("Put in lots of different colored glass.");
        }
    }
}
```

Following the interface "contract" analogy, the cathedral implementation stuck with the method names and return structure, but put in wholly different `trace()` statements. (Instead of `trace()` statements, any other type of statements, functions, or anything else consistent with the interface would be acceptable as well.)

To compare the two different implementations of the `Windows` interface, a test class instantiates both implementations in juxtaposition. This way, you can see how the same methods were implemented differently.

Test Windows (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;
    public class DoWindows extends Sprite
    {
        var myDepartmentStore:Windows;
        var myCathedral:Windows;
        public function DoWindows()
        {
```

```
            myDepartmentStore=new DepartmentStore();
            myDepartmentStore.putInFrame();
            myDepartmentStore.putInGlass();

            myCathedral=new Cathedral();
            myCathedral.putInFrame();
            myCathedral.putInGlass();
        }
    }
}
```

The output window shows the following:

```
==Department Store==
Put in one big frame.
Put in one big clear piece of glass.
==Cathedral==
Put in lots of little frames made of lead.
Put in lots of different colored glass.
```

As the output shows, the same methods result in entirely different messages. However, it's clear that the messages are consistent with their different contexts. This is polymorphism at work.

Design Patterns

As a final item in this document, I'd like to introduce *design patterns*. Since ActionScript 3.0 is a true object-oriented programming language, you might as well go to the next level of OOP and become familiar with what most professional programmers already know and use (or would like to know and use).

Design patterns were developed by and for professional programmers as a set of coding practices to optimize the quality of OO programs. They constitute a framework for solving recurring software development problems. Developed and tested over years of work by programmers, they were codified and presented as a set of programming guidelines in a 1995 work entitled, *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. (Better known as "The Gang of Four"—or just GoF—the coauthors' epithet has become synonymous with design patterns.)

For those just learning about OOP, design patterns serve as a model for getting OOP right, albeit at a relatively advanced level. And now that ActionScript 3.0 is a serious OOP language, any effort to acquire OOP skills should also include a set of best practices. In this short introduction, we'll take a look at a simple design pattern called a Singleton and explain its purpose.

The DP quick and dirty primer

Like OOP itself, design patterns offer a strategy for approaching recurring issues that all programmers face. If you think of design patterns as clothing patterns, you

can better understand how to approach both design patterns and the idea of programming architecture.

Imagine that you're a clothing manufacturer, and you have a set of different patterns for different types of clothing. A summer collection that includes patterns for shorts, short-sleeved shirts, light materials, swimwear, and bright summer colors. The same is true for fall, winter, and spring clothing patterns. Likewise, you will likely have patterns for informal and formal occasions as well as patterns for everything from outdoor camping to school clothes for children.

With each seasonal change, you are faced with the recurring challenge of coming up with clothes for the next season. Instead of treating each season anew, you pull out patterns for that season. Fall season calls for more woolen patterns with long sleeves, made for layered wear. Winter brings out heavier materials and designs that will keep your customers warm in frigid temperatures.

It's not rocket science to determine that once you've made a pattern for a certain season, you might as well reuse it when the same season rolls around again the following year. You need flexibility because styles change, but otherwise, you know that summer patterns work best in the summer and winter patterns work best in the winter.

The same is true with software design patterns. If you've spent time working out a solid OOP solution to a programming problem, you can reuse it again and again when the same problem rolls around.

The important feature of design patterns is that they address specific problems, yet are flexible enough to deal with a wide variety of similar programming issues. At the same time, the patterns are designed to optimize every aspect of good practices in OOP. So, while they are shortcuts to solving complex, recurring problems, they also represent good programming practices.

A design pattern example: The Singleton

In working with ActionScript 3.0, you will be working with classes and objects implemented from a class. Sometimes you need to be sure that there's only a single instance of a class and yet global access to the class. For example, if you create an application that plays MP3 files, you want to ensure that only a single instance of the class that plays the MP3 files is going to be instantiated at a time. Otherwise, multiple instances could lead to multiple MP3 files playing at the same time. Also, you want a single global access to the player so that you can select what you want and be sure that only one tune at a time is played.

The following Singleton example is an abstract one, but it serves to illustrate a Singleton structure, allowing only one instance to be instantiated at a time. It more or less follows a "classic" Singleton design in the use of a `private` constructor

function and a test of previous instantiation. The only problem is that in ActionScript 3.0, constructor functions have to be `public`. To get around this, you can use an `internal` class created outside of the class, or even `package` definition.

In the class constructor, if you use the `internal` class object as an argument, it serves to work like a `private` constructor function. A higher level of security is achieved by keeping the `internal` class outside both the `class` and `package` definition, but you can place the `internal` class outside the `class` definition and still keep it inside the `package` definition.

