

# Debugging Scripts

---

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 now available for the latest browsers. Because the debuggers must work so closely with the internal workings of the browser, the tools so far have come from the browser makers themselves. While this chapter shows how Netscape's script debugger works, most of the discussion is of value even if you debug your scripts the "old-fashioned" way — by understanding the error messages.

## 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 around 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

# 45

CHAPTER



## In This Chapter

Identifying the type of error plaguing a script

Interpreting error messages

Preventing problems before they occur



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 Alerts

Navigator produces a large error dialog box whenever it detects even the slightest problem (Figure 45-1). At the top of the box is the URL of the document causing the problem and a line number. Below that is a (somewhat) plain-language description of the nature of the problem, followed by an extract of the code on which JavaScript is choking.

The line number provided in the error message is a valiant effort to help draw your attention to the problem line of code. The line number, however, may not accurately point to the true source of the problem.

Normally, the first line of such a count is the opening `<SCRIPT>` tag line in the HTML document. If you have more than one set of `<SCRIPT>` tags in your document, the line number is counted from the opening `<SCRIPT>` tag of the group that contains the line with the error. The capability of Navigator to focus on the truly erroneous line has improved substantially with each new release, but you cannot rely on the line number accurately pointing to the problem. This is especially true if your page loads an external .js script library file.

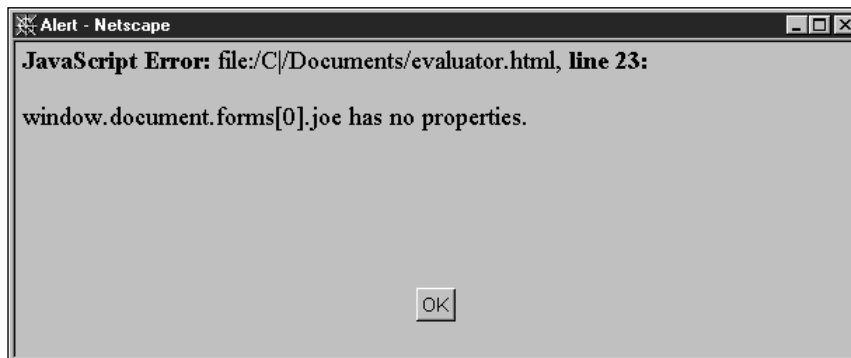


Figure 45-1: A typical error message window

More revealing at times is the code extract in the error dialog. For some syntax errors (such as missing halves of brace, parenthesis, and quote pairs), the extract comes from the line of code truly affected by the problem. The extract won't help at all, unfortunately, on some missing items, such as a missing right brace in a function definition. At best, the error message "missing } after function body" lets you know that an imbalance exists in one of your function definitions. The blank line of code it shows you isn't of much help, and the line number may be off by several lines.

While I'm on the subject of a missing closing brace, worse still is that the missing item doesn't trouble JavaScript until it encounters something later in the code that causes it to scratch its head. For example, if you have two function definitions and forget to insert the closing brace for the first definition, the error message shows as a plain "syntax error" pointing to the completely error-free second function. In truth, the problem is in the function immediately above that, but the JavaScript error detection can't figure that out.

In Navigator 3 and later, whenever a function, property, or method is causing a problem, the error message often includes clues about where in the object model the reference is being directed. For example, if a script attempts to retrieve a color value from a button form input element (a button named “hello”), the error message says

```
Window.Document.Form.Input.hello has no property named 'color'.
```

You can see from this message precisely which object is being examined by the script, and that information should send you to Appendix B of this book to see what properties exist for a button. This error message assistance is especially valuable when working with multiple-frame environments.

## Multiple error message windows

It is not uncommon for multiple error windows to appear when a script trips over a mistake. 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 error message that is displayed once the script stops running. That, however, is usually the error message least likely to lead you to the true problem. Error messages display their windows in the time order in which an error is found. This means that the first window to appear — the bottom error window of the stack of error windows — is the most important error message of them all. More than likely, the error in that window points to a problem that throws off the rest of the script, thus triggering all of the other error message windows that cover the original blooper.

