23
CHAPTER

JavaScript Programming Practices

In this chapter we bring to a close our discussion of JavaScript by highlighting some recommended practices for and salient issues regarding JavaScript in the "real world." Our focus is on errors and debugging as well as on writing robust JavaScript that utilizes defensive programming techniques. We also touch on some distribution issues, such as protecting your code and decreasing its download time, and discuss where JavaScript fits into the "big picture" of the Web. The discussion in this chapter condenses many years worth of programming experience into a few dozen pages, so that developers—new ones in particular—can save themselves and their users some headaches by careful consideration of the content presented here.

Errors

Before launching into a discussion of how errors can be found and handled, it is useful to understand the taxonomy of errors found in typical scripts. The wide variety of errors that can occur during the execution of a script can be roughly placed into three categories: syntax errors, runtime errors, and semantic errors.

Syntax Errors

Of the three types of errors, s*yntax errors* are the most obvious. They occur when you write code that somehow violates the rules of the JavaScript language. For example, writing the following,

L 23-1
```
var x = y + * z;
```

is a syntax error because the syntax of the * operator requires two expressions to operate upon, and "y +" does not constitute a valid expression. Another example is

L 23-2
```
var myString = "This string doesn't terminate
```

because the string literal isn't properly quoted.

Syntax errors are generally *fatal* in the sense that they are errors from which the interpreter cannot recover. The reason they are fatal is that they introduce *ambiguity*, which the language syntax is specifically designed to avoid. Sometimes the interpreter can make some sort of assumption about what the programmer intended and can continue to execute the rest of the script. For example, in the case of a non-terminated string literal, the interpreter might assume that the string ends at the end of the line. However, scripts with syntax errors should, for all intents and purposes, be considered incorrect, even if they do run in some manner, as they do not constitute a valid program and their behavior can therefore be erratic, destructive, or otherwise anomalous.

Luckily, syntax errors are fairly easy to catch because they are immediately evident when the script is parsed before being executed. You cannot hide a syntax error from the interpreter in any way except by placing it in a comment. Even placing it inside a block that will never be executed, as in

L 23-3
```
if (false) { x = y + * z }
```

will still result in an error. The reason, as we have stated, is that these types of errors show up during the parsing of the script, a step that occurs before execution.

You can easily avoid syntax errors by turning on error warnings in the browser and then loading the script or by using one of the debuggers discussed later in this chapter.

Runtime Errors

The second category of errors are *runtime errors*, which are exactly what they sound like: errors that occur while the script is running. These errors result from JavaScript that has the correct syntax but which encounters some sort of problem in its execution environment. Common runtime errors result from trying to access a variable, property, method, or object that does not exist or from attempting to utilize a resource that is not available.

Some runtime errors can be found by examination of source code. For example,

L 23-4
```
window.allert("Hi there");
```

results in a runtime error because there is no **allert()** method of the **Window** object. This example constitutes perfectly legal JavaScript, but the interpreter cannot tell until runtime that invoking **window.allert()** is invalid, because such a method might have been added as an instance property at some previous point during execution.

Other kinds of runtime errors cannot be caught by examination of source code. For example, while the following might appear to be error-free,

L 23-5
```
var products = ["Widgets", "Snarks", "Phasers"];
var choice = parseInt(prompt("Enter the number of the product you are
interested in"));
alert("You chose: " + products[choice]);
```

what happens if the user enters a negative value for *choice*? A runtime error indicating the array index is out of bounds.

Although some defensive programming can help here,

L 23-6
```
var products = ["Widgets", "Snarks", "Phasers"];
var choice = parseInt(prompt("Enter the number of the product in which
you are interested"));
if (choice >= 0 && choice < products.length)
  alert("You chose: " + products[choice]);
```

the reality is that you cannot catch all potential runtime errors before they occur. You can, however, catch them at runtime using JavaScript's error and exception handling facilities, which are discussed later in the chapter.

Semantic Errors

The final category of errors, *semantic errors*, occur when the program executes a statement that has an effect that was unintended by the programmer. These errors are much harder to catch because they tend to show up under odd or unusual circumstances and therefore go unnoticed during testing. The most common semantic errors are the result of JavaScript's weak typing; for example:

L 23-7

```
function add(x, y)
{
   return x + y;
}
var mySum = add(prompt("Enter a number to add to five",""), 5);
```

If the programmer intended **add()** to return the numeric sum of its two arguments, then the code above is a semantic error in the sense that *mySum* is assigned a string instead of a number. The reason, of course, is that **prompt()** returns a string that causes **+** to act as the string concatenation operator, rather than as the numeric addition operator.

Semantic errors arise most often as the result of interaction with the user. They can usually be avoided by including explicit checking in your functions. For example, we could redefine the **add()** function to ensure that the type and number of the arguments are correct:

L 23-8

```
function add(x, y)
{
   if (arguments.length != 2 || typeof(x) != "number" || typeof(y) !=
"number")
      return(Number.NaN);
   return x + y;
}
```

Alternatively, the **add()** function could be rewritten to attempt to convert its arguments to numbers—for example, by using the **parseFloat()** or **parseInt()** functions.

In general, semantic errors can be avoided (or at least reduced) by employing defensive programming tactics. If you write your functions anticipating that users and programmers will purposely try to break them in every conceivable fashion, you can save yourself future headaches. Writing "paranoid" code might seem a bit cumbersome, but doing so enhances code reusability and site robustness (in addition to showcasing your mature attitude towards software development).

A summary of our error taxonomy is found in Table 23-1, and the next few sections will cover each of the mitigation techniques in detail.