In the Singleton example, the `internal` class at the bottom of the script is outside the package, so it isn't visible outside the source file. (Also note that the `internal` attribute is missing in the function `PrivateClass`. That's because the `internal` attribute is the default and is automatically placed there.) By using the `private` class instance in the constructor, there's no way that a Singleton can be instantiated outside of the class. Only by using the `getInstance()` accessor can an instance be instantiated.

The logic of the Singleton code is both simple and profound. The constructor function opens with a `getInstance()` function that checks to see if a Singleton instance exists. If no such instance is currently instantiated, it creates one; otherwise it falls through and leaves a message that an extant Singleton already exists. The script then falls through to the `return` statement, which returns either the new Singleton instance it just created or one that already exists. In either case, only a single instance of the Singleton class is allowed. Type in the following example in an ActionScript file and save it as `Singleton.as`.

Singleton Example (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    public class Singleton
    {
        private static var _instance:Singleton;
        public function Singleton(pvt:PrivateClass) {
        }
        public static function getInstance():Singleton
        {
            if(Singleton._instance == null)
            {
                Singleton._instance=new Singleton(new PrivateClass());
                trace("Singleton instantiated");
            }
            else
            {
                trace("Sorry, we already have a Singleton instantiated")
            }
            return Singleton._instance;
```

```
        }
    }
}
class PrivateClass
{
    public function PrivateClass() {
        trace("Private class is up");
    }
}
```

To test the Singleton, you'll need a test that attempts to instantiate more than a single instance of the class. Open a new ActionScript file and type in the Test Singleton script and save the file as `SingletonTest.as` in the same folder as the `Singleton.as` file.

Test Singleton (download this code sample at http://examples.oreilly.com/actionscript3qr)

```
package
{
    import flash.display.Sprite;
    public class SingletonTest extends Sprite
    {
        public function SingletonTest()
        {
            var initialSingleton:Singleton = Singleton.getInstance();
            var subsequentSingleton:Singleton = Singleton.getInstance();
        }
    }
}
```

Finally, open a new Flash document and save it as `SingletonTest.fla` in the same folder as the `Singleton.as` folder. In the Document class window, type in `SingletonTest` and test the script. You should see the following output:

```
Private class is up
Singleton instantiated
Sorry, we already have a Singleton instantiated
```

The pseudo-private class has been included as evidenced by the first line of output. This indicates that the trick to instantiate the Singleton as a `private` class worked. Then, the first attempt to create a Singleton instance works fine as well. However, a second attempt to do the same thing indicates that it was unsuccessful. That's exactly what a Singleton is supposed to do!

The point of all of this is not that it's difficult or even especially brainy to use design patterns. Rather, you can use the architecture in the design patterns to create good OOP. Like good algorithms that have been used over and over since the dawn of computer programming, good design patterns provide effective models for programming.

To learn more about design patterns, besides the original canon by the Gang of Four, you can't find a better book on design patterns than the highly readable *Head First Design Patterns* by Eric and Elisabeth Freeman (O'Reilly, 2004). Even though all of the examples are written in Java, going through it will give you a

thorough grounding in design patterns by some very smart people. Also, the forthcoming book *ActionScript 3.0 Design Patterns* by William B. Sanders and Chandima Cumaranatunge (O'Reilly, 2007) is designed to help you understand object-oriented programming through design patterns.

Next Step for Object-Oriented Programming

At its heart, OOP makes a lot of sense, and conceptually, it's very clear. However, as with everything else, the devil is in the details. With short programs, knocked out for a particular task of no great shakes, it's easy to ignore OOP and just pound out what you need. Also, to do OOP right, you need to plan. I don't mean to just think up a few data types you might need, but to sit down and think about your project—even if it's just a short one. And, yes, learning OOP does take some effort, and you will lose your old sequential programming habits in the process, and that's a good thing. However, the rewards are the joy of solving computing problems at a whole new level, becoming a better programmer, and in the satisfaction of doing a job well.

To get started with OOP, get a good book that covers OOP. Ideally, get one that's written for ActionScript 3.0; even one written for an earlier version of ActionScript is encouraged if you can't find a current one. *Essential ActionScript 3.0* by Colin Moock and *ActionScript 3.0 Cookbook* by Joey Lott, Darron Schall, and Keith Peters are both good choices for seeing how to create object-oriented programs in ActionScript 3.0. If you have an OOP book written for another language, such as Java, C++, JavaScript, or C#, that can be helpful as well. In any case, you need to look at something more than was provided here. Take this as a starting point, because now more than ever, you have a version of ActionScript that can truly create OOP structures and take advantage of OOP and design pattern programming.