When you encounter multiple windows (you have to watch, because each window is positioned exactly atop the one underneath it), you should look briefly at the message of each window as you work your way down to the first one. To know whether a window is the last one or has others underneath it, drag the error message window's title bar to show what, if anything, is underneath it. Some of those extra error messages may correctly point out an error, so examine them briefly to see if something obvious (for example, a missing close parenthesis) caused the window to show up. But don't start any serious debugging until you get down to the first error message. You must tackle this one before any others. And the solution may cause the other errors to go away.

## Error messages

Because so many permutations exist of the potential errors you can make in scripts and the ways the JavaScript interpreter regards 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 nonobvious problems that can trigger such messages.

### *“Something is not defined”*

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 word to be a string, but you forgot to enclose it in quotes. 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.

Users of Internet Explorer 3 will see this message referencing an image object. This is a case in which the browser does not support that kind of object, and thus any reference to a nonexistent object yields an error. The more each browser brand diverges from the other with its document object model, the more likely these kinds of messages will appear if you implement browser-specific features for deployment across all browsers.

### ***"Something is not a function"***

Like 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.

### ***"Unterminated string literal"***

In the code fragment displayed in the error window, the pointer usually appears at the first quote of a string it thinks is unterminated. 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. You can see an example of this method in bonus application Chapter 50 on the CD-ROM.

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, 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”**

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 the function, although not necessarily at the end. 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 value, a null value, or no value at all (the variable has been declared with the `var` statement, but not initialized with any 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 . . .”**

When a statement trips this error message, an object reference has usually gone awry. You've probably attempted to reference a property of a JavaScript object, but something is wrong with the object reference, or you're trying to retrieve a property that doesn't exist for that object. The error often has to do with the kinds of objects stored as arrays inside the browser (forms or links). If you're trying to retrieve the value property of a button named `calcMe`, for example, the following incorrect reference triggers the “has no property named” error:

```
document.forms.calcMe.value
```

The error here would read “Window.Document.FormArray.calcMe has no property named 'value'.” Notice that the reference in the error message includes something called `FormArray` (without any array index value in square brackets). You cannot have an entire array in a reference, just a single form. What this error message is very subtly telling you is that you forgot to single out a specific form:

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

JavaScript has many of these kinds of references that include arrays (radio buttons in a radio object, options in a select object, and so on). Always look very closely at the error message to see if a reference erroneously includes an entire array rather than just one of its elements.

***“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.

***“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 will have to do a bit of logic digging yourself.

***“Access disallowed from scripts at URL to documents at URL”***

Cross-domain security, which was beefed up during the reign of Navigator 2.0x, got even tighter in Navigator 3. This message indicates that a script in one frame is trying to access information in another frame 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 might 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 there is no workaround for accessing a completely different domain unless you use signed scripts (see Chapter 40).

Another possibility 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 40 for details on these issues and the restrictions placed on scripts that mean well, but that could be used for evil purposes.

### **“Lengthy JavaScript still running. Continue?”**

Although not a genuine error message (it appears in a JavaScript confirm dialog box), this alert dialog box provides a safeguard against inadvertent infinite loops and intentional ones triggered by JavaScript tricksters. Instead of permanently locking up the browser, Navigator — after processing a large number of rapidly executing script cycles — asks the user whether the scripts should continue.

### **“Syntax error”**

JavaScript is best at detecting true syntax errors and showing you the location of the problem. Even if the line counter is off, the chance that the error dialog box accurately shows the problem-causing code fragment and pointer is still good.

The errors you’ve seen here aren’t the only error messages that you will encounter in your scripts. Other error messages are pretty smart (for example, a message alerting you that you have only one equal sign when you meant to use two for a conditional test on the equality of two values). The real headaches occur when the error message and your code don’t seem to mesh.

### **“Too many JavaScript errors”**

You see this after several error message windows appear on the screen. Navigator limits the number of error windows it opens before getting out of control. This error usually crops up when the error appears inside a repeat loop and generates more error windows than Navigator allows. Close error windows to work your way back to the first window, and start your bug hunt with that window’s message.

