19

Debugging

So far we've explored a lot of techniques and also the syntax to accomplish many goals. Inevitably, however, when you begin writing your own ActionScript, you'll encounter innumerable errors (especially at first when you are still making syntax and conceptual errors). Do not lose heart! Even experienced programmers spend a lot of time *debugging* (fixing broken code).

It is important that you test your product thoroughly so that you can find the bugs in the first place. This means testing in various browser brands and versions of those brands on all platforms that you intend to support. Test under different flavors of Windows and, if applicable, older versions of the Flash plug-in, which you can find at:

http://www.macromedia.com/support/flash/ts/documents/oldplayers.htm

A discussion of testing and quality assurance (QA) is beyond the scope of this book. Suffice to say that you should have a testing and QA process in place and a bug report form on which you can receive reports with sufficient detail (such as the platform, browser version, Flash plug-in version, and reproducible steps) for you to reproduce the error, which is the first step toward fixing it.

Debugging is an essential part of programming and what sets great programmers apart from average ones. Beginners are often happy if a bug that was seen earlier inexplicably disappears. Experienced programmers know that the bug will undoubtedly resurface at the most inopportune time, and although it is intermittent (perhaps especially so), it warrants further investigation. On the other hand, inexperienced programmers tend to shy away from error messages or be unnerved by obvious errors, whereas skilled programmers rely heavily on error messages and know that easily reproducible errors are the easiest kind to fix.

Successful debugging requires logical, disciplined investigative skills and a decent understanding of troubleshooting tools. In this chapter, we'll briefly consider the basics of debugging tools and some general techniques for solving code problems. Remember that debugging is characterized by the systematic challenging of our assumptions. Any given problem is often caused by some other problem upstream (i.e., the disease). We'll use the debugging tools to investigate whether things are in fact operating as designed, and that will lead to an understanding and resolution of the manifest bug (i.e., the symptom).

Debugging Tools

ActionScript comes equipped with the following debugging tools:

- The *trace()* function
- The List Variables command
- The List Objects command
- The Bandwidth Profiler
- The Debugger

All of these tools are used in Test Movie mode. To enter Test Movie mode, we export a movie from the authoring tool using Control → Test Movie.

In addition to these formal debugging tools, Flash also sends error messages to the Output window when a movie is exported or Check Syntax is performed. (Check Syntax is a command listed under the arrow button in the top right of the Actions panel.) Error messages often identify the exact cause of a problem down to the problematic line number in a block of source code. Comprehensive explanations for the various error messages are provided in Macromedia's ActionScript Reference Guide.

Note that not all bugs cause error messages. For example, a calculation that yields the wrong result is a bug even if it doesn't crash your browser. Also note that there are two types of error messages, so-called *compile-time* error messages that occur when you try to export your scripts and so-called *runtime* error messages that don't occur until you run your Flash movie and reach the point that causes the error. Compile-time errors indicate some sort of syntax problem such as a missing parenthesis or unclosed quotation. Refer to Part III, *Language Reference*, for the exact syntax needed for each command, and refer to Chapter 14, *Lexical Structure*, for an explanation of proper ActionScript syntax.

Runtime errors can take a wide variety of forms and may not indicate a problem with the current code under examination but rather may be caused by using the incorrect result of an earlier operation. For example, suppose you try to use the

values received back from a *loadVariables()* command sent to a web server. If the Perl script responding to the command didn't supply the correct data in the correct format, you need to correct the Perl script. Your Flash script may be perfectly correct and yet fail because it received incorrect input.

Which brings up an important technique—defensive programming. You can avoid a lot of errors and potential errors by always checking for potential problematic conditions, which is known as *error checking* (or sometimes *data validation* if it pertains to user input). For example, before trying to display the questions of a quiz, you might check that those questions loaded properly. You might also check each question to be sure it's in the correct format for display. If the provided data was improperly entered, you should display an appropriate error message that allows the programmer or the user to take corrective action.

The trace() Function

In ActionScript, one of the most effective tools for identifying the source of a bug is also one of the simplest—the *trace()* function. As we've seen throughout this book, *trace()* sends the value of an expression to the Output window in Test Movie mode. For example, if we add the following code to a movie:

```
trace("hello world");
```

the text "hello world" appears in the Output window. Similarly, here we *trace()* the value of a variable:

```
var x = 5;
trace(x);  // Displays 5 in the Output window
```

Using *trace()* we may check the status of variables, properties, and objects, and we may track the progression of our code. Often by confirming the result of each operation in a script, we can figure out where a problem lies. For example, suppose a function is supposed to return a value but we find, using the *trace()* command, that the return value is `undefined` (i.e., it prints out as nothing in the Output window). We'd know that we have to examine the function in more detail and make sure that it is properly using a *return* command to pass back a meaningful value.