Error Type	Results From	Mitigation Technique
Syntax error	Violating the rules of the JavaScript language	Turn on scripting error reporting and use a debugger
Runtime error	Syntactically valid script that attempts to do something impossible while running (e.g., invoking a function that doesn't exist)	Defensive programming, use exception handling, turn on scripting error reporting, use a debugger
Semantic error	Script that does something unintended by the programmer	Defensive programming and use a debugger

TABLE 23-1 Categories of JavaScript **Programming Errors**

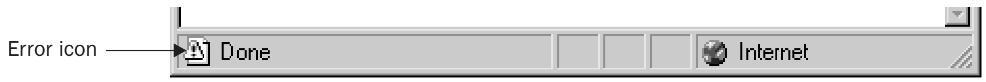
Debugging

Every programmer makes mistakes, and a large part of becoming a more proficient developer is honing your instincts for finding and rooting out errors in your code. Debugging is a skill that is best learned through experience, and although basic debugging practices can be taught, each programmer must develop his/her own approach. In this section we cover tools and techniques that can help you with these tasks.

Turning on Error Messages

The most basic way to track down errors is by turning on error information in your browser. By default, Internet Explorer shows an error icon in the status bar when an error occurs on the page:

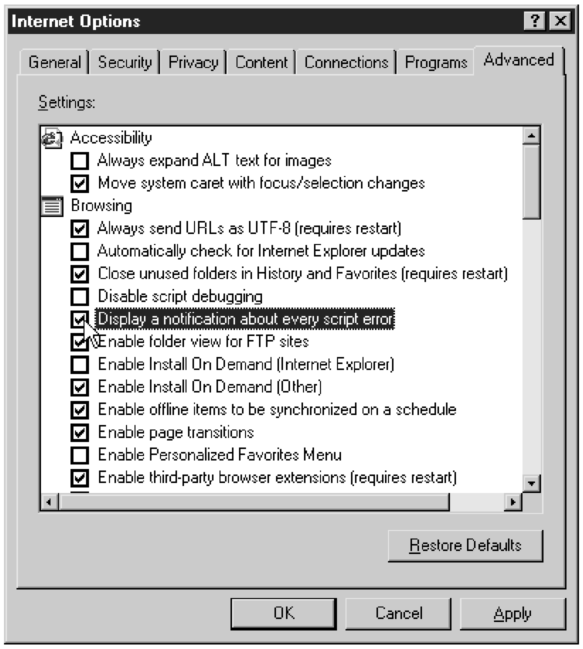
Ill 23-1



Double-clicking this icon takes you to a dialog box showing information about the specific error that occurred.

Because this icon is easy to overlook, Internet Explorer gives you the option to automatically show the Error dialog box whenever an error occurs. To enable this option, select Tools | Internet Options, and click the Advanced tab. Check the Display a Notification About Every Script Error box, as shown in Figure 23-1.

Although Netscape 3 shows an error dialog each time an error occurs, Netscape 4+ and Mozilla browsers send error messages to a special window called the *JavaScript Console*. To view the Console in Netscape and Mozilla, type **javascript:** in the browser's Location bar. In Netscape 7+ and Mozilla you can also pull up the Console using the Tools menu (select Tools | Web Development). Unfortunately, since Netscape 6+ and Mozilla give no visual indication when an error occurs, you must keep the JavaScript Console open and watch for errors as your script executes.



Accessibility
- ☐ Always expand ALT text for images
- ✓ Move system caret with focus/selection changes

Browsing
- ✓ Always send URLs as UTF-8 (requires restart)
- ☐ Automatically check for Internet Explorer updates
- ✓ Close unused folders in History and Favorites (requires restart)
- ☐ Disable script debugging
- ✓ Display a notification about every script error
- ✓ Enable folder view for FTP sites
- ☐ Enable Install On Demand (Internet Explorer)
- ✓ Enable Install On Demand (Other)
- ✓ Enable offline items to be synchronized on a schedule
- ✓ Enable page transitions
- ☐ Enable Personalized Favorites Menu
- ✓ Enable third-party browser extensions (requires restart)

FIGURE 23-1 Enabling notification of script errors in Internet Explorer

---

NOTE *In Netscape 6 the JavaScript Console is found in the Tasks menu (select Tasks | Tools).*

Error Notifications

Error notifications that show up on the JavaScript Console or through Internet Explorer dialog boxes are the result of both syntax and runtime errors. Loading a file with the syntax error from a previous example, **var myString = "This string doesn't terminate** results in the error dialog and JavaScript Console messages in Figure 23-2. Loading a file with the runtime error from a previous example, **window.allert("Hi there");** results in the error dialog and JavaScript Console shown in Figure 23-3.

A very helpful feature of this kind of error reporting is that it includes the line number at which the error occurred. However, you should be aware that occasionally line numbers can become skewed as the result of externally linked files. Most of the time error messages are fairly easy to decipher, but some messages are less descriptive than others, so it is useful to explicitly mention some common mistakes here.

Common Mistakes

Table 23-2 indicates some common JavaScript mistakes and their symptoms. This list is by no means exhaustive, but it does include the majority of mistakes made by novice programmers. Of this list, errors associated with type mismatches and access to form elements are probably the hardest for beginners to notice, so you should take special care when interacting with forms or other user-entered data.

