Chapter 45: Debugging Scripts

In This Chapter

Identifying the type of error plaguing a script

Interpreting error messages

Preventing problems before they occur

One of the first questions that an experienced programmer asks about a programming environment is what support is there for debugging code. Even the best coders in the world make mistakes when they draft programs. Sometimes, the mistakes are a mere slip of a finger on the keyboard; other times, they result from not being careful with expression evaluation or object references. The cause of the mistake is not the issue: finding the mistake and getting help to fix it is.

Some debugging tools are available for the latest browsers. For the most part, they have come from the browser makers themselves, or they are tied very closely to a particular authoring environment. Some of these tools are very quirky; others require significant investments in authoring environments. Discussion about debugging tools in this chapter, however, focuses on simple tools provided on the companion CD-ROM. By understanding the true meaning of error messages and working out the problem with the tools provided here, you should be able to overcome your bugs.

Syntax versus Runtime Errors

As a page loads into a JavaScript-enabled browser, the browser attempts to create an object model out of the HTML and JavaScript code in the document. Some types of errors crop up at this point. These are mostly syntax errors, such as failing to include a closing brace after a function's statements. Such errors are structural in nature, rather than about values or object references.

Runtime errors involve failed connections between function calls and their functions, mismatched data types, and undeclared variables located on the wrong side of assignment operators. Such runtime errors can occur as the page loads if the script lines run immediately as the page loads. Runtime errors located in functions won't crop up until the functions are called — either as the page loads or in response to user action.

Because of the interpreted nature of JavaScript, the distinction between syntax and runtime errors blurs. But as you work through whatever problem halts a page from loading or a script from running, you have to be aware of differences between true errors in language and your errors in logic or evaluation.

Error Message Notification

As the browsers have evolved through several generations, the ways in which script errors are reported to the user (and to you as the author) have also changed. The biggest changes came in IE4/Windows and NN4.5. Prior to those versions, script errors always displayed some kind of alert dialog box with information about the error. Because these alerts could confuse non-technical users, the newer browsers (except for IE/Mac) are much more subtle about the presence of errors. In fact the notification mechanism is so subtle, that it is easy to miss the fact that a script error has occurred. Even if you do notice, you must then exercise your mouse a bit more to view the details.

When a script error occurs in IE4+/Windows, the statusbar displays a yellow alert icon plus a brief text message indicating that an error has occurred. A syntax error that occurs while the page loads usually signifies that the page has loaded, but with errors. A runtime error's message simply indicates that an error occurred. To view details about the error, you must double-click the yellow icon in the statusbar. The default appearance of the error message alert dialog box window includes a button named Show Details. Clicking this button expands the window to reveal whatever details the browser is reporting about the error. If you leave the window expanded, the next time it opens, it will also be expanded. It is a good idea for scripters to also check the box that forces the browser to show the error dialog box whenever an error occurs. This is simply a shortcut to manually double-clicking the statusbar error icon.

Netscape console windows

For NN4 browsers starting with NN4.5, a user receives error notification through a message in the statusbar. The instructions there indicate how to view the error details: If you type

```
javascript:
```

into the toolbar's Location box or into the dialog box that lets you open a new page, an entirely new, non-modal window appears. This window is called the Communicator Console. In contrast to the one message per window approach of IE, the Communicator Console window continues to record all script errors in sequence (in a scrolling frame), even when the Console window is closed. You can keep this window open all the time, and simply bring it to the front whenever you need to view errors. If you are developing on a large video monitor, you can let the Console window stick out to the right of the browser window. If an error occurs, not only does the message appear in the browser statusbar, but you'll also see the scrollbar of the Console window's top frame appear — an even more explicit indication that an error occurred (assuming you click the Clear Console button every time you are ready to try another test run).

Netscape changed the name of the window for NN6, now calling it the JavaScript Console. Opening this window is less cryptic than before: Choose Tasks/Tools/JavaScript Console from the menu bar. NN6 does not provide notification of errors in the statusbar, so it is up to you to be vigilant for something running amok. This is all the more reason to

keep the JavaScript Console window open while you are writing and debugging your scripts. Even if things appear to be OK, periodically check the Console window to be sure.

Multiple error messages

The modality of IE error message alert dialog boxes tends to force just one message to appear. In other words, when the first error occurs, the browser stops recording further errors. In NN, however, it is not uncommon for multiple errors to be listed (or, in older versions, multiple error windows to show up). But you need to understand how to treat these multiple errors to get to the root of the problem.

The usual reaction is to look at the last message to appear in the sequence. That, however, is usually the error message least likely to lead you to the true problem. Error messages are dumped to the NN Console window in the order in which they occur. This means that the first error in the list is the most important error message of them all. More than likely, the first error points to a problem that throws off the rest of the script, thus triggering all of the other error messages. For example, if a statement that initializes a variable has a syntax error in it, all other statements that rely on that variable will fail, because the variable appears to be undefined.

When you encounter multiple errors, don't start any serious debugging until you locate the first error message. You must tackle this one before any others. The solution to the first one may cause the other errors to go away. This is all the more reason, when authoring in NN4.5+, to keep the Console window open, and clear it before loading any page or executing any scripts.

Error Message Details

Error reporting comes in three flavors depending on the browser: NN, IE/Windows, IE/Mac. One of these groups may be better (that is, more accurate and explicit) at reporting some kinds of errors than the other groups. By and large, however, you can count on error details to include three basic clues to help you track down the error: the file in which the error occurred, the location of the error within the source code, and a textual description of the error.

