APPENDIX C

ActionScript Primer

An interesting dilemma in this book is that, in parts, we had to use ActionScript but the book was not able to fully cover ActionScript. This appendix is a brief primer that can help shed some light on the concepts of ActionScript.

In Hour 15, "Basic Interactivity" you were introduced to placing ActionScript into the Actions panel. You learned that not only does Flash interpret and execute each line of code sequentially, but when code is placed in later frames, the code is not executed until that frame is reached. In addition, you saw how to use `addEventListener()` to broadcast events from one object (say the `CLICK` event from a button) to trigger code inside a custom function.

But the text was also peppered with terms such as "variables" "instances" and "properties." The rest of this appendix, in sort of a narrative form, describes the core features in ActionScript. I don't expect this quick lesson will turn you into a hardcore programmer, but at least you'll be familiar with the terms that appear in Hours 15–17. To explain what's going on in those hours I had to use a programmer's vocabulary.

It's easiest to start with the most common object type, Movie Clips, because they have a visual representation onstage. (By the way, a components is an extended type of Movie Clip, meaning that components are everything Movie Clips are, plus they include some additional features.) A Movie Clip symbol onstage is called an instance. You can give each instance on stage a unique instance name (using the Properties panel). The instance name is used in your ActionScript to refer to the clip—or, more technically, to address the clip. The reason you don't use the symbol name (from the master Movie Clip symbol in the Library) is because you might have multiple instances of that same symbol onstage and you want to address each one individually. Why would you want to address a clip? Usually to change one of its properties such as its position onstage (either its `x` property or `y` property, although there are many others). Movie Clip instances are the easiest type of object to understand because you can see them. But there are also instances of Sound and Date—just to name two. Think of this analogy: People have properties (like hair color, weight, and age) and cars have properties (horsepower, make, and model). Interestingly, sometimes two different object types share the same property. Cars and humans both have a weight property. Movie Clip instances and Button instances both have a `width` property. Often, however, the available properties depend on the type of object. Sound instances have a `duration` property but Movie Clip instances don't (though Movie Clips do have a `totalFrames` property, which is as close to a "duration" as you will find).

The good news is that the code you write to address clips and their properties uses the same form (or syntax) in every case. Namely, the syntax to refer to an instance's property is always *`myInstance.myProperty`* (or "object dot property"). (I use the prefix "my" to indicate something I made up so that you don't think the terms *`myInstance`* or *`myProperty`* are built into ActionScript.) Note that sometimes you want to set a property (perhaps set a clip instance's `rotation` property to make it spin) and other times you'll just need to get a property's value to use within a larger statement. (A statement is a complete instruction—basically one line of code.) Check out this example:

```
myClip.x = otherClip.x
```

When Flash encounters this line of code it interprets the code and executes the instructions (meaning it does what the code says to). In this example, the instance called `myClip` gets its `x` property set to a value equal to the `x` property of another instance called `otherClip`. (Any time you see a single equals sign it's an assignment meaning "is now equal to" as in "myClip's x is now equal to otherClip's x".) Notice that you're setting the `x` property of `myClip` but only getting the `x` property of `otherClip`.

Another important concept is methods. Methods are like processes or procedures applied to a single instance. Back to the human analogy: walk, talk, and comb your hair are all methods. Methods that you can apply to Movie Clip instances include `play()` `stop()`, and `gotoAndPlay()`. I like to compare properties to methods because their syntax is nearly identical. It's always "object dot method" as in `myClip.play()`. Methods always have parentheses. Some methods accept parameters (also called arguments) that provide needed additional details. For example, when you say `myClip.gotoAndPlay(1)`, `1` is a parameter indicating which frame you want to "go to."