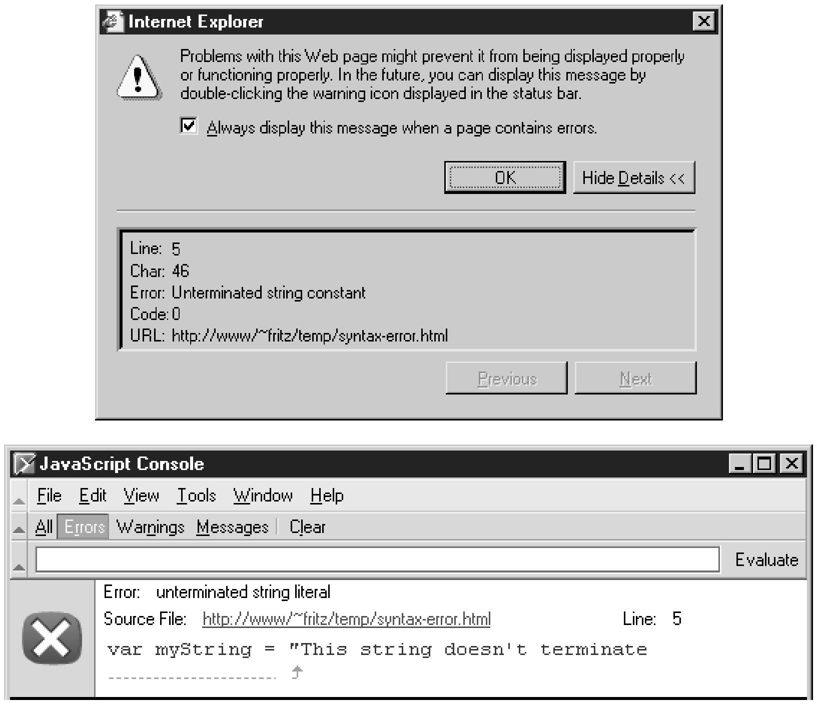


FIGURE 23-2 Syntax errors in a) Internet Explorer and b) Mozilla

Internet Explorer
Problems with this Web page might prevent it from being displayed properly or functioning properly. In the future, you can display this message by double-clicking the warning icon displayed in the status bar.
Always display this message when a page contains errors.
OK
Hide Details <<
Previous
Next
JavaScript Console
File Edit View Tools Window Help
All Errors Warnings Messages Clear
Evaluate

```
Line:  5
Char:  46
Error: Unterminated string constant
Code: 0
URL:  http://www/~fritz/temp/syntax-error.html
```

```
Error:  unterminated string literal
Source File:  http://www/~fritz/temp/syntax-error.html        Line:  5
var myString = "This string doesn't terminate
```

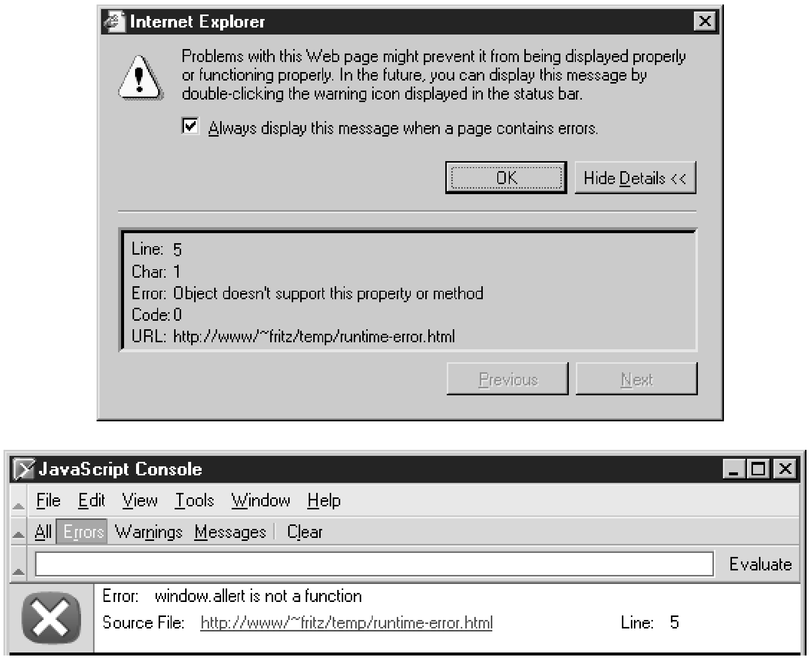


FIGURE 23-3 Runtime errors in a) Internet Explorer and b) Mozilla

Internet Explorer
Problems with this Web page might prevent it from being displayed properly or functioning properly. In the future, you can display this message by double-clicking the warning icon displayed in the status bar.
Always display this message when a page contains errors.
OK
Hide Details <<
Previous
Next
JavaScript Console
File Edit View Tools Window Help
All Errors Warnings Messages Clear
Evaluate

```
Line:  5
Char:  1
Error: Object doesn't support this property or method
Code: 0
URL:  http://www/~fritz/temp/runtime-error.html
```

```
Error:  window.allert is not a function
Source File:  http://www/~fritz/temp/runtime-error.html        Line:  5
```

Mistake	Example	Symptom

Infinite loops	while (*x*<myrray.length)
	dosomething(myarray[x]);
		A stack overflow error or a totally unresponsive page.

Using assignment instead of comparison (and vice versa)	if (*x* = 10)
	// or
	var x == 10;
		Clobbered or unexpected values. Some JavaScript implementations automatically fix this type of error. Many programmers put the variable on the right-hand side of a comparison in order to cause an error when this occurs. For example, "if (10 = x)".

Unterminated string literals	var myString = "Uh oh	An "unterminated string literal" error message or malfunctioning code.

Mismatched parentheses	if (typeof(*x*) == "number"
	alert("Number");
		A "syntax error," "missing ')'", or "expected ')'" error message.

Mismatched curly braces	function mult(*x*,*y*)
	{
	return (*x*,*y*);
		Extra code being executed as part of a function or conditional, functions that are not defined, and "expected '}'", "missing '}'", or "mismatched '}'" error messages.

Mismatched brackets	*x*[0 = 10;	"invalid assignment," "expected ']'", or "syntax error" error messages.

Misplaced semicolons	if (isNS4 == true);
	hideLayers();
		Conditional statements always being executed, functions returning early or incorrect values, and very often errors associated with unknown properties.

Omitted "break" statements	switch(browser)
	{
	case "IE": // IE-specific
	case "NS": // NS-specific
	}
		Statements in the latter part of the **switch** always being executed and very often errors associated with unknown properties will occur as well.