Error file name

Although IE/Mac error messages do not explicitly reveal the name of the file whose source code contains the error, in practice, only the NN browsers do the best job of telling the truth. Of course, when the script and HTML are all on one page, it doesn't require a brain surgeon to know that the error occurs from that page's source code. But if you link in external `.js` libraries, the NN browsers provide the URL to the `.js` file. IE/Windows, on the other hand, indicates the HTML page that loads the external library, making it difficult to know precisely where the error is.

Error location

All browsers provide a source code line number and character position where the error supposedly occurs. For self-contained pages with no dynamically created content, the reporting tends to be accurate (also see the IE “Object expected” error message details described later in this chapter), but the accuracy is much closer in NN browsers than IE. And if your page links in an external library, the line number provided by IE/Windows and IE/Mac is practically useless. The sense you get is that the lines of the `.js` file become embedded within the main page’s script, but how that is supposed to help an author find the precise problem line is a mystery — even the most feature-laden text editor knows only how to display line numbers for a single document.

NN browsers, however, not only point to the correct `.js` file, but to the line number within that file. You are much more likely to get to the root of a problem, especially in an external `.js` file, through NN error messages.

Line number reporting has improved with each browser generation, but anomalies still exist. Perhaps the most egregious is the tendency for IE to report a problem at a line number whose source code is HTML with an event handler. The problem, it turns out, will be somewhere in the function being invoked by the event handler. Another possibility in all browsers is that the line number being reported is below the line that contains the problem. Consider the following simple source code listing (with line numbers from the source code editor) that intentionally contains a syntax error:

```
1:    <HTML>
2:    <HEAD>
3:    <SCRIPT LANGUAGE="JavaScript">
4:    function tarzan() {
5:        var x = 1
6:
7:    function jane() {
8:        var y = 3
9:    }
10:   </SCRIPT>
11:   </HEAD>
12:   <BODY>
13:   Hello.
14:   </BODY>
15:   </HTML>
```

When you load this page into browsers, all of them report a problem with a missing right brace (NN is a bit more explicit with its message, indicating that a right brace is missing after a function body). But where do the browsers point to the error? By looking at the code as a human, you can see that the missing brace belongs in Line 6. But now examine the code from the point of view of a script interpreter engine: It sees the opening brace on Line 4, and then a new function declaration starting on Line 7. To the interpreter, this means that the `jane()` function is probably nested inside the `tarzan()` function, and it is the `tarzan()` function that is lacking the right brace following the `jane()` function. Therefore, the error line number comes in at Line 10 (although IE5/Mac reports Line 9). Your scripts won’t likely be this simple, so the distance between the reported error line number and the location of the actual problem can be substantial and difficult to spot without using some of the tips and tools described later in this chapter.

IE sometimes has a nasty habit of identifying the location of the problem at Line 1, Character 1. All this means is that you need to put your detective skills to work that much harder. Common causes for this behavior are references to HTML objects that don't exist (or there is a mismatch between the identifier of the element and your script reference to it) and errors that affect global functions or window methods. To find the genuine problem line, you can use tracing techniques described later in this chapter.

Error message text

Because so many permutations exist of the potential errors you can make in scripts and the ways the JavaScript interpreters in different browsers regard these errors, presenting hard-and-fast solutions to every JavaScript error message is impossible. What I can do, however, is list the most common and head-scratch-inducing error messages and relate the kinds of non-obvious problems that can trigger such messages.

"Object expected"

This error message is often one of the least helpful that you see in IE. The line number associated with the message typically points to line in the source code that invokes a function. If you define event handlers as attributes of element tags, the line number being reported may correspond to the line containing that HTML tag.

The most obvious problem is that the function being invoked is not regarded as a valid function in the page (the "object" referred to here is the function object). This problem can be the result of an HTML or script error earlier in the document. The problem can also be the result of some error in the function itself that failed to let the interpreter treat the function as a genuine function object. Most typically, these kinds of problems are detected as syntax errors while the page loads (for example, an imbalanced set of parentheses or braces), but not always.

As a first-strike tactic, you need to determine if the function is being invoked at all. By placing an alert in the first line of the function and triggering the function, you can see if script execution is reaching that point. If that works okay, then move the alert downward through the function to find out where the error is actually being triggered. The line before the alert that fails is the likely culprit.

"Expected <something>"

This message usually points straight at the problem line. Most of the "things" that the statement expects are self-explanatory. If a right parenthesis is missing from a pair, that is the "thing" shown to be expected. Detecting in the message the difference between a brace and parenthesis isn't always easy, so look at the message carefully. Not quite as intuitive is when the message says "Expected identifier". This error refers to an expression that typically is trying to use a reserved word as a variable name. Look into Appendix B for a list of reserved words, none of which you may use as names of things or variables.

“<Something> is undefined”

This message is fairly easy to understand, yet at times difficult to diagnose. For variable names, the message usually means that you have an uninitialized variable name sitting in the position of a right-hand operand or a unary operand. This variable name has not been declared or assigned with any value prior to this erroneous statement. Perhaps you’re attempting to use a variable name that has been initialized only as a local variable in another function. You may also have intended the right-hand value to be a string, but you forgot to enclose it in quotes, forcing JavaScript to look upon it as a reference to something. Another possibility is that you misspelled the name of a previously declared variable. JavaScript rightly regards this item as a new, undeclared variable. Misspellings, you will recall, include errors in upper- and lowercase in the very case-sensitive JavaScript world.

If the item is a function name, you may have to perform a bit of detective work. Though the function may be defined properly, a problem in the script above the function (for example, imbalanced braces) makes JavaScript fail to see the function. In other cases, you may be trying to invoke a function in another window or frame but forgot to include the reference to that distant spot in the call to the function.

