The Three R's: Readability, Reusability, and Reachability

Each Flash programmer tends to develop his or her own style of writing, organizing, and commenting programs. Companies that employ Flash programmers also have different standards. However, whatever your individual style, you probably want to write code that is readable, reachable , and perhaps reusable. The longer and more complex your programs are, the more important it is to think about these goals before and as you write.

Readability: Writing Readable Code

When you go back to change, enhance, or debug a program, it's nice to have code that you can decipher without too much strain on your eyes or brain. Readability is partly just a matter of spacing, indentation, and capitalization. Comments can also add a lot to readability, as can the use of meaningful names for functions, variables , arrays, and objects. Finally, the best (and hardest) way of making code more readable is by solving coding problems in more elegant, simpler ways.

Spacing, Indentation, and Capitalization

With few exceptions, ActionScript is very flexible about spacing and indentation. FP6 (and FP7 when playing movies published for FP6) is also forgiving when it comes to capitalization.

For example, when playing movies published for FP6, these two functions look the same to the ActionScript interpreter (the software that interprets SWF files):

```
function addonetobasescore(basescore){return basescore+1;} function addOneToBaseScore (baseScore)
{ return baseScore + 1; }
```

Whitespace, capitalization, and indentation make the second significantly more readable, and thus make it easier for you to understand and edit your program.

Remember, however, when publishing for FP7, capitalization affects the way your program works, not just the way it looks.

Adding Comments to Your Script

Just formatting the code is seldom enough to make it clear what's going on in a complex program. Comments are usually necessary, too. Single-line comments are marked by double slashes :

```
// input: base score, returns: one more than the base score
```

A comment can also follow code:

```
baseScore += 1; // baseScore declared frame 1, main timeline
```

Enclose multiline comments like this:

```
/* ADD ONE TO BASE SCORE Michael Hurwicz - January, 2004 www.greendept.com baseScore is an integer */
```

Some people prefer double slashes on every line because they make it clear that each line is still part of the comment:

```
///////////////////////////////////////////////////////////////// // ADD ONE TO BASE SCORE // Michael Hurwicz - January, 2004 www.greendept.com // baseScore is an integer
/////////////////////////////////////////////////////////////////
```

In the Script pane, comments are color-coded, so you know where they start and stop. You can customize the color coding by selecting Edit, Preferences, ActionScript, and clicking in a color swatch in the Syntax coloring section. A selection of color swatches pops up, and you can select one. In this way, you can use colors for displaying foreground, background, keywords, comments, identifiers, and strings.

Creating Readable Names

You've seen that ActionScript programs have several types of elements that require names, such as functions, arrays, objects, and variables. What makes such names readable?

You've already encountered one aspect of readability for names: capitalization. Reading addOneToBaseScore is easier than reading addonetobasescore . Traditionally, by the way, most names in Flash begin with lowercase letters . An initial capital indicates a class.

Another good technique is to give things names that you (and others, if necessary) will understand intuitively. For instance, you probably won't have to examine the details of the addOneToBaseScore function to remember what it does.

Simplifying, Standardizing, and Optimizing Code

The best way to make your code more readable is to come up with straightforward ways of accomplishing your programming goals. Simple code will make your life as a programmer much easier. Of course, creating masterfully elegant code is easier said than done, but it's a worthy objective.

Better Coding with Pseudo-Code

One tool that may help you write simpler, more elegant code is pseudo-code. Pseudo-code represents your program, or some part of it, in your own words, but with as much specificity and detail as possible, and by trying to follow the flow and layout of the program as you envision it. Pseudo-code enables you to focus on the logic of your program, without getting bogged down in the mechanics of ActionScript. It allows you to mentally experiment with different approaches to a problem, without wasting time implementing approaches that you ultimately reject. When you do start coding, the pseudo-code serves as a detailed outline for the finished product, helping you stay on track.

Here's a very simple example, taken from leglift.fla, which is discussed in the next section. The central function in leglift.fla is lift() , which is called repetitively to gradually lift a leg off the ground or lower it back down. It can also stop the leg from moving. Movements are accomplished by rotating the leg at the hip. Here is the pseudo-code:

```
function graduallyApproachGoal() receives a parameter ( goal ).
```

If the current rotation of the leg is less than goal , increase the rotation of the leg one degree.

If the current rotation of the leg is greater than goal , decrease the rotation of the leg one degree.

If the current rotation is the same as goal , do nothing. (The leg will not move.)

You can see this function implemented as lift() on page 451 in the next section.

Trying to standardize your code is also helpful so that you can use the same techniques or syntax over and over. You'll understand blocks of code at a glance because you'll be so familiar with the techniques and syntax involved.