Type errors	var sum = 2 + "2";	Values with an unexpected type, functions requiring a specific type not working correctly, and computations resulting in **NaN**.
Accessing undefined variables	var *x* = variableName;	"*variableName* is not defined" error messages.
Accessing non-existent object properties	var *x* = window.propertyName;	**undefined** values where you do not expect them, computations resulting in **NaN**, "*propertyName* is null or not an object," or "*objectName* has no properties" error messages.
Invoking non-existent methods	window.methodName()	"*methodName* is not a function," or "object doesn't support this property or method" error messages.

TABLE 23-2 Common JavaScript **Errors and Their Symptoms**

Mistake	Example	Symptom
Invoking undefined functions	noSuchFunction();	"object expected" or "*noSuchFunction* is not defined" error messages.
Accessing the document before it has finished loading	<head><script>var myElement= document.all.myElement;</ script></head>	**undefined** values, errors associated with nonexistent properties and methods, transitory errors that go away after page load.
Accessing a form element rather than its value	var *x* = document.myform.myfield;	Computation resulting in **NaN**, broken HTML-JS references, and form "validation" that always rejects its input.
Assuming that detecting an object or method assumes the existence of all other features related to the detected object	if (document.layers) { // do Netscape 4 stuff } if (document.all) { // do all sorts of IE stuff }	Probably will result in an error message complaining about a nonexistent object or property, because other proprietary objects beyond the detected ones were assumed to be presented and then used.

TABLE 23-2 Common JavaScript Errors and Their Symptoms *(continued)*

Using some sort of integrated development environment (IDE) or Web editor that matches parentheses and that colors your code is often helpful in avoiding syntax errors. Such programs automatically show where parentheses and brackets match and provide visual indications of the different parts of the script. For example, comments might appear in red while keywords appear blue and string literals appear in black.

Debugging Techniques

Although turning on error messages and checking for common mistakes can help you find some of the most obvious errors in your code, doing so is rarely helpful for finding semantic errors. There are, however, some widespread practices that many developers employ when trying to find the reason for malfunctioning code.

Manually Outputting Debugging Information

One of the most common techniques is to output verbose status information as the script runs in order to verify the flow of execution. For example, a debugging flag might be set at the beginning of the script that enables or disables debugging output included within each function. The most common way to output information in JavaScript is using the **alert()** method; for example, you might write something like

L 23-9
```
var debugging = true;
var whichImage = "widget";
```

```
if (debugging)
   alert("About to call swapImage() with argument: " + whichImage);
var swapStatus = swapImage(whichImage);
if (debugging)
   alert("Returned from swapImage() with swapStatus="+swapStatus);
```

and include **alert()**s marking the flow of execution in *swapImages()*. By examining the content and order of the **alert()**s as they appear, you are granted a window to the internal state of your script.

Because using many **alert()**s when debugging large or complicated scripts may be impractical (not to mention annoying), output is often sent to another browser window instead. Using this technique, a new window, say, *debugWindow* is opened at the beginning of the script, and debugging information is written into the window using syntax like ***debugWindow*.document.write()** method. The only potential gotcha is that you need to wait for the window to actually be opened before attempting to **write()** to it. See Chapter 12 for more information on inter-window communication.

Stack Traces Whenever one function calls another, the interpreter must keep track of the calling function so that when the called function returns, it knows where to continue execution. Such records are stored in the *call stack*, and each entry includes the name of the calling function, the line number of invocation, arguments to the function, and other local variable information. For example, consider this simple code:

L 23-10

```
function a(x)
{
  document.write(x);
}
function b(x)
{
  a(x+1);
}
function c(x)
{
  b(x+1);
}
c(10);
```

At the **document.write** in *a()*, the call stack looks something like:

a(12), line 3, local variable information…
b(11), line 7, local variable information…
c(10), line 11, local variable information…

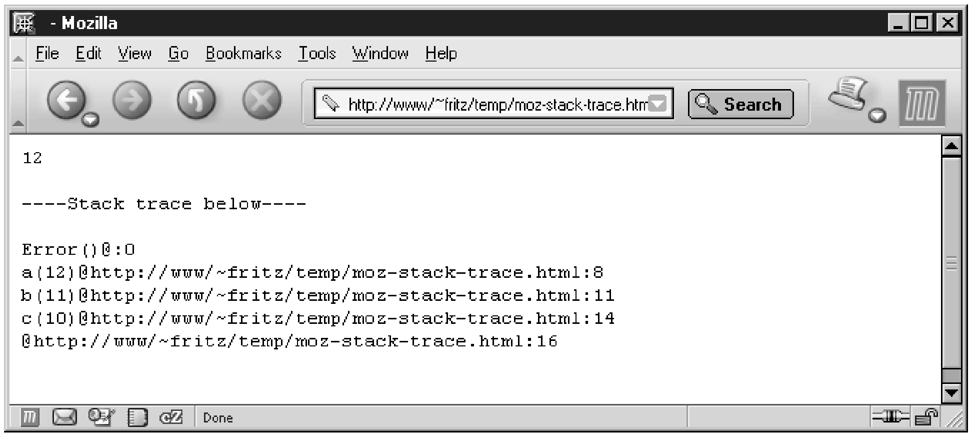
When *a()* returns, *b()* will continue executing on line 8 and when it returns, *c()* will continue executing on line 12.

A listing of the call stack is known as a *stack trace*, and can be useful when debugging. Mozilla provides the **stack** property of the **Error** object (discussed in detail in a following section) for just such occasions. We can augment our previous example to output a stack trace in Mozilla:

L 23-11

```
function a(x)
{
  document.writeln(x);
  document.writeln("\n----Stack trace below----\n");
  document.writeln((new Error).stack);
}
function b(x) {
  a(x+1);
}
function c(x) {
  b(x+1);
}
c(10);
```

The output is shown in Figure 23-4. The top of the trace shows that the **Error()** constructor is called. The next line indicates that the function that called the error constructor is *a()* and its argument was **10**. The other data on the line indicates the filename where this function is defined (after the @) as well as the line number (after the colon) the interpreter is currently executing. Successive lines show the calling functions as we'd expect, and the final line shows that *c()* was called on line 16 of the currently executing file (the call to *c()* isn't within any function, so the record on the stack doesn't list a function name).