A less likely case, but a confusing one to diagnose, is when you are assembling string versions of function calls or array references out of literal strings and variable values. The following simplified example is assembling a string that is a function call to be triggered by `setTimeout()`:

```
function doA() {
    var x = "joe"
    setTimeout("doB(" + x + ")", 5000)
}
```

Even though the value of `x` is a string when it is concatenated to the call to the `doB()` function, the value gets evaluated as if it were a variable name. An error crops up saying that “joe is undefined”. Because you want to pass the value of `x` as a parameter, you must nest its value inside a pair of quotes, as follows:

```
function doA() {
    var x = "joe"
    setTimeout("doB('" + x + "')", 5000)
}
```

The difference in the code is extremely subtle, but absolutely necessary.

“<Something> is not a function”

As with the preceding one, this error message can be one of the most frustrating, because when you look at the script, it appears as though you have clearly defined a function by that name, and you’re simply having an event handler or other running statement call that function. The first problems to look for are mismatched case of letters in the calling statement and function and the reuse of a variable or HTML object name by the function name.

This latter item is a no-no — it confuses JavaScript into thinking that the function doesn't exist, even though the object name doesn't have parentheses appended to it and the function does. I've also seen this error appear when other problems existed in the script above the function named in the error message, and the named function was the last one in a script.

In NN, this message appears when you attempt to invoke a function that is not implemented for a particular object. For example, if you attempt to use a W3C DOM method in NN4, the error reports that the method you tried to invoke "is not a function."

"Object doesn't support this property or method"

This IE message reports that a valid object does not provide support for a method you just attempted to invoke. In practice, this message rarely appears as the result of referencing an object's nonexistent property, because the language allows for extending an object's list of properties by assignment. If you do a lot of development in IE5+ for Windows, you may see a lot of this message when testing the page in IE5 for the Macintosh, whose complement of implemented object methods is somewhat smaller.

"Unterminated string literal"

"Unterminated string constant"

NN is far more helpful with this type of message, because along with the error message, it displays the code fragment that tripped the error. You will see the beginning (or all) of the string that is the culprit. If you simply forgot to close a string quote pair, the error most frequently appears when you try to concatenate strings or nest quoted strings. Despite the claim that you can nest alternating double and single quotes, I often have difficulties using this nesting method beyond the second nested level (single quotes inside a double-quoted string). At different times, I've gotten away with using a pair of \" inline quote symbols for a third layer. If that syntax fails, I break up the string so that nesting goes no deeper than two layers. If necessary, I even back out the most nested string and assign it to a variable in the preceding line — concatenating it into the more complex string in the next line.

In the Windows 3.1 versions of Navigator, you may also see this error if a string value is longer than about 250 characters. But you can divide such a string into smaller segments and concatenate these strings later in the script with the add-by-value (+=) operator.

And in all versions of Navigator through NN4, avoid statements in scripts that extend for more than 255 characters. If you use a text editor that counts the column number as you type, use this measure as a guide for long statements. Break up long statements into shorter lines.

“Missing } after function body”

“Expected }”

This error usually is easy to recognize in a simple function definition because the closing brace is missing at the end of the function. But when the function includes additional nested items, such as `if...else` or `for` loop constructions, you begin dealing with multiple pairs of braces within the function. The JavaScript interpreter doesn’t always determine exactly where the missing brace belongs, and thus it simply defaults to the end of the function. This location is a natural choice, I guess, because from a global perspective of the function, one or more of the right braces that ripple down to the end of the function usually are missing.

In any case, this error message means that a brace is missing somewhere in a function above the referenced line number. Do an inventory count for left and right braces and see whether a discrepancy occurs in the counts. One of those nested constructions is probably missing a closing brace. Some programmer-oriented text editors also include tools for finding balanced pairs of braces and parentheses.

“<Something> is not a number”

The variable name singled out in this error message is most likely a string or null value. The line of JavaScript that trips it up has an operator that demands a number. When in doubt about the data type of a variable destined for a math operation, use the `parseInt()` or `parseFloat()` functions to convert strings to numbers.

I have also encountered this error when it provides no clue about what isn’t a number — the error message simply says, “is not a number.” The root of the problem ended up having nothing to do with numbers. A structural imbalance in the script triggered this bogus error message.

“<Something> has no property named . . .”

“<Something> has no properties”

When a statement trips this error message, an object reference has usually gone awry in an assignment or comparison expression. You probably attempted to reference a property of an object, but something is wrong with the object reference, or you’re trying to retrieve a property that doesn’t exist for that object. If the reference is an extended one, you may have to dig to find the precise problem with the reference. Consider the following two statements that attempt to access the `value` property of a button named `calcMe`:

```
document.forms.calcme.value
document.forms[0].calcme.value
```

The NN errors for these two statements would read “document.forms.calcme has no properties” and “document.forms[0].calcme has no properties”. Causes for the two errors are quite different. The obvious problem with them both may seem to be that the button’s name is incorrectly referenced as `calcme` instead of `calcMe`. That, indeed, is the error

for the second statement. But a more fundamental problem also plagues the first statement: the `document.forms` reference (a valid object, returning an array of forms) needs an array index in this instance, because it needs to look into a particular form for one of its objects. Unfortunately, both error messages look alike at first glance, and you cannot tell from them which statement has two errors and which has one.

But what you can do when this kind of error appears is use the reference that is returned with the error message to check your work. Start verifying the accuracy of your references from left to right. Later in this chapter, you see how to use the embeddable Evaluator tool to verify the existence of object references.