The List Variables Command

When a movie is running in Test Movie mode, we can check the value of current variables defined in the movie via the Debug → List Variables command. List Variables tells us the name and location of all the variables currently active in our movie and also reports their values. Because functions and movie clips are stored in variables, the List Variables command also shows us the functions and movie clips of a movie.

Example 19-1 shows sample output from List Variables. Notice that the variable `rate` is shown as declared but `undefined`. This subtlety is often difficult to detect with *trace()* because *trace()* converts the value `undefined` to the empty string (`""`).

Example 19-1. List Variables Sample Output

```
Level #0:
  Variable _level0.$version = "WIN 5,0,30,0"
  Variable _level0.calcDist = [function]
  Variable _level0.deltaX = 194
  Variable _level0.deltaY = 179
  Variable _level0.rate = undefined
  Variable _level0.dist = 264
Movie Clip:  Target="_level0.clip1"
Movie Clip:  Target="_level0.clip2"
```

Note that both *trace()* and the List Variables command give only a snapshot in time. Often, you'll want to monitor the value of a variable over time or check it repeatedly. The Debugger (discussed later) allows you to track the value of a variable as it changes.

The List Objects Command

The List Objects command produces a catalog of text, shapes, graphics, and movie clips defined in a movie. To execute it, select Debug → List Objects while in Test Movie mode. Note that List Objects does not include a list of data objects (instances of a class) in a program; those are reported by List Variables.

Example 19-2 shows some sample output from List Objects. Notice that editable text fields are clearly labeled and that automatically named movie clip instances are revealed (e.g., `_level0.instance1`).

Example 19-2. List Objects Sample Output

```
Level #0: Frame=1
  Shape:
  Text: Value = "variables functions clip events startDrag stopDrag Math"
  Text: Value = "this movie demonstrates a little math, variables, movie clip events"
  Text: Value = "draggable distance"
  Text: Value = "calculator"
  Movie Clip: Frame=1 Target="_level0.instance1"
    Shape:
  Text: Value = "distance between clipstotal:horizontal:vertical:"
  Edit Text: Variable=_level0.dist Text="222"
  Edit Text: Variable=_level0.deltaX Text="174"
  Edit Text: Variable=_level0.deltaY Text="138"
  Movie Clip: Frame=1 Target="_level0.obj1"
    Shape:
  Movie Clip: Frame=1 Target="_level0.obj2"
    Shape:
```

Again, List Objects provides only a snapshot in time. You need to run it again to get the current value of objects whenever they may have changed.

The Bandwidth Profiler

The Bandwidth Profiler is used to simulate movie download at various modem speeds. Using the Bandwidth Profiler, we may gauge the performance of a movie, test preloading code, and track the position of the main movie's playhead during movie playback. Here's how to turn the Bandwidth Profiler on:

1. While in Test Movie mode, select View → Bandwidth Profiler.
2. Under the Debug menu, select the desired download rate.
3. To simulate the download of a movie at that rate, select View → Show Streaming.

There are many things that can affect Flash performance, such as the assets in use and the rendering demands on the Player. For example, using large bitmaps, rendering complex shapes with many curves, and excessive use of alpha channels can all degrade performance. Asset downloading and rendering times usually dwarf the bandwidth and processor time required for ActionScript to execute. That said, ActionScript is generally much slower than compiled languages such as C.

From an ActionScript perspective, the most time-consuming operations are those that either must wait for data to be uploaded or downloaded or those that are performed repetitively (such as examining a large array).

Displaying items in the Output window is very slow compared to "invisible" operations in ActionScript. If a simple movie seems excessively choppy, try disabling all *trace()* statements, or play the movie outside Test Movie mode.

A discussion of writing optimized code is beyond the scope of this book, but some quick tips should suffice:

- Don't perform an operation repeatedly within a loop if it can be performed once outside of a loop with no loss of functionality.
- Don't wait in a loop for some event to occur. The event may take a long time or may never occur, causing your performance to slow or your application to lock up entirely. Instead, rely on event handlers, such as *on (load)* to be triggered when an event occurs or completes.