- Mozilla
File Edit View Go Bookmarks Tools Window Help
http://www/~fritz/temp/moz-stack-trace.htm
Search
Done

```
12

----Stack trace below----

Error()@:0
a(12)@http://www/~fritz/temp/moz-stack-trace.html:8
b(11)@http://www/~fritz/temp/moz-stack-trace.html:11
c(10)@http://www/~fritz/temp/moz-stack-trace.html:14
@http://www/~fritz/temp/moz-stack-trace.html:16
```

FIGURE 23-4 Using Error.stack to get a stack trace in Mozilla

Other browsers don't provide an easy mechanism to get a stack trace, but given the **Function** properties discussed in Chapter 5, we can construct what it must look like ourselves.

L 23-12
```
// Helper function to parse out the name from the text of the function
function getFunctionName(f)
{
  if (/function (\w+)/.test(String(f)))
    return RegExp.$1;
  else
    return "";
}

// Manually piece together a stack trace using the caller property
function constructStackTrace(f)
{
  if (!f)
    return "";

  var thisRecord = getFunctionName(f) + "(";

  for (var i=0; i<f.arguments.length; i++) {
    thisRecord += String(f.arguments[i]);
    // add a comma if this isn't the last argument
    if (i+1 < f.arguments.length)
      thisRecord += ", ";
  }

  return thisRecord + ")\n" + constructStackTrace(f.caller);
}

// Retrieve a stack trace. Works in Mozilla and IE.
function getStackTrace() {
  var err = new Error;
  // if stack property exists, use it; else construct it manually
  if (err.stack)
    return err.stack;
  else
    return constructStackTrace(getStackTrace.caller);
}
```

We can now write out the example as:

L 23-13
```
function a(x)
{
  document.writeln(x);
  document.writeln("\n----Stack trace below----\n");
  document.writeln(getStackTrace());
}
```

```
function b(x)
{
  a(x+1);
}
function c(x)
{
  b(x+1);
}
c(10);
```

The output in Internet Explorer is shown in Figure 23-5.

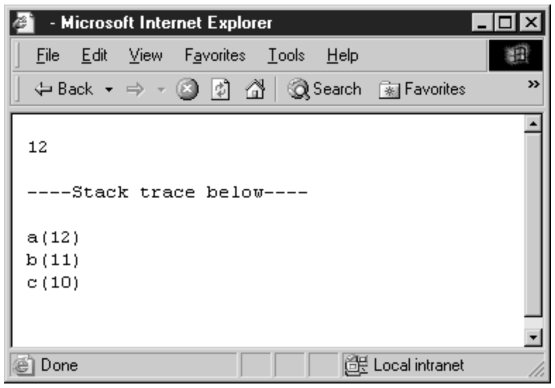
This is a handy function to have in an external script for debugging. However, the capabilities of this function and the techniques we've discussed so far leave a lot to be desired. They rely on manual insertion of debugging code into your scripts, and don't provide any interactivity. Fortunately, specialized tools enable far more in-depth examination of your code at runtime.

Using a Debugger

A *debugger* is an application that places all aspects of script execution under the control of the programmer. Debuggers provide fine-grain control over the state of the script through an interface that allows you to examine and set values as well as control the flow of execution.

Once a script has been loaded into a debugger, it can be run one line at a time or instructed to halt at certain *breakpoints*. The idea is that once execution is halted, the programmer can examine the state of the script and its variables in order to determine if something is amiss. You can also *watch* variables for changes in their values. When a variable is watched, the debugger will suspend execution whenever the value of the variable changes. This is tremendously useful in trying to track down variables that are mysteriously getting clobbered. Most debuggers also allow you to examine stack traces, the call tree representing the flow of execution through various pieces of code that we saw in the previous section. And to top it all off, debuggers are often programmed to alert the programmer when a potentially problematic piece of code is encountered. And because