"<Something> is null or not an object"

This message is the IE version of the previous NN error message. A big difference is that the reference returned as part of the error message includes the complete reference. Therefore, a reference to a nonexistent calcme button in a form yields the error message "'document.forms[0].calcme.value' is null or not an object". Your first instinct is to be suspicious of the `value` property part of the reference. The detective work to find the problem is the same as in the NN version: verify the reference piece by piece, working from left to right. Again, the embeddable Evaluator can assist in this task.

"<Something> has no property indexed by [i]"

Look carefully at the object reference in this error message. The last item has an array index in the script, but the item is not an array value type. Users commonly make this mistake within the complex references necessary for radio buttons and SELECT options. Make sure that you know which items in those lengthy references are arrays and which are simply object names that don't require array values.

"<Something> can't be set by assignment"

This error message tells you either that the property shown is read-only or that the reference points to an object, which must be created via a constructor function rather than by simple assignment.

"Test for equality (==) mistyped as assignment (=)? Assuming equality test."

The first time I received this error, I was amazed by JavaScript's intelligence. I had, indeed, meant to use the equality comparison function (==) but had entered only a single equal sign. JavaScript is good at picking out these situations where Boolean values are required. In NN6, this message has been demoted to just a warning rather than an error.

"Function does not always return a value"

Often while designing deeply nested `if...else` constructions, your mind follows a single logic path to make sure that a particular series of conditions is met, and that the

function returns the desired values under those conditions. What is easy to overlook is that there may be cases in which the decision process may "fall through" all the way to the bottom without returning any value, at which point the function must indicate a value that it returns, even if it is a 0 or empty (but most likely a Boolean value). JavaScript checks the organization of functions to make sure that each condition has a value returned to the calling statement. The error message doesn't tell you where you're missing the return statement, so you have to do a bit of logic digging yourself.

"Access disallowed from scripts at <URL> to documents at <URL>"

"Access is denied"

These messages (NN and IE versions, respectively) indicate that a script in one frame or window is trying to access information in another frame or window that has been deemed a potential security threat. Such threats include any location object property or other information about the content of the other frame when the other frame's document comes from a protocol, server, or host that is different from the one serving up the document doing the fetching.

Even the best of intentions can be thwarted by these security restrictions. For example, you may be developing an application that blends data in cooperation with another site. Security restrictions, of course, don't know that you have a cooperative agreement with the other Web site, and you have no workaround for accessing a completely different domain unless you use signed scripts for NN (see Chapter 46) or an IE user has browser security levels set dangerously loose.

Another possible trigger for these errors is that you are using two different servers in the same domain or different protocols (for example, using `https:` for the secure part of your commerce site, while all catalog info uses the `http:` protocol). If the two sites have the same domain (for example, `giantco.com`) but different server names or protocols, you can set the `document.domain` properties of documents so that they recognize each other as equals. See Chapter 46 for details on these issues and the restrictions placed on scripts that mean well, but that can be used for evil purposes.

IE, especially Windows versions, frequently clamps down too severely on inter-window and inter-frame communication. Don't be surprised to encounter security problems trying to communicate between a main window and another window whose content is dynamically generated by scripts in the main window. This error can be incredibly frustrating. Sometimes, serving the main page from a server (instead of reading it from a local hard disk) can solve the problem, but not always. You are safest if the content of both windows or frames are HTML documents served from the same server and domain.

"Lengthy JavaScript still running. Continue?"

Although not a genuine error message, this NN3 alert dialog box provides a safeguard against inadvertent infinite loops and intentional ones triggered by JavaScript tricksters.

Instead of permanently locking up the browser, NN3 — after processing a large number of rapidly executing script cycles — asks the user whether the scripts should continue. This error was not adopted in later versions of NN or ever implemented in IE.

"Unspecified error"

This completely unhelpful IE error message is not a good sign because it means that whatever error is occurring is not part of the well-traveled decision tree that the browser uses to report errors. All is not lost, however. That the browser has not crashed means that you can still attempt to get at the root of the problem through various tracing tactics described later in this chapter.

"Uncaught exception"

You may encounter these messages in NN6, although usually not as a result of your scripts unless you are using some of the browser's facilities to dive into inner workings of the browser. These messages are triggered by the browser's own programming code, and indicate a processing error that was not properly wrapped inside error trapping mechanisms. The details associated with such an error point to NN6's own source code modules and internal routines. If you can repeat the error and can do so in a small test case page, you are encouraged to submit a report to `http://bugzilla.mozilla.org`, the bug tracking site for the engine inside NN6.

"Too many JavaScript errors"

You may see this message in NN if it detects a runaway train generating errors uncontrollably. This message was far more important in the days of separate error windows, because a buggy repeat loop could cause NN to generate more error windows than it could do safely.

Sniffing Out Problems

It doesn't take many error-tracking sessions to get you in the save-switch-reload mode quickly. Assuming that you know this routine (described in Chapter 3), the following are some techniques I use to find errors in my scripts when the error messages aren't being helpful in directing me right to the problem.

Check the HTML tags

Before I look into the JavaScript code, I review the document carefully to make sure that I've written all my HTML tags properly. That includes making sure that all tags have closing angle brackets and that all tag pairs have balanced opening and closing tags. Digging deeper, especially in tags near the beginning of scripts, I make sure that all tag attributes that must be enclosed in quotes have the quote pairs in place. A browser may be forgiving about sloppy HTML as far as layout goes, but the JavaScript interpreter isn't as

accommodating. Finally, I ensure that the <SCRIPT> tag pairs are in place (they may be in multiple locations throughout my document) and that the LANGUAGE="JavaScript" attribute value has both of its quotes.

View the source

Your success in locating bugs by viewing the source in the browser varies widely with the kind of content on the page and the browser you use. Very frequently, authors place perhaps too much importance to what they see in the source window.