## **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 has both of its quotes.

### **View the source**

Before Navigator 3, debugging HTML that was generated by script (via `document.write()` statements) was difficult. Scripted HTML may be embedded

amid regular HTML, or it may make up an entire page itself, usually inside another frame or window. Navigator 3 and later, however, simplify access to scripts in frames — you just click in a frame to select it and then choose Frame Source from the View menu or (if available) right-click the pop-up menu. The displayed result includes any HTML that the script generates. You can see how your expressions evaluate on their way to the page you see in the regular browser window.

This feature, incidentally, is what enables JavaScript-written HTML to be printed, previewed, and saved. Notice the special protocol that such frames have: `wysiwyg://`.

This statement is an internal protocol to the Navigator memory cache. Whenever you see the `wysiwyg://` protocol in a source window title bar, you know that some or all of the page was created with `document.write()` statements.

## Intermittent scripts

Without question, the most common bug in Navigator 2.0x 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 Navigator 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 you will certainly have visitors running Navigator 2 versions for some time to come, 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 Navigator 3. 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 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.

## 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.



## Find out what works

When an error message gives you 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 all scripts except the one(s) that might 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 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 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 best tool for checking evaluation expression is the JavaScript Debugger. See "Using the JavaScript Debugger" later for details.

In lieu of using the debugger, 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 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 to track 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 would be better off with a listing of values along the way. In "Writing Your Own Trace Utility" later in this chapter, I show you how to use Navigator's Java Console window and LiveConnect to provide you with quick trails of values through a script.

## Check object references and properties

Another tool to keep in your back pocket is the object property inspector. Using the special `for...in` loop construction of JavaScript (Chapter 31), you can call the function in Listing 45-1 from anywhere within your scripts to view the values of all properties of an object. You may find, for instance, that your script inadvertently changes the property of an object when you aren't looking. I also find that with more complex object relationships in newer browsers (for example, nested layers and their parent objects), it is not inconceivable to have a reference to the wrong object assigned to a variable. By showing the properties of the object, you can see exactly what object your script is pointing to. The function in Listing 45-1 is designed to accommodate objects that have dozens of properties by showing 25 at a time in an alert dialog. Pass as parameters to this function both the actual object (as an unquoted name) and the name of the object (as a quoted string) so that the resulting alert dialog box explains what's what.

### Listing 45-1: Property Inspector Function

```
function showProps(obj,objName) {
    var result = ""
    var count = 0
    for (var i in obj) {
        result += objName + "." + i + " = " + obj[i] + "\n"
        count++
        if (count == 25) {
            alert(result)
            result = ""
            count = 0
        }
    }
    alert(result)
}
```

```
}
```

## Using the JavaScript Debugger

Still in preview release form as of this writing, the JavaScript Debugger from Netscape is a byproduct of the Visual JavaScript tool (see Chapter 46). The debugger is a Java applet that lives in a separate window. With the debugger on, you can single-step through JavaScript code execution and observe the values of variables or expressions with each step. You can even modify the value of a variable or form element on the fly to test other outcomes. I provide here an overview of how to use the debugger in your authoring.

### Installing the debugger

You can download the JavaScript Debugger from Netscape's DevEdge Web site (<http://developer.netscape.com>). The precise URL may change over time, so I suggest you start at the DevEdge home page and look for developer downloads (the debugger may also be listed on the home page). Visit the site with Navigator 4 so that the installation process can properly connect the debugger to your copy of Navigator. The debugger is a signed applet and will ask for your permission to modify the browser. Grant that permission to complete the installation.

The debugger installs in a directory inside the Navigator/Communicator Program directory. While all of the code is stashed in a .jar file, the access to it is via an HTML file, called `jsdebugger.html`.

### Starting the debugger

To get the debugger going, open the `jsdebugger.html` file in Navigator. I highly recommend adding a bookmark to this file so you can start it up easily during debugging sessions. The file generates a small browser window, which is the window that loads the applet. This frees the main browser window for your own pages. If you close this small window, you quit the debugger.