FIGURE 23-5 A manually constructed stack trace

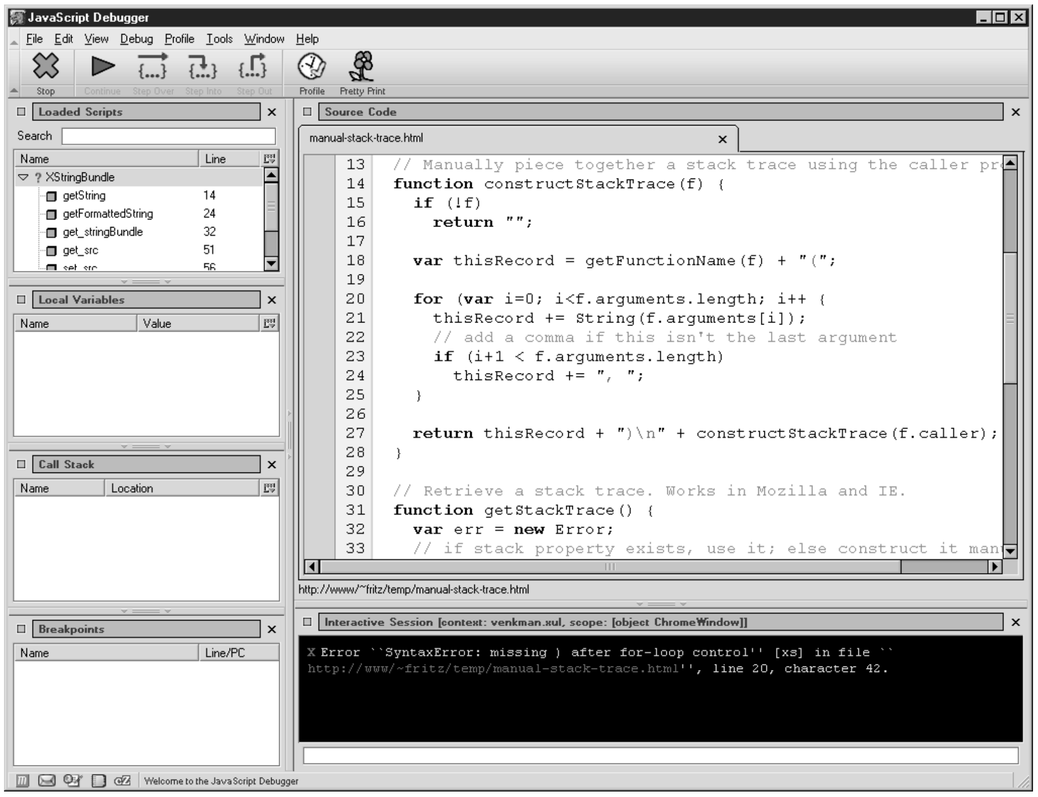

- Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Done
Local intranet

```
12

----Stack trace below----

a(12)
b(11)
c(10)
```

debuggers are specifically designed to track down problems, the error messages and warnings they display tend to be more helpful than those of the browser.

There are several major JavaScript debuggers in current use. By far the most popular free debugger is Venkman, the debugger of the Mozilla project. It integrates with Mozilla and Netscape 6+ and offers all the features most developers might need, including a profiler enabling you to measure the performance of your code. If you've installed the "Full" version of a Mozilla-based browser, this debugger is already available to you. If not, use a Mozilla-based browser to access **http://www.mozilla.org/projects/venkman/** and follow the installation instructions. This should be as simple as clicking on the .xpi file for the version you want. To start the debugger, select Tools | Web Development | JavaScript Debugger. Figure 23-6 shows a screenshot of Venkman.

A somewhat popular free utility for Internet Explorer 4 and later is the Microsoft Script Debugger. It is available from **http://msdn.microsoft.com/scripting** and integrates with Internet Explorer if installed. To enable this integration, select Tools | Internet Options. In the Advanced tab, uncheck Disable Script Debugging, as shown in Figure 23-7. Whenever debugging is turned on in IE and you load a page that has errors, the following dialog



Name	Line
? XStringBundle	
getString	14
getFormattedString	24
get_stringBundle	32
get_src	51
set_src	56

Name	Value

Name	Location

Name	Line/PC

```
13  // Manually piece together a stack trace using the caller pr
14  function constructStackTrace(f) {
15    if (!f)
16      return "";
17
18    var thisRecord = getFunctionName(f) + "(";
19
20    for (var i=0; i<f.arguments.length; i++ {
21      thisRecord += String(f.arguments[i]);
22      // add a comma if this isn't the last argument
23      if (i+1 < f.arguments.length)
24        thisRecord += ", ";
25    }
26
27    return thisRecord + ")\n" + constructStackTrace(f.caller);
28  }
29
30  // Retrieve a stack trace. Works in Mozilla and IE.
31  function getStackTrace() {
32    var err = new Error;
33    // if stack property exists, use it; else construct it man
```

http://www/~fritz/temp/manual-stack-trace.html

```
X Error ``SyntaxError: missing ) after for-loop control'' [xs] in file ``
http://www/~fritz/temp/manual-stack-trace.html'', line 20, character 42.
```

FIGURE 23-6 The Venkman JavaScript debugger in action

PART VI

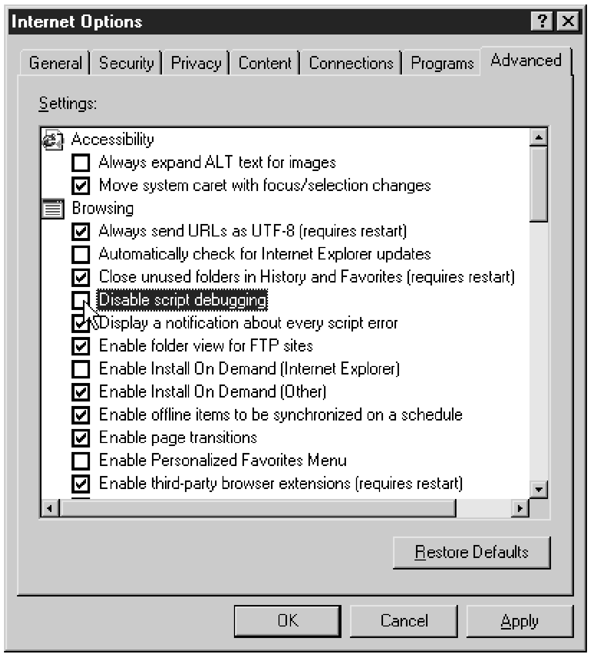


FIGURE 23-7 Enabling script debugging in Internet Explorer

Settings:

- Accessibility
  - ☐ Always expand ALT text for images
  - ☑ Move system caret with focus/selection changes
- Browsing
  - ☑ Always send URLs as UTF-8 (requires restart)
  - ☐ Automatically check for Internet Explorer updates
  - ☑ Close unused folders in History and Favorites (requires restart)
  - ☐ Disable script debugging
  - ☑ Display a notification about every script error
  - ☑ Enable folder view for FTP sites
  - ☐ Enable Install On Demand (Internet Explorer)
  - ☑ Enable Install On Demand (Other)
  - ☑ Enable offline items to be synchronized on a schedule
  - ☑ Enable page transitions
  - ☐ Enable Personalized Favorites Menu
  - ☑ Enable third-party browser extensions (requires restart)