For a straight, no-frame HTML page, viewing the source provides a modicum of comfort by letting you know that the entire page has arrived from the server. Some versions of NN might flash a questionable HTML construction, but don't expect miracles.

Note: NN4 exhibits a notorious bug in the source view if your HTML tags include STYLE attributes for element-specific style sheets. You may "see double" in these lines, whereby the STYLE attribute appears to be repeated (although usually beginning with "TTYLE...") in what looks to be gibberish. This problem is a bug in the source viewer and does not accurately represent what the browser-rendering engine is using as source code.

Examining the source code for framesetting documents or individual frames, you must first give focus to the desired element. For an individual frame, click in the frame, and then right-click (or click and hold on the Mac) on the frame's background to get the contextual menu. One of the items should indicate a source view of the frame. To view the framesetter's source, press the Tab key until the Address/Location field of the browser is selected. Then choose to view the source from the Edit menu.

Where the source view would be most helpful, but often fails, is to display dynamically generated HTML. Your best chance will be for pages whose entire content is generated by script. This is about the only place you can appreciate the difference between document.write() and document.writeln(), because the latter puts carriage returns after the end of each string passed as a parameter to the method. The result is a more readable source view. Most recent browsers, with the exception of NN6, display the HTML as written by your script. NN4 does this in a window whose URL indicates the wysiwyg: protocol — an internal indication of dynamically generated content.

But when only part of the page is generated by script, few browsers combine the hard-wired and dynamic code in the source view. Instead, you see only the hard-wired HTML and scripts. To work around this for IE4+ and NN6, you can use the embeddable Evaluator and read the innerHTML property of any elements you want.

Intermittent scripts

Without question, the most common bug in Navigator 2.0*x* is the one that makes scripts behave intermittently. Buttons, for example, won't fire onClick event handlers unless the page is reloaded. Or, as a result of the same bug, sometimes a script runs and

sometimes it doesn't. The problem here is that NN2 requires all <IMG> tags to include HEIGHT and WIDTH attributes, even when the images are not scripted. Because doing so is good HTML practice anyway (it helps the browser's layout performance and is technically required according to the formal HTML specification), if you include these attributes without fail throughout your HTML documents, you won't be plagued by intermittent behavior.

Scripts not working in tables

Tables have been a problem for scripts through NN3. The browser has difficulty when a <SCRIPT> tag is included inside a <TD> tag pair for a table cell. The workaround for this is to put the <SCRIPT> tag outside the table cell tag and use document.write() to generate the <TD> tag and its contents. I usually go one step further, and use document.write() to generate the entire table's HTML. This step is necessary only when executable statements are needed in cells (for example, to generate content for the cell). If a cell contains a form element whose event handler calls a function, you don't have to worry about this problem.

Timing problems

One problem category that is very difficult to diagnose is the so-called timing problem. There are no hard-and-fast rules that govern when you are going to experience such a problem. Very experienced scripters develop an instinct about when timing is causing a problem that has no other explanation.

A timing problem usually means that one or more script statements are executing before the complete action of an earlier statement has finished its task. JavaScript runs within a single thread inside the browser, meaning that only one statement can run at a time. But there are times when a statement invokes some browser service that operates in its own thread and therefore doesn't necessarily finish before the next JavaScript statement runs. If the next JavaScript statement relies on the previous statement's entire task having been completed, the script statement appears to fail, even though it actually runs as it should.

These problems crop up especially when scripts work with another browser window, and especially in IE for Windows (ironic in a way). In discussions in this book about form field validation, for example, I recommend that after an instructive alert dialog box notifies the user of the problem with the form, the affected text field should be given focus and its content selected. In IE/Windows, however, after the user closes the alert dialog box, the script statements that focus and select operate before the operating system has finished putting the alert away and refreshing the screen. The result is that the focused and selected text box loses its focus by the time the alert has finally disappeared.

The solution is to artificially slow down the statements that perform the focus and select operations. By placing these statements in a separate function, and invoking this function via the window.setTimeout() method, the browser catches its breath before

executing the separate function — even when the delay parameter is set to zero. A similar delay is utilized when opening and writing to a new window, as shown in the example for `window.open()` in Chapter 16.

Reopen the file

If I make changes to the document that I truly believe should fix a problem, but the same problem persists after a reload, I reopen the file via the File menu. Sometimes, when you run an error-filled script more than once, the browser's internals get a bit confused. Reloading does not clear the bad stuff, although sometimes an unconditional reload (clicking Reload while holding Shift) does the job. Reopening the file, however, clears the old file entirely from the browser's memory and loads the most recently fixed version of the source file. I find this situation to be especially true with documents involving multiple frames and tables and those that load external `.js` script library files. In severe cases, you may even have to restart the browser to clear its cobwebs, but this is less necessary in recent browsers. You should also consider turning off the cache in your development browser(s).

Find out what works

When an error message supplies little or no clue about the true location of a runtime problem, or when you're faced with crashes at an unknown point (even during document loading), you need to figure out which part of the script execution works properly.

If you have added a lot of scripting to the page without doing much testing, I suggest removing (or commenting out) all scripts except the one(s) that may get called by the document's `onLoad` event handler. This is primarily to make sure that the HTML is not way out of whack. Browsers tend to be quite lenient with bad HTML, so that this tactic won't necessarily tell the whole story. Next, add back the scripts in batches. Eventually, you want to find where the problem really is, regardless of the line number indicated by the error message alert.