It takes several seconds for the debugger applet to load, but while it is doing that, you can navigate in the main browser window to the HTML page you want to debug. When the debugger is loaded, its default window appears on screen (see Figure 45-2). If you are using Windows with a taskbar, the JavaScript Debugger appears as a taskbar item with a Java icon (and the small HTML stub window is also shown in the taskbar).

### Debugger layout

The main window of the debugger contains three subwindows. The large one at the top is for the current document source code. If you start the debugger normally, this area will be blank until you either open a document or let the debugger interrupt code execution (described in a moment).

At the lower left is a call stack, which lists the current line number and function name when you are stepping through code. If a function calls another function, the stack list starts to grow. The function currently being stepped through is at the top of

the stack; immediately below it is the name of the function that called the current function, along with the line number where the call to the current function was made.

The lower-right window (Console) has two elements in it. At the top is where you can view the values of variables or expressions as a script steps through statements. The one-line text field at the bottom is where you can type an expression to see what its value is. You can also use this field to assign a new value to an existing variable or object property.

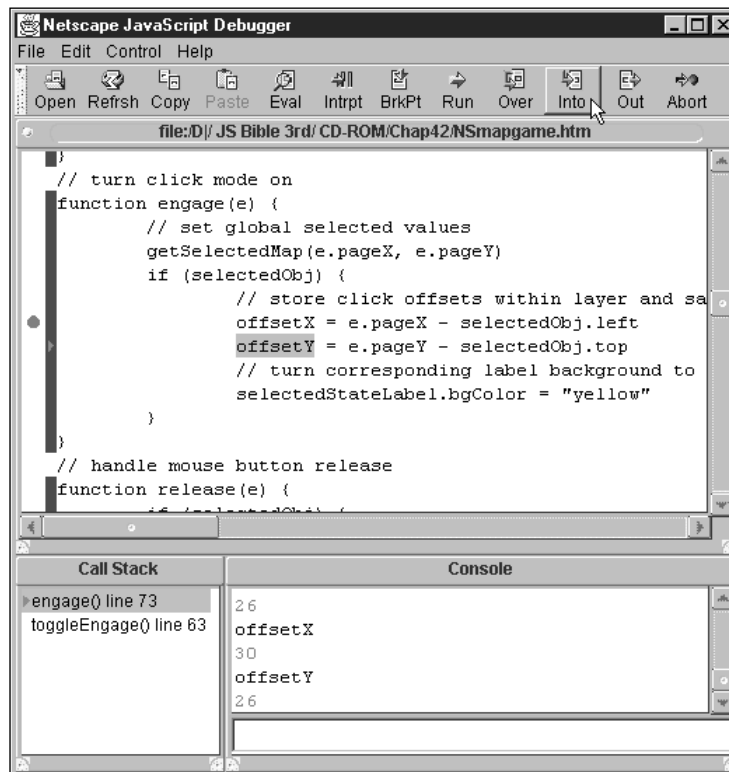


Figure 45-2: The JavaScript Debugger window

All windows are resizable, although you have to do a bit of manual work to get it done. To enlarge the work area, drag an edge or corner of the entire debugger window. This opens up blank space in the window. Then you can drag the individual window title bars to move the windows and/or drag the lower-right corners to resize the windows. In the prerelease version, the windows do not remember their sizes for the next session.

You will encounter some windows in the debugger that appear to float atop the others. But because all of the activity here is within the context of an applet, the view to the debugger area is clipped to the outer edge of the debugger window. Don't be alarmed if you try to drag a window out of the debugger window only to have it slip out of view.

## Getting the debugger to single-step

You have two ways to instruct the JavaScript Debugger to pause script execution while the display switches to the debugger for single-stepping through the code. The easiest way is to click the BreakPoint (BrkPt) button in the toolbar. When you switch back to the browser window, the next script that runs (whether it be a script that runs during a document reload or in response to user action) will switch the view to the debugger window, with the page's source code appearing in the top window.