Events are things that happen while a Flash movie plays. The most intuitive event types are things the user does: clicks, drags, or presses a key. Naturally there are all kinds of events, and like properties and methods they vary depending on the object type. For example, Sound instances broadcast (or fire) an event for `COMPLETE` (when sound ends). A Button has a `CLICK` event (but no `COMPLETE` event). Just as trees fall in the woods without anyone to hear them, events fire in Flash and—unless your code is listening for that event—they go unnoticed. You have to write code to listen for specific events and define exactly how you want to handle that event. Luckily in ActionScript 3.0 all event handling follows the same convention. You use the `addEventListener()` method on the object for which you want to trap events, and you specify both which event you're listening for and where that even should be handled (specifically, which homemade function should trigger). A complete block of code to handle a button's `CLICK` would look like this:

```
myButton.addEventListener ( MouseEvent.CLICK, myHandler )
function myHandler (evt ){
   trace ("you clicked")
}
```

I guess I lied because buttons don't really have a CLICK but it's a lowercase string: `"click"`. The thing is that `MouseEvent.CLICK` has a value of `"click"`. Either one will work (that is, the first parameter in the `addEventListener()` method should be the string name for the event you're trapping). However, `MouseEvent.CLICK` is safer because if you accidentally spell anything wrong Flash warns you when you test the movie.

Another core concept is using custom variables. You can create variables as a way to store data for later use. For example, you could assign a variable's value with this code: `var myName = "Phillip"` (which translates to "the variable named `myName` is now equal to the string `"Phillip"`"). (The first time you refer to a variable you need to precede it with `"var"`.) You can change the value of a variable by simply reassigning a value, but at any one instant a variable has a single value. You can store any data type in your variable, but you'll often want to type your variables as a particular data type (meaning you define the data type they are allowed to contain). The way you type a variable is by saying

```
var myName:String = "Phillip"
```

The reason to type a variable is simply so Flash will give you a warning (when you publish) if—anywhere in your code—you try to assign a value that doesn't match the data type you declared. That way, it helps uncover mistakes in your code.

Finally, terms such as *object* and *class* get thrown around a lot but they're actually quite simple. I've been talking about Movie Clip instances because they're so common, but they're actually instances of the class `MovieClip`. You could say their object type is `MovieClip`. Instances of the class `MovieClip` are easy to understand because you make an instance by simply dragging the symbol from the Library to the Stage. But you can also create a `MovieClip` instance using the following:

```
var myMovieClip:MovieClip = new MovieClip()
```

You need to follow that with some code that puts something into the `myMovieClip` and you'll also need to use the following code to make the `MovieClip` appear onstage:

```
addChild( myMovieClip )
```

For most other object types you need to formally instantiate them ("make an instance"). You always instantiate an instance of a class by using the following form:

```
var myInstance = new MyClass()
```

You replace `MyClass` with the class you're instantiating. (By convention all classes begin with an uppercase character and instance names always begin with a lowercase character.) Sometimes the class you instantiate is part of Flash and other times the definition for its behavior resides in a homemade class file (`MyClass.as` for example).

After you have an instance of a class stored in a variable you can do anything you want with that instance. That is, the class will probably expose public methods (meaning methods you're allowed to access and trigger). Just as you can say `myMovieClip.play()`, you'll be able to say `myInstance.startMonitoring()` because I defined a public method called `startMonitoring()`. (Exactly what happens when that method gets triggered depends on what the programmer designed the class to do.) I should note that only public methods can only be triggered from outside the class as shown here—other qualifiers (private, internal, and protected) prevent access to methods from outside the class itself. Non-public methods are like internal utilities to help the class do its work. I've actually gone farther into custom classes than is needed for this book, but I wanted to show you the consistencies. There are a few more (more fundamental) concepts I want to cover now: arrays, ActionScript objects, and flow control.

Think of arrays as simply another data type. That is, you can store a string into a variable (and type it as a `String` just to be formal):

```
var myName:String = "phillip"
```

Alternatively, you can store a number:

```
var myAge:Number = 42
```

If you think of variables as storing a value for later use, you should think of a variable that holds an array as storing multiple values for later use. Here's one way to create and populate a variable to contain an array:

```
var myBooks:Array = [ "teach yourself", "actionscripting", "flash at work",
"RIAS" ]
```

The cool part about arrays is that not only do they hold multiple values but you can access each item individually by using bracket reference. For example, this code traces `"teach yourself"`:

```
trace ( myBooks[0] )
```

Notice that the first index is index 0. You can do other cool things with an array, such as sort the items in an array, add or remove items, or simply use the `length` property to ascertain how many items are present.