To narrow down the problem spot, insert one or more alert dialog boxes into the script with a unique, brief message that you will recognize as reaching various stages (such as `alert("HERE-1")`). Start placing alert dialog boxes at the beginning of any groups of statements that execute and try the script again. Keep moving these dialog boxes deeper into the script (perhaps into other functions called by outer statements) until the error or crash occurs. You now know where to look for problems. See also an advanced tracing mechanism described later in this chapter.

Comment out statements

If the errors appear to be syntactical (as opposed to errors of evaluation), the error message may point to a code fragment several lines away from the problem. More often than not, the problem exists in a line somewhere above the one quoted in the error

message. To find the offending line, begin commenting out lines one at a time (between reloading tests), starting with the line indicated in the error message. Keep doing this until the error message clears the area you're working on and points to some other problem below the original line (with the lines commented out, some value is likely to fail below). The most recent line you commented out is the one that has the beginning of your problem. Start looking there.

Check runtime expression evaluation

I've said many times throughout this book that one of the two most common problems scripters face is an expression that evaluates to something you don't expect (the other common problem is an incorrect object reference). In lieu of a debugger that would let you step through scripts one statement at a time while watching the values of variables and expressions, you have a few alternatives to displaying expression values while a script runs.

The simplest approaches to implement are an alert box and the statusbar. Both the alert dialog box and statusbar show you almost any kind of value, even if it is not a string or number. An alert dialog box can even display multiple-line values.

Because most expression evaluation problems come within function definitions, start such explorations from the top of the function. Every time you assign an object property to a variable or invoke a string, math, or date method, insert a line below that line with an `alert()` method or `window.status` assignment statement (`window.status = someValue`) that shows the contents of the new variable value. Do this one statement at a time, save, switch, and reload. Study the value that appears in the output device of choice to see if it's what you expect. If not, something is amiss in the previous line involving the expression(s) you used to achieve that value.

This process is excruciatingly tedious for debugging a long function, but it's absolutely essential for tracking down where a bum object reference or expression evaluation is tripping up your script. When a value comes back as being undefined or null, more than likely the problem is an object reference that is incomplete (for example, trying to access a frame without the `parent.frames[i]` reference), using the wrong name for an existing object (check case), or accessing a property or method that doesn't exist for that object.

When you need to check the value of an expression through a long sequence of script statements or over the lifetime of a repeat loop's execution, you are better off with a listing of values along the way. In the section "A Simple Trace Utility" later in this chapter, I show you how to capture trails of values through a script.

Using the embeddable Evaluator

As soon as a page loads or after some scripts run, the window contains objects whose properties very likely reveal a lot about the environment at rest (that is, not while scripts

are running). Those values are normally disguised from you, and the only way to guarantee successful access to view those values is through scripting within the same window or frame. That's where the embeddable Evaluator comes in handy.

As you probably recall from Chapter 13 and the many example sections of Parts III and IV of this book, the code within the standalone Evaluator provides two text boxes for entry of expressions (in the top box) and object references (the bottom box). Results of expression evaluation and object property dumps are displayed in the Results textarea between the two input boxes. A compact version of The Evaluator is contained by a separate library version called `evaluator.js` (located in the Chapter 45 folder of listings on the companion CD-ROM). As you embark on any substantial page development project, you should link in the library with the following tag at the top of your HEAD section:

```
<SCRIPT LANGUAGE="JavaScript" SRC="evaluator.js"></SCRIPT>
```

Be sure to either have a copy of the `evaluator.js` file in the same directory as the document under construction or specify a complete file: URL to the library file on your hard drive for the SRC attribute.

Immediately above the closing BODY tag of your document, include the following:

```
<SCRIPT LANGUAGE="JavaScript">
printEvaluator()
</SCRIPT>
```

The `printEvaluator()` function draws a horizontal rule (HR) followed by the complete control panel of The Evaluator, including the codebase principle support for NN4+. From this control panel, you can reference any document object supported by the browser's object model or global variable. You can even invoke functions located in other scripts of the page by entering them into the top text box. Whatever references are available to other scripts on the page are also available to The Evaluator, including references to other frames of a frameset and even other windows (provided a reference to the newly opened window has been preserved as a global variable, as recommended in Chapter 16).

If you are debugging a page on multiple browsers, you can switch between the browsers and enter property references into The Evaluator on each browser and make sure all browsers return the same values. Or, you can verify that a DOM object and property are available on all browsers under test. If you are working on W3C DOM compatible browsers, invoke the `walkChildNodes()` function of The Evaluator to make sure that modifications your scripts make to the node tree are achieving the desired goals. Experiment with direct manipulation of the page's objects and node tree by invoking DOM methods as you watch the results on the page.

You should be aware of only two small cautions when you embed The Evaluator into the page. First, The Evaluator declares its own one-letter lowercase global variable names (`a` through `z`) for use in experiments. Your own code should therefore avoid one-letter global variables (but local variables, such as the `i` counter of a `for` loop, are fine

provided they are initialized inside a function with a `var` keyword). Second, while embedding The Evaluator at the bottom of the page should have the least impact on the rest of your HTML and scripts, any scripts that rely on the length of the `document.forms` array will end up including the form that is part of The Evaluator. You can always quickly turn off The Evaluator by commenting out the `printEvaluator()` statement in the bottom script to test your page on its own.

The embeddable Evaluator is without question the most valuable and frequently used debugging tool in my personal arsenal. It provides x-ray vision into the object model of the page at any resting point.

Emergency evaluation

Using The Evaluator assumes that you thought ahead of time that you want to view property values of a page. But what if you haven't yet embedded The Evaluator, and you encounter a state that you'd like to check out without disturbing the currently loaded page?