Simpler code also usually performs better than more complicated code. Because you will, of course, standardize on your best code examples, standardization is likely to make your program run better all around. Concise coding will also permit faster debugging and easier editing.

In some cases, however, a conflict occurs between fast code and easy-to-read code. In those cases, one approach is to actually use the fast code in the program but retain the easy-to-read code as a comment.

Reusability: Modularizing Code

Most programmers reuse code to some extent. After all, programming is essentially problem-solving. If you've solved a problem before, why reinvent the wheel? The techniques used to achieve reusability also make code more readable and easier to debug. Any time you find yourself typing similar code over and over, or cutting and pasting code, look for a better approach.

The two main vehicles of reusability are functions and objects. Both allow you to package functionality in modules that you can reuse both within a program and among multiple programs. Arrays can also play the same role, if you want access via numeric indices.

In addition, packaging functionality in functions and objects makes your code more readable. For instance, suppose you find this code inside an "enter frame" event handler attached to the left leg of a figure:

```
degrees++; if (degrees < 30) { _rotation++; }
```

You know that every time the movie clip enters a frame, the ActionScript interpreter is going to do whatever this code says. But what is that, exactly?

Compare that example to the following line, which invokes a lift() function, passing it an argument of 30.

NOTE

An argument is data that you pass to a function, enclosed in parentheses after the function name .

```
lift (30);
```

You don't need to be a genius to guess that the figure is probably lifting its left leg, and that the argument, 30, has something to do with how much it's lifting the leg. You get that increase in readability just by putting the less intuitive code in a function with an intuitive name.

Now, let's make that leg part of an object named guy , with a leftLeg property. The leftLeg property is also an object, with a lift() method:

```
guy.leftLeg.lift(30);
```

It's practically a full sentence , with subject, object, verb, and adjective: A guy is lifting his left leg.

How do you make the leg part of an object named guy ? You take advantage of the fact that movie clips are objects. Here, guy is a movie clip, and leftLeg is a clip within it. The lift() function becomes a method of leftLeg when it is defined as a property of the leftLeg object.

Both lift() and guy.leftLeg.lift() assume that somewhere else in your program you have defined a function called "lift" that does something similar to the less intuitive code shown on page 451. If you look at that function definition, you can get a more precise idea about what lifting means in this context ”namely, that it has to do with rotating the clip:

```
function lift (degrees) { if (_rotation > degrees) { _rotation--; } }
```

Add a comment to the function definition ”"used to lift leftLeg" ”and you have some very readable code.

The sample program leglift.fla shows a slight elaboration of this approach. It adds a second part to the lift() function so that the function lowers the leg if _rotation is less than degrees and raises the leg if _rotation is greater than degrees . If degrees equals _rotation , the function does nothing; that's the basis for stopping the leg.

```
// used to raise, lower and stop left leg function lift(degrees) { if (_rotation < degrees) { _rotation++; //
lower the leg } if (_rotation > degrees) { _rotation--; // raise the leg } }
```

The sample also defines three functions that change the degrees variable, thus raising, lowering, or stopping the leg:

```
function raise() { degrees = 20; } function lower() { degrees = 68; } function halt() { degrees = _rotation; }
// lift() will do nothing
```

These three functions are invoked from three buttons on the Stage. The "enter frame" event handler of the leg clip is constantly executing the lift() function, so the appropriate action is triggered as soon as degrees changes.

Using functions and objects also tends to make you a better programmer. This goes back to the fact that programming is problem-solving, and the first steps in problem-solving are defining the problem domain and defining the problem itself.

In the process of defining objects, you also define problem domains precisely. In the leg lift example, for instance, the domain of your problem is guy.leftLeg . Similarly, in the process of defining a function, you also define the problem. For instance, the previous function defines the problem as lifting the leg a certain number of degrees.

Functions and objects also help you break down large problems into smaller problems. Suppose you want a cartoon figure to walk across a room. A walkAcrossTheRoom() function might be relatively complicated. Perhaps you could write a takeOneStep() function and repeat it until the figure gets across the room. So, what is involved in taking one step? The figure has to swing an arm, move a leg, and move a certain distance forward.

If you keep going like this, eventually you'll come to problems that are small and precise enough to get your mind around and solve. You may start by implementing your solutions as functions. You could then combine these functions into more complex functions, objects, or sections of code. Objects and functions are great problem-solving tools because they help you solve dauntingly complex problems one simple step at a time.

Ready to start programming, but don't know where to start? See the "Troubleshooting" section later in this chapter, page 458 .

Avoiding the supposed complexities of functions and objects is like avoiding the complexities of folders when you're managing email or files on a hard disk. As the number of messages or files grows, the lack of organization begins to hamper your ability to work efficiently . Similarly, objects and functions are basic to organizing ActionScript. They should make your life easier, not harder.