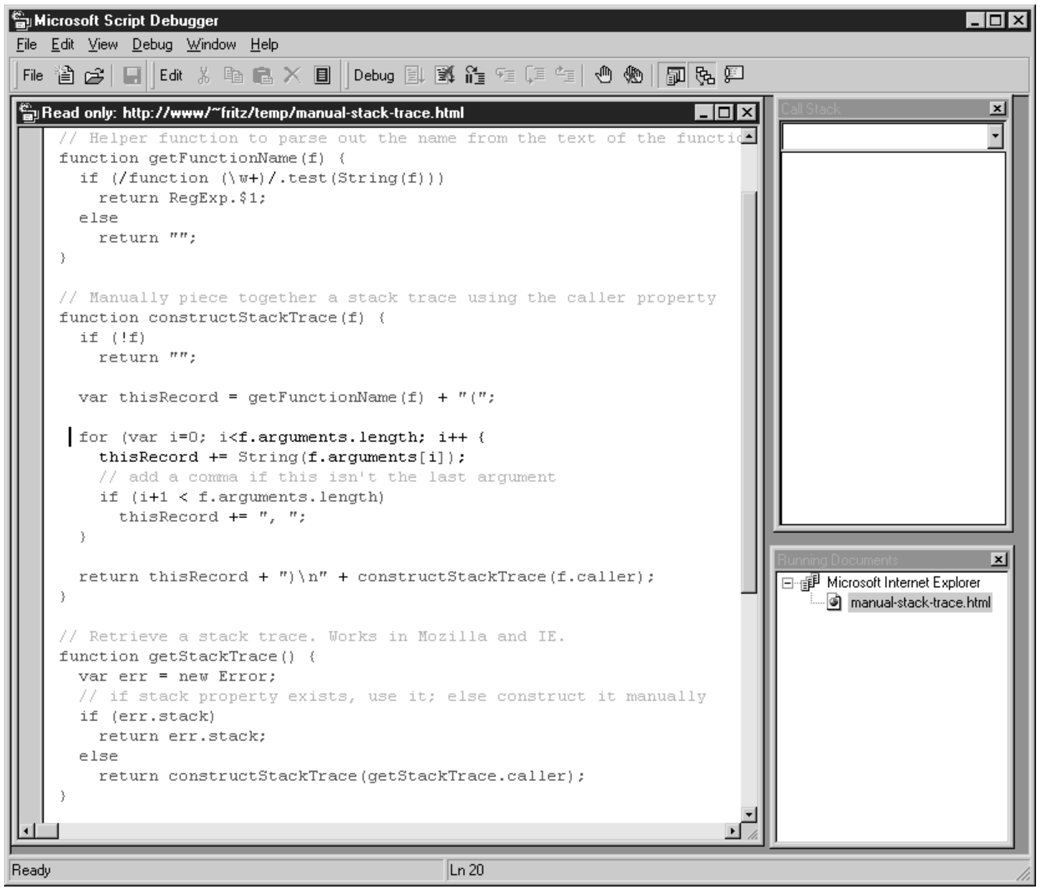
is shown in place of the normal error message, allowing you to load the page into the debugger.

Of course, you can also load a document directly into the debugger without having an error occur.

The Microsoft Script Debugger has the advantage of close coupling with Microsoft's JScript and Document Object Model, but no longer appears to be under active development. Microsoft Script Debugger is shown in Figure 23-8.

The final major option you have is to use a commercial development environment. A JavaScript debugger is usually just one small part of such development tools, which can offer sophisticated HTML and CSS layout capabilities and can even automate certain aspects of site generation. This option is often the best choice for professional developers, because chances are you will need a commercial development environment anyway, so you might as well choose one with integrated JavaScript support. A typical example of such an environment is Macromedia's Dreamweaver, available from **http://www.macromedia.com/software/dreamweaver/**. There are two primary drawbacks to such environments. The first and most obvious is the expense. The second is the fact that such tools tend to emit spaghetti code, so trying to hook your handwritten code into JavaScript or HTML and CSS generated by one of these tools can be tedious.

Now that we have covered some tools for tracking down errors in your code, we turn to techniques you can use to prevent or accommodate problems that might be outside of your direct control.


Microsoft Script Debugger
File Edit View Debug Window Help
File Edit Debug
Read only: http://www/~fritz/temp/manual-stack-trace.html
Call Stack
Running Documents
Microsoft Internet Explorer
manual-stack-trace.html
Ready
Ln 20

```
// Helper function to parse out the name from the text of the functio
function getFunctionName(f) {
  if (/function (\w+)/.test(String(f)))
    return RegExp.$1;
  else
    return "";
}

// Manually piece together a stack trace using the caller property
function constructStackTrace(f) {
  if (!f)
    return "";

  var thisRecord = getFunctionName(f) + "(";

  for (var i=0; i<f.arguments.length; i++ {
    thisRecord += String(f.arguments[i]);
    // add a comma if this isn't the last argument
    if (i+1 < f.arguments.length)
      thisRecord += ", ";
  }

  return thisRecord + ")\n" + constructStackTrace(f.caller);
}

// Retrieve a stack trace. Works in Mozilla and IE.
function getStackTrace() {
  var err = new Error;
  // if stack property exists, use it; else construct it manually
  if (err.stack)
    return err.stack;
  else
    return constructStackTrace(getStackTrace.caller);
}
```

FIGURE 23-8 Use Microsoft Script Debugger to help track down errors.