To the rescue comes the `javascript:` URL and the Location/Address box in your browser's toolbar. By invoking the `alert()` method through this URL, you can view the value of any property. For example, to find out the content of the cookie for the current document, enter the following into the Location/Address box in the browser:

```
javascript: alert(document.cookie)
```

Object methods or script functions can also be invoked this way, but you must be careful to prevent return values from causing the current page to be eliminated. If the method or function is known to return a value, you can display that value in an alert dialog box. The syntax for a function call is:

```
javascript:alert(myFunction("myParam1"))
```

And if you want to invoke a function that does not necessarily return a value, you should also protect the current page by using the void operator, as follows:

```
javascript:void myFunction("myParam1")
```

A Simple Trace Utility

Single-stepping through running code with a JavaScript debugger is a valuable aid when you know where the problem is. But when the bug location eludes you, especially in a complex script, you may find it more efficient to follow a rapid trace of execution and viewing intermediate values along the way. The kinds of questions that this debugging technique addresses include:

* How many times is that loop executing?

* What are the values being retrieved each time through the loop?

* Why won't the while loop exit?

* Are comparison operators behaving as I'd planned in `if...else` constructions?

* What kind of value is a function returning?

A bonus feature of the embeddable Evaluator is a simple trace utility that lets you control where in your running code values can be recorded for viewing after the scripts run. The resulting report you get after running your script can answer questions like these and many more.

The trace() function

Listing 45-1 shows the `trace()` function that is built into the `evaluator.js` library file. By embedding the Evaluator into your page under construction, you can invoke the `trace()` function wherever you want to capture an interim value.

Listing 45-1
trace() function

```
function trace(flag, label, value) {
    if (flag) {
        var msg = ""
           if (trace.caller) {
            var funcName = trace.caller.toString()
            funcName = funcName.substring(9, funcName.indexOf(")") + 1)
            msg += "In " + funcName + ": "
        }
        msg += label + "=" + value + "\n"
        document.forms["ev_evaluator"].ev_output.value += msg
    }
}
```

The `trace()` function takes three parameters. The first, `flag`, is a Boolean value that determines whether the trace should proceed (I show you a shortcut for setting this flag later). The second parameter is a string used as a plain-language way for you to identify the value being traced. The value to be displayed is passed as the third parameter. Virtually any type of value or expression can be passed as the third parameter — which is precisely what you want in a debugging aid.

Only if the flag parameter is `true` does the bulk of the `trace()` function execute. The first task is to extract the name of the function from which the `trace()` function was called. Unfortunately, the `caller` property of a function is missing from NN6 (and ECMAScript), so this information is made part of the result only if the browser running the trace supports the property. By retrieving the rarely used caller property of a function, the script grabs a string copy of the entire function that has just called `trace()`. A quick extraction of a substring from the first line yields the name of the function. That information becomes part of the message text that records each trace. The message identifies the calling function followed by a colon; after that comes the label text passed as the second parameter plus an equals sign and the value parameter. The format of the output message adheres to the following syntax:

```
In <funcName>: <label>=<value>
```

The results of the trace — one line of output per invocation — are appended to the Results textarea in The Evaluator. It's a good idea to clear the textarea before running a script that has calls to `trace()` so that you can get a clean listing.

Preparing documents for trace.js

As you build your document and its scripts, you need to decide how granular you want tracing to be: global or function-by-function. This decision affects at what level you place the Boolean "switch" that turns tracing on and off.

You can place one such switch as the first statement in the first script of the page. For example, specify a clearly named variable and assign either false or zero to it so that its initial setting is off:

```
var TRACE = 0
```

To turn debugging on at a later time, simply edit the value assigned to TRACE from zero to one:

```
var TRACE = 1
```

Be sure to reload the page each time you edit this global value.

Alternatively, you can define a local TRACE Boolean variable in each function for which you intend to employ tracing. One advantage of using function-specific tracing is that the list of items to appear in the Results textarea will be limited to those of immediate interest to you, rather than all tracing calls throughout the document. You can turn each function's tracing facility on and off by editing the values assigned to the local TRACE variables.

Invoking trace()

All that's left now is to insert the one-line calls to `trace()` according to the following syntax:

```
trace(TRACE,<"label">,<value>)
```

By passing the current value of ~~TRACE~~ as a parameter, you let the library function handle the decision to accumulate and display the trace. The impact on your running code is kept to a one-line statement that is easy to remember. To demonstrate how to make the calls to `trace()`, Listing 45-2 shows a pair of related functions that convert a time in milliseconds to the string format "hh:mm". To help verify that values are being massaged correctly, the script inserts a few calls to `trace()`.

Listing 45-2
Calling trace()

```
function timeToString(input) {
   var TRACE = 1
   trace(TRACE,"input",input)
   var rawTime = new Date(eval(input))
   trace(TRACE,"rawTime",rawTime)
```

```
   var hrs = twoDigitString(rawTime.getHours())
   var mins = twoDigitString(rawTime.getMinutes())
   trace(TRACE,"result", hrs + ":" + mins)
   return hrs + ":" + mins
}

function twoDigitString(val) {
   var TRACE = 1
   trace(TRACE,"val",val)
   return (val < 10) ? "0" + val : "" + val
}
```

After running the script, the Results box in The Evaluator shows the following trace:

```
In  timeToString(input): input=976767964655
In  timeToString(input): rawTime=Wed Dec 13 20:26:04 GMT-0800 2000
In  twoDigitString(val): val=20
In  twoDigitString(val): val=26
In  timeToString(input): result=20:26
```

Having the name of the function in the trace is helpful in cases in which you might justifiably reuse variable names (for example, `i` loop counters). You can also see more clearly when one function in your script calls another.