- Generalize your code wherever possible (perhaps even use Smart Clips to do so). This reduces the size of the code that needs to be downloaded. For example, instead of writing two nearly identical routines that are each 5 KB long, you can save 5 KB by writing one generalized routine and calling it twice with different parameters. (Generalizing code is explained in Chapter 9, *Functions*, and Smart Clips are explained in Chapter 16, *ActionScript Authoring Environment*).
- If you're using the Flash 5 Player and optimized ActionScript performance is critical to your project, try using old-style Flash 4 syntax instead of newer techniques. In Flash 5, certain operations are faster when phrased with Flash 4 syntax. For example, Flash 4's *substring()* function is faster than Flash 5's *substring()* and *substr()* methods, and Flash 4's *Tell Target* is faster than Flash 5's dot notation.
- Export your movies without *trace()* statements by selecting Publish Settings → Flash → Options → Omit Trace Actions.
- Remember that removing and reattaching a clip is more costly than moving an existing one; reuse your movie assets whenever possible.
- For a list of general Flash optimization techniques, see *http://www.macromedia.com/support/flash/publishexport/stream_optimize/stream_optimize.html*.

The Debugger

The Debugger is a highly useful tool that gives us organized access to the values of properties, objects, and variables in a movie and even allows us to change variable values at runtime.

To enable the Debugger, select Control → Debug Movie in the Flash authoring tool (*not* in Test Movie mode). You may also use the Debugger in a web browser, provided that:

- The movie being viewed was originally exported with debugging permitted.
- The Player being used to view the movie is a debugging Player.
- The Flash authoring tool is running when you attempt to debug.

To export a movie with in-browser debugging permitted, select File → Publish Settings → Flash → Debugging Permitted, then optionally provide a password to prevent prying eyes from snooping around your code. To install a debugging Player in your browser, use the installers provided in the */Players/Debug/* folder where you installed Flash on your hard drive. To enable debugging while viewing a movie, right-click in Windows (Ctrl-click on Macintosh) on the movie and select Debugger.

Not all versions of the Flash Player have a corresponding debugging Player. Check Macromedia's support site for the newest versions of the debugging Player, *http://www.macromedia.com/support/flash*.

The top half of the Debugger (the Display List) shows the movie clip hierarchy of the movie. To inspect the properties and variables of a specific movie clip, select it in the Display List. The bottom half of the Debugger contains three tabs, Properties, Variables, and Watch, which update dynamically to show the properties and variables for the selected clip. To set the value of any property or variable, double-click its value and enter the new data. To single out one or more items for convenient scrutiny, select them in the Properties or Variables tab, then choose Add Watch from the arrow button in the upper-right corner of the Debugger. All "watched" variables are added to the Watch tab (this lets us view variables in different movie clips simultaneously).

For more information about the mechanics of using the Flash Debugger, consult Macromedia's thorough documentation under "Troubleshooting ActionScript" in the ActionScript Reference Guide. If you've lost your Reference Guide, remember that it's available on Macromedia's web site at *http://www.macromedia.com/support/flash* and also under the Help menu in the Flash authoring tool.

Debugging Methodology

Let's take a quick look at some techniques involved in code debugging. Debugging can be broken into three stages:

- Recognizing and reproducing a problem
- Identifying the source of the problem
- Fixing the problem

Recognizing Bugs

Very often, we recognize code problems as part of the active process of programming. That is, we write some code, test our movie, and find that the movie doesn't work properly. Problem recognized.

The earlier a problem is discovered, the better. The process of writing code should therefore be a constant ebb and flow of writing and testing—write a few lines, export the movie, make sure the lines work as expected, then write a few more

lines, export the movie, and so on. Make sure each component of a program works on its own before testing the program as a whole. Try not to get carried away writing a complex body of code without testing it frequently along the way.

Don't assume your movie is perfect just because you can't find any bugs on your own. Always schedule time for external testing by target users, particularly if the code you are delivering is part of a product or a service intended for a client. As described earlier, implement error checking to head off possible problems with incorrect data input. For example, if you write a function that expects an integer argument, you might use the *typeof* operator to verify that the input parameters are of the correct type. Also test *end conditions* such as extremely large, small, and negative values, including zero.

Don't underestimate the value of finding the *minimum reproducible steps* that replicate the problem. These should be the fewest steps that recreate the error reliably. A bug report such as, "I played it for an hour and then it froze" is not very helpful. Useful bug reports include numbered steps such as:

1. Enter 0 for the number of years.
2. Click the Calculate button.
3. The results field shows "NaN" instead of a dollar amount.