Arrays are good when you have multiple values that you want to store. But there's another way to store multiple values called ActionScript objects. The main thing ActionScript objects offer is a way to label each item in the variable. That wouldn't be terribly useful for the content in the `myBooks` array (maybe label the books book1, book2, and so on). In fact, an array is better in this case. Arrays are most appropriate for when you don't know how many items will be present, when you don't care (that is, maybe you'll add more later), or when it really doesn't make sense to think up a name for each item. Every item in an ActionScript object has a property name and a value. You can create and populate ActionScript objects two different ways. First I'll show you the literal (think "inline") manner:

```
var myObject:Object = { firstname: "Phillip", lastname:"Kerman", age:42 }
```

Notice that, between the curly braces, there are three items (separated by commas). Each item has a property name (which must not have spaces) followed by a colon and then a value. (I'll come back to the fact that the value can be any data type, though I'm currently using strings and numbers.)

A more formal way to create and populate an ActionScript object is to do it in two or more steps:

```
var myObject:Object = new Object();
myObject.firstname = "Phillip"
//and so on
```

The cool part is you can add arbitrary properties to an ActionScript object. The way to access the property values in an ActionScript object (`firstname`, `lastname`, and `age` in this case) is the same way you set (or reset) them:

```
trace ( myObject.firstname ) //displays "Phillip"
myObject.firstname = "Joe"
trace ( myObject.firstname ) //now displays "Joe"
```

What's really wild is that you can have property values that contain any data type, including arrays or other ActionScript objects. For example:

```
myObject.dogs = ["Max", "Hank", "Grettle"]
myObject.cats = []
```

Here I have an array for both—even though one array has a length of zero as I have no cats.

What's even more common is to store multiple ActionScript objects in an array like this:

```
var myArray:Array = []
myArray[0] = {firstname:"Joe", lastname:"Smith" }
myArray[1] = {firstname:"Bob", lastname:"Jones" }
```

You should see some similarities here with the `DataProvider` class described in Hour 17.

Finally, the last thing I want to cram into this appendix is a quick explanation of controlling the flow through your ActionScript. Normally, Flash executes each line of code in sequence. When a block of code is stored inside a custom function, Flash executes all those lines of code every time the function triggers. But what if you don't want all the code to execute every time? For this you can use the most basic of flow-control statements, the `if` statement. Depending on whether you think in the affirmative (half full) or negative (half empty), you can think of the `if` statement as doing certain lines of code "if" a certain condition is true…or similarly, if a certain condition is true, skip certain lines of code. Here's the basic skeleton:

```
if  ( condition ) {
   //code to execute when condition is true
}
```

You replace "condition" with an expression that results in true or false. If you write that expression dynamically, it will be true sometimes and false sometimes. For example:

```
if  (  clip.x > 100 ) {
    clip.x = 0
}
```

When this code executes, Flash looks at the condition and, if it's true (that is, the clip's `x` property is greater than 100), then Flash executes the second line. That is, Flash executes all code that appears before this code, then if the condition is true, it executes all the lines between the curly braces (only one line in this case) and then (always) proceeds to execute any code that appears after the closing brace.

People understand that the code between the curly braces of an `if` statement execute only when the condition is true, but a common misunderstanding is they think that code automatically executes any time the condition becomes true. Remember though, that only after an `if` statement is encountered will the condition be considered and the inside be (potentially) executed. That is, if the `if` statement appears in the first frame of your movie, it is considered only once when that frame is first loaded. However, if you put the `if` statement inside a function that gets called every time the user clicks a button, then the `if` statement will be encountered repeatedly.

I could go on, but for me I distinctly remember various epiphanies I've had while learning ActionScript (which, by the way, I'm still doing). A few highlights include really understanding `if` statements, arrays, and objects. I hope this primer has helped you move ahead from wherever you are in your programming career.