An alternate method is more useful if you know where in a document you want to start watching the script details. In the debugger window, click Open and select the document you wish to debug. When the source code window opens, scroll to the desired script statement where you want to begin stepping through the code. Set a breakpoint by clicking along the left margin of the desired line. A red dot appears where the breakpoint is. You can set more than one breakpoint in a document. Then return to the browser window, and work with the page until the script statement with the breakpoint executes. At that point, the view changes to the debugger window, and you can begin stepping through code while examining values.

## Stepping through code

When the debugger is running, the source document window displays vertical yellow bars where the script statement is as well as a blue arrow pointing to the line that is the *next one* to execute. To execute that statement, click the Into or Over buttons in the toolbar. The difference between the buttons becomes apparent if the statement about to be executed contains a call to a custom function in your application. If you click the Into button, the debugger scrolls to that subroutine function in the document (or opens a new source document window if it is in a different document). This allows you to follow the execution logic through nested functions. In the Call Stack window, you see the nested function at the top of the stack.

While stepping through the nested function, you can continue to click the Into or Over buttons, with the Into button leading to any further nested function. When the stepping reaches the end of the nested function, the view returns to the statement of the calling function (and the nested function name pops off the stack). From inside a nested function you can also zoom back out to the calling function by clicking the Out button. If you don't want to step through nested functions at all, click only the Over button.

Occasionally, you may need to inspect only a few places within your script, even though they may be separated by a lot of executing lines (for example, you want to step through lines above and below a repeat loop, but not through dozens of loop iterations). By setting multiple breakpoints in a document, you can single-step where you want and then let processing continue at normal speed until the next breakpoint. Click the Run button to exit single-stepping while letting the script continue its normal course. Or if you want to bail out of the script entirely, click Abort.

## Observing values manually

As the debugger single-steps through your code, you can inspect the current value of any expression in the code. This is a manual way of observing variables and expressions, and is convenient when you don't need continued views to certain values. To observe a value, select a variable or expression in the source code, and then click the Eval button in the toolbar (or right-click the selection with a two-button mouse). Appearing in the Console window is the item you selected and (in red) its current value. If a variable has not yet been initialized its value displays as `null`.

As a second manual method, you can also type an expression in the type-in field at the bottom of the Console window. This expression can be any valid JavaScript expression, including one that uses variables and document objects from the document currently being stepped through. When you press Enter, the value appears in the upper area of the Console window.

## Observing values automatically – watches

A more efficient way of keeping tabs on expression values while stepping through code is to set up what are called *watches*. These are values that evaluate for each step through the program (even if the value is not directly affected by the code in the step) and display their results in the Console window.

You can set up watches two ways. The easy way is to select an expression in the code (while the debugger is stepping or is at rest) and choose Copy to Watch from the Edit menu. To view the expressions being watched, choose Watches from the Edit menu. The window that appears lists each item. You can also click the New button in the Watches window to type in an expression to watch.

If expressions rely on each other, you may want to control the order in which expressions are evaluated at each step. You can select an item in the Watches window and click the Move Up or Move Down buttons to adjust the item's position in the list of watched items.

As the debugger steps through a script, all items in the Watches window are evaluated, and their results are displayed in the Console window. This lets you keep an eye on several items all at the same time without any extra clicking or selecting beyond the buttons that control the stepping.

## Writing Your Own Trace Utility

Single-stepping through running code with the 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 with 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?

With thoughtfully placed calls to the `trace()` method of the external library shown next, the resulting report you get after running your script can answer questions like these and many more.

## The `trace.js` library

Listing 45-2 is the listing for the `debug.js` library file. It consists of one global variable and one function.

### Listing 45-2: `trace.js` library

```
var timestamp = 0
function trace(flag, label, value) {
    if (flag) {
        var funcName = debug.caller.toString()
        funcName = funcName.substring(10, funcName.indexOf("(")) + 1)
        var msg = "In " + funcName + ": " + label + "=" + value
        var now = new Date()
        var elapsed = now - timestamp
        if (elapsed < 10000) {
            msg += " (" + elapsed + " msec)"
        }
        timestamp = now
        java.lang.System.out.println(msg)
    }
}
```

When this file loads into a document, the `timestamp` variable becomes a global variable in the document. This variable is used to maintain the last time the `trace()` function is called. The value must persist so that subsequent calls to `trace()` permit calculations of the time differences between the previous and current invocations. The timestamping is optional, but it can be useful for performing some benchmark tests of alternate ways of processing data.