Identifying the Source of a Bug

Once we've recognized a bug, our quest for a solution has only begun. Our first task is to find the source of the bug, however far upstream that may be. A bug can be thought of like a heart attack that was caused by bad dietary habits years earlier. The heart attack is merely the most manifest symptom, but you must often correct something earlier in the process. Most bugs are caused by false assumptions; we assume we've typed the name of a variable correctly but we haven't, or we assume a text field stores numeric data but it doesn't. By executing a series of *trace()* statements or using the Debugger or List Variables command, we can test our assumptions against the interpreter's understanding of our code.

Here, for example, is some code with a bug. It incorrectly sets `status` to "equal":

```
var x = 11;
isTen(x);
function isTen(val) {
  if (val = 10) {
    status = "equal";
  }
}
```

To find out what's wrong with the code, we compare what we *think* the code should be doing against what it *actually* is doing, one step at a time:

```
// This should set x to 11
var x = 11;

// Let's see if it really does
trace(x);     // Yup...this displays: 11

// This should invoke the isTen() function
isTen(x);

// Now on to our function
function isTen(val) {
  // Let's make sure our function is being called
  trace("isTen was called");    // Yup...this displays: "isTen was called"

  // Now let's make sure our parameter was passed correctly
  trace("val is " + val);       // Yup...this displays: "val is 11"
```

Let's pause here for a second. Notice what's happened—we've made it most of the way through our code and so far everything has worked as expected. Our variable was set correctly; *isTen()* was called and received its argument properly.

Many errors occur because the code that you think is being executed has never even been reached! We can use *trace()* statements to verify that a particular portion of our code is reached.

By process of elimination, we already know that our code's problem must lie either in the conditional statement `if(val = 10)` or in the text field assignment `status = "equal"`. We next check our conditional statement by using *trace()* to display the value of its test expression (we're expecting either `true` or `false`):

```
trace(val = 10);
```

Eureka! The Output window displays `10`, not `true` or `false` as we had expected.

On closer inspection, we see that the test expression is an assignment statement, not a comparison statement! We forgot an equal sign in our equality comparison operator. The expression `if(val = 10)` should be `if(val == 10)`.

Obviously, not all bugs are as simple as our conditional statement bug (which is an exceedingly common error), but the approach we used is applicable to most bug hunts: execute a series of *trace()* functions to create a running, step-by-step report on the actual behavior of a movie's code and use the Debugger as explained in the Macromedia documentation.

Common Sources of Bugs

Table 19-1 lists some common sources of bugs in ActionScript.

Table 19-1. ActionScript Gotchas

Problem	Description
Code in the wrong place	All code must be attached to a movie clip, frame, or button. Take care that your code is actually attached to what you intend by observing the title of the Actions panel—when attaching code to a frame, the Actions panel's title reads Frame Actions; when attaching code to a movie clip or button, the Actions panel title reads Object Actions. If you want a script to be on a particular frame, make sure that frame is selected in the timeline before you start coding, and that there's a key-frame where you want to place your code. If you want a script to be on a movie clip or button, make sure that object is selected on stage before you start coding. Use the Movie Explorer (Window → Movie Explorer) to keep track of exactly where code is attached.

Missing event handler	Code attached to movie clips and buttons *must* be contained by an event handler. For movie clips, use:
	`onClipEvent (event) {`
	`  // statements`
	`}`
	For buttons, use:
	`on (event) {`
	`  // statements`
	`}`
	where *event* is the name of the event to handle. The error, "Statement must appear within on handler," indicates that you're missing an event handler. See Chapter 10.

Bad movie clip reference	A movie clip that doesn't exist is referenced, or a reference to a movie clip is malformed. Check that all instances are named, and that instance names match the reference supplied. See "Referring to Nested Instances" in Chapter 13 for information on composing valid movie clip references.
Unexpected type conversion	The result of a data conversion yields an unexpected result. For example, `3 + "4"` yields the string "34", not the number 7. Similarly, the string "true" converts to the Boolean value `false`! Study type conversion rules in Chapter 3. Check datatypes using the *typeof* operator.
Missing semicolon	A statement ends prematurely because a semicolon is missing. See Chapter 14 for proper semicolon usage.
Problem quotation mark	A string includes an unescaped quotation character that interferes with the string literal. See "String Literals" in Chapter 4.
Bad text field data usage	A text field is treated as a number or other datatype, not a string. User input in text fields is always a string value and should be converted manually before being treated as any other type.