One of the most valuable applications of the `trace()` function comes when your scripts accumulate HTML that gets written to other windows or frames, or replaces HTML segments of the current page. Because the source view may not display the precise HTML that you assembled, you can output it via the `trace()` function to the Results box. From there, you can copy the HTML and paste it into a fresh document to test in the browser by itself. You can also verify that the HTML content is being formatted the way that you want it.

Browser Crashes

Each new browser generation is less crash-prone than its predecessor, which is obviously good news for everyone. It seems that most crashes, if they occur, do so as the page loads. This can be the result of some ill-advised HTML, or something happening as the result of script statements that either run immediately as the page loads or in response to the `onLoad` event handler.

Finding the root of a crash problem is perhaps more time consuming because you must relaunch the browser each time (and in some cases, even reboot your computer). But the basic tactics are the same. Reduce the page’s content to the barest minimum HTML by commenting out both scripts and all but HEAD and BODY tags. Then begin enabling small chunks to test reloading of the page. Be suspicious of META tags inserted by authoring tools. Their removal can sometimes clear up all crash problems. Eventually you will add something into the mix that will cause the crash. It means that you are close to finding the culprit.

Preventing Problems

Even with help of authoring and debugging tools, you probably want to avoid errors in the first place. I offer a number of suggestions that can help in this regard.

Getting structure right

Early problems in developing a page with scripts tend to be structural: knowing that your objects are displayed correctly on the page; making sure that your <SCRIPT> tags are complete; completing brace, parenthesis, and quoted pairs. I start writing my page by first getting down the HTML parts — including all form definitions. Because so much of a scripted page tends to rely on the placement and naming of interface elements, you will find it much easier to work with these items after you lay them out on the page. At that point, you can start filling in the JavaScript.

When you begin defining a function, repeat loop, or ~~if~~ construction, fill out the entire structure before entering any details. For example, when I define a function named `verifyData()`, I enter the entire structure for it:

```
function verifyData() {

}
```

I leave a blank line between the beginning of the function and the closing brace in anticipation of entering at least one line of code.

After I decide on a parameter to be passed and assign a variable to it, I may want to insert an `if` construction. Again, I fill in the basic structure:

```
function verifyData(form) {
    if (form.checkbox.checked) {

    }
}
```

This method automatically pushes the closing brace of the function lower, which is what I want — putting it securely at the end of the function where it belongs. It also ensures that I line up the closing brace of the `if` statement with that grouping. Additional statements in the `if` construction push down the two closing braces.

If you don't like typing or don't trust yourself to maintain this kind of discipline when you're in a hurry to test an idea, you should prepare a separate document that has templates for the common constructions: <SCRIPT> tags, function, `if`, `if...else`, `for` loop, `while` loop, and conditional expressions. Then if your editor and operating system support it, drag and drop the necessary segments into your working script.

Build incrementally

The worst development tactic you can follow is to write tons of code before trying any of it. Error messages may point to so many lines away from the source of the problem that it

could take hours to find the true source of difficulty. The save-switch-reload sequence is not painful, so the better strategy is to try your code every time you have written a complete thought — or even enough to test an intermediate result in an alert dialog box — to make sure that you're on the right track.

Test expression evaluation

Especially while you are learning the ins and outs of JavaScript, you may feel unsure about the results that a particular string, math, or date method yields on a value. The longer your scripted document gets, the more difficult it will be to test the evaluation of a statement. You're better off trying the expression in a more controlled, isolated environment, such as The Evaluator. By doing this kind of testing in the browser, you save a great deal of time experimenting by going back and forth between the source document and the browser.

Build function workbenches

A similar situation exists for building and testing functions, especially generalizable ones. Rather than test a function inside a complex scripted document, drop it into a skeletal document that contains the minimum number of user interface elements that you need to test the function. This task gets difficult when the function is closely tied to numerous objects in the real document, but it works wonders for making you think about generalizing functions for possible use in the future. Display the output of the function in a text or textarea object or include it in an alert dialog box.

Testing Your Masterpiece

If your background strictly involves designing HTML pages, you probably think of testing as determining your user's ability to navigate successfully around your site. But a JavaScript-enhanced page — especially if the user enters input into fields or implements Dynamic HTML techniques — requires a substantially greater amount of testing before you unleash it to the online masses.

A large part of good programming is anticipating what a user can do at any point and then being sure that your code covers that eventuality. With multiframe windows, for example, you need to see how unexpected reloading of a document affects the relationships between all the frames — especially if they depend on each other. Users will be able to click Reload at any time or suspend document loading in the middle of a download from the server. How do these activities affect your scripting? Do they cause script errors based on your current script organization?

The minute you enable a user to type an entry into a form, you also invite the user to enter the wrong kind of information into that form. If your script expects only a numeric value from a field, and the user (accidentally or intentionally) types a letter, is your script ready to handle that "bad" data? Or no data? Or a negative floating-point number?

Just because you, as author of the page, know the “proper” sequence to follow and the “right” kind of data to enter into forms, your users will not necessarily follow your instructions. In days gone by, such mistakes were relegated to “user error.” Today, with an increasingly consumer-oriented Web audience, any such faults rest solely on the programmer — you.

If I sound as though I’m trying to scare you, I have succeeded. I was serious in the early chapters of this book when I said that writing JavaScript is *programming*. Users of your pages are expecting the same polish and smooth operation (no script errors and certainly no crashes) from your site as from the most professional software publisher on the planet. Don’t let them or yourself down. Test your pages extensively on as many browsers and as many operating systems as you can and with as wide an audience as possible before putting the pages on the server for all to see.