The `trace()` function takes three parameters. The first, `flag`, is a Boolean value that determines whether the trace should proceed (I'll 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. By retrieving the rarely used `caller` property of a function (see Chapter 34), 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>
```

Beyond this trace data, the script subtracts the value of the `timestamp` variable from the current time to achieve an elapsed time. If that elapsed time is less than ten seconds, the time value is appended to the message (in parentheses).

The final statement in the function is a LiveConnect call to a native Java class method. Experienced Java programmers will recognize the `System.out.println()` method as the one that writes a value to the Java Console window. If you haven't done any Java programming, you may not even know that Navigator has a Java Console window available under the Options/Communicator menu. Java applet errors automatically appear in this window (even if the window is hidden). LiveConnect gives you the power to make a direct call to the built-in Java method, `java.lang.System.out.println()`, to write anything you like to the window. Anything written to that window is appended to the end of whatever text is already in the window. Therefore, you can write to that window as many trace messages as you like, and they'll all be there for you to see after the script runs.

## Loading the trace.js library

To include this debugging library in your document, add the following `<SCRIPT>` tag set at the start of the document's head section:

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

The syntax assumes that you have saved `trace.js` in the same directory as your HTML file.

## Preparing documents for trace.js

As you build your document and its scripts, you need to decide how granular you'd like 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 Java Console window 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 print 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 45-3 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 45-3: 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 Java Console window in Navigator shows the following trace:

```
In timeToString(input): input=869854500000
In timeToString(input): rawTime=Fri Jul 25 11:15:00 Pacific Daylight
Time 1997 (60 msec)
In twoDigitString(val): val=11 (0 msec)
In twoDigitString(val): val=15 (0 msec)
In timeToString(input): result=11:15 (220 msec)
```

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.

## About the timer

In the trace results of Listing 45-3 you might wonder how some statements appear to execute literally “in no time.” You have to take the timing values of this tracing scheme with a grain of salt. For one thing, these are not intended to be world-class standard benchmarks. Some of the processing in the `trace()` function itself occupies CPU cycles. And with rapid-fire execution in the small scripts of Listing 45-3, the timings are not very meaningful. But in a more complex script, especially one involving numerous calls to subroutines and nested loops, you can place the `trace()` function calls in statements not so deeply nested that the intervals are too small to be of any value. In any case, regard the values as relative, rather than absolute, values. And always run the script several times to help you see a pattern of performance.

## Navigator Crashes

Navigator, though seemingly reliable in its browser role, is not so stable when it comes to trying JavaScript statements that would normally be considered illegal. Fortunately, each Navigator generation significantly improves robustness over its previous version.

The seriousness of the crash depends partly on the internal error and the operating system. For instance, I’ve seen crashes on the Macintosh that range from just “unexpectedly quitting” Navigator (leaving everything else intact) to taking down the entire computer (necessitating a hard restart). Windows 95, on the other hand, protects the rest of the applications running when Navigator takes a dive.

As you develop scripts in a modern version of Navigator, you should be aware that improved robustness makes it all the more dangerous, as some methods could crash earlier Navigator versions. This is particularly true when invoking document-modifying methods, such as `document.write()` and `document.open()`. In Navigator 2, for example, such methods directed to the current document frequently lead to crashes. If you’re developing scripts for a production-quality Web site that invites visitors of all browser types, be sure to test your scripts on Navigator 2.02.

## Preventing Problems

Even with help of authoring tools and debuggers, 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 once 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 like the process of compiling code, 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 in a separate evaluation tester document you write with a couple text or textarea objects in it. Navigator users can use the internal `javascript: URL` to test expressions. 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 — 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 Navigator hardware platforms and JavaScript-enabled browsers as you can and with as wide an audience as possible before putting the pages on the server for all to see.