Table 19-1. ActionScript Gotchas (continued)

Problem	Description
Scope problems	A variable, property, clip, or function is referenced in the wrong scope. For example, a statement in a clip handler attempts to invoke a function scoped to that clip's parent timeline. See "Event Handler Scope" in Chapter 10, "Variable Scope" in Chapter 2, and "Function Availability and Life Span" in Chapter 9.
Global function versus method confusion	Some global functions have the same name as movie clip methods. Occasionally, this overlap causes problems. See "Method versus global function overlap issues," in Chapter 13.
Content not yet loaded	A reference to a clip, property, function, or variable can't be resolved because the content is not yet loaded. Be sure all content is loaded by checking the *MovieClip*.`_framesloaded` property as shown in Part III.
Incorrect capitalization	Some keywords are case sensitive in ActionScript. If you miscapitalize *onClipEvent* as *onclipevent*, ActionScript will think you are trying to call a custom function named *onclipevent* instead of using the built-in *onClipEvent* handler keyword. As such, it will give you an error when it encounters the `{` at the beginning of the *onClipEvent* statement block (it expects a semicolon indicating the end of what it perceives to be an *onclipevent* function call). See "Case Sensitivity" in Chapter 14.

Fixing Bugs

In some cases, the fix for an identified bug is self-evident. For example, if we discover a bug caused by a missing quotation mark on a string, we fix the bug by adding the quotation mark.

In more involved programs, fixing bugs can be a serious challenge. If a bug is proving difficult to fix, consider the following:

- Don't be afraid to rewrite code. In many cases the best way to fix overly complicated code is to rearchitect the system and start from scratch. Recreating a program nearly always goes faster and smoother than creating the program in the first place. Most experts agree on this one (for example, Quake III was a complete rewrite of the Quake II engine). That said, new code still needs to be debugged. Don't throw out perfectly good code close to a deadline. Keep what's good and rewrite only the problematic code.
- Break problematic components out into separate test movies. Work on each aspect of a system in complete isolation, then integrate working sections one at a time.

- Have a peer review your code. Don't be afraid. We're all embarrassed by the code we wrote a year earlier.
- Ask for help at one of the resources cited in Appendix A, *Resources*. For example, the FlashCoders mailing list is devoted entirely to ActionScript questions.

For lots of good advice on programming techniques, see *Extreme Programming Explained* by Kent Beck (Addison Wesley) and *Code Complete* by Steve McConnell (Microsoft Press).

Onward!

Our ActionScript conversation is over, but yours has just begun. By reading Part I, *ActionScript Fundamentals* and Part II, *Applied ActionScript*, you've learned to speak ActionScript—now it's time to apply that knowledge to your own projects. Before you embark, here are a few parting thoughts:

- As with any art form, learning to program is a process not an event. For as long as you program, you'll learn more about programming. Consider the multiple-choice quiz example from Chapters 1, 9, 11, and 13—we rebuilt it four times! Each time we refined our approach, added features, and learned something we hadn't considered before. Just as each painting teaches the painter something new about her subject matter, so creating and recreating applications will reveal new approaches to you.
- There's practical help in Part III, which contains detailed descriptions of ActionScript's built-in functions, properties, classes, and objects. While you may not want to read it from start to finish, you should definitely skim each topic so you have a sense of ActionScript's capabilities.
- Revisit this book and Part III often. You'll pick up new insights each time through because you'll be considering the information with a higher level of understanding and will be able to relate concepts to real-world experiences. Treat Part III as a dictionary and keep it by your side while you work.
- There's a thriving community of Flash developers out there offering ideas and solutions and—most importantly—sharing source code! Dissect as much as you can. Identify the things you can't understand and look those topics up in this book. Macromedia maintains a list of web sites and mailing lists devoted to Flash at *http://www.macromedia.com/support/flash/ts/documents/flash_websites.htm*.
- Don't limit your exploration of ActionScript to this book. Look at things from multiple angles by consulting other sources of knowledge and inspiration, such as those listed in Appendix A and the Preface. You'll also find a long list

of Flash resources at *http://www.moock.org/moockmarks* and reviews of worthwhile Flash books at *http://www.moock.org/webdesign/books*.

- Finally, remember to drop by the ActionScript Definitive Guide support site, *http://www.moock.org/asdg*, for lots of code samples, tech notes, and discussions of new topics.

With that, I wish you happy coding! Throw an extra iteration into a *while* loop for me sometime. :)