Defensive Programming

Defensive programming is the art of writing code that functions properly under adverse conditions. In the context of the Web, an “adverse condition” could be many different things: for example, a user with a very old browser or an embedded object or frame that gets stuck while loading. Coding defensively involves an awareness of the situations in which something can go awry. Some of the most common possibilities you should try to accommodate include

- Users with JavaScript turned off
- Users with cookies turned off
- Embedded Java applets that throw an exception

- Frames or embedded objects that load incorrectly or incompletely
- Older browsers that do not support modern JavaScript objects or methods
- Older browsers with incomplete JavaScript implementations—for example, those that do not support a specific feature such as the **push()**, **pop()**, and related methods in the **Array** object of versions of Internet Explorer prior to 5.5
- Browsers with known errors, such as early Netscape browsers with incorrectly functioning **Date** objects
- Users with text-based or aural browsers
- Users on non-Windows platforms
- Malicious users attempting to abuse a service or resource through your scripts
- Users who enter typos or other invalid data into form fields or dialog boxes, such as entering letters in a field requiring numbers

The key to defensive programming is flexibility. You should strive to accommodate as many different possible client configurations and actions as you can. From a coding standpoint, this means you should include HTML (such as **<noscript>**s) and browser sensing code that permit graceful degradation of functionality across a variety of platforms. From a testing standpoint, this means you should always run a script in as many different browsers and versions and on as many different platforms as possible before placing it live on your site.

In addition to accommodating the general issues described above, you should also consider the specific things that might go wrong with your script. If you are not sure when a particular language feature you are using was added to JavaScript, it is always a good idea to check a reference, such as Appendix B of this book, to make sure it is well supported. If you are utilizing dynamic page manipulation techniques or trying to access embedded objects, you might consider whether you have appropriate code in place to prevent execution of your scripts while the document is still loading. If you have linked external .js libraries, you might include a flag in the form of a global variable in each library that can be checked to ensure that the script has properly loaded.

The following sections outline a variety of specific techniques you can use for defensive programming. While no single set of ideas or approaches is a panacea, applying the following principles to your scripts can dramatically reduce the number of errors your clients encounter. Additionally, they can help you solve those errors that *are* encountered in a more timely fashion, as well as "future proof" your scripts against new browsers and behaviors.

However, at the end of the day, the efficacy of defensive programming comes down to the skill, experience, and attention to detail of the individual developer. If you can think of a way for the user to break your script or to cause some sort of malfunction, this is usually a good sign that more defensive techniques are required.

Error Handlers

Internet Explorer 3+ and Netscape 3+ provide primitive error-handling capabilities through the nonstandard **onerror** handler of the **Window** object. By setting this event handler, you can augment or replace the default action associated with runtime errors on the page. For example, you can replace or suppress the error messages shown in Netscape 3 and Internet Explorer (with debugging turned on) and the output to the JavaScript Console in Netscape 4+. The values to which **window.onerror** can be set and the effects of doing so are outlined in Table 23-3.

Value of window.onerror	Effect
Null	Suppresses reporting of runtime errors in Netscape 3+.
A function	The function is executed whenever a runtime error occurs. If the function returns **true** then the normal reporting of runtime errors is suppressed. If it returns **false** the error is reported in the browser as usual.

TABLE 23-3 **window.onerror** Values and Effects

NOTE *The **onerror** handler is also available for objects other than **Window** in many browsers, most notably the **<img>** and **<object>** elements.*

For example, to suppress error messages in older browsers you might use

L 23-14
```
function doNothing() { return true; }
window.onerror = doNothing;
window.noSuchProperty()   // throw a runtime error
```

Since modern browsers don't typically display script errors unless users specifically configure them to do so, the utility of the return value is limited.

The truly useful feature of **onerror** handlers is that they are automatically passed three values by the browser. The first argument is a string containing an error message describing the error that occurred. The second is a string containing the URL of the page that generated the error, which might be different from the current page if, for example, the document has frames. The third parameter is a numeric value indicating the line number at which the error occurred.

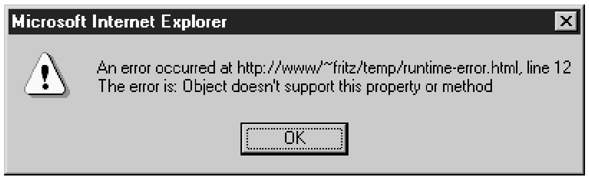
NOTE *Early versions of Netscape 6 did not pass these values to **onerror** handlers.*

You can use these parameters to create custom error messages, such as:

L 23-15
```
function reportError(message, url, lineNumber)
{
 if (message && url && lineNumber)
   alert("An error occurred at "+ url + ", line " + lineNumber +
"\nThe error is: " + message);
 return true;
}
window.onerror = reportError;     // assign error handler
window.noSuchProperty();          // throw an error
```

The result of which in Internet Explorer might be

Ill 23-2



An error occurred at http://www/~fritz/temp/runtime-error.html, line 12
The error is: Object doesn't support this property or method

There are two important issues regarding use of the **onerror** handler. The first is that this handler fires only as the result of runtime errors; syntax errors do not trigger the **onerror** handler and in general cannot be suppressed. The second is that support for this handler is spotty under some versions of Internet Explorer. While Internet Explorer 4, 5.5, and 6 appear to have complete support, some versions of Internet Explorer 5.0 might have problems.

Automatic Error Reporting

An interesting use for this feature is to add automatic error reporting to your site. You might trap errors and send the information to a new browser window, which automatically submits the data to a CGI or which loads a page that can be used to do so. We illustrate the concept with the following code. Suppose you have a CGI script "submitError.cgi" on your server that accepts error data and automatically notifies the webmaster or logs the information for future review. You might then write the following page, which retrieves data from the document that opened it and allows the user to include more information about what happened. This file is named "errorReport.html" in our example:

L 23-16

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Error Submission</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<script type="text/javascript">
<!--
/* fillValues() is invoked when the page loads and retrieves error data
from the offending document */
function fillValues()
{
  if (window.opener && !window.opener.closed && window.opener.lastErrorURL)
   {
    document.errorForm.url.value = window.opener.lastErrorURL;
    document.errorForm.line.value = window.opener.lastErrorLine;
    document.errorForm.message.value = window.opener.lastErrorMessage;
    document.errorForm.userAgent.value = navigator.userAgent;
   }
}
//-->
</script>
</head>
<body onload="fillalues()">
<h2>An error occurred</h2>
Please help us track down errors on our site by describing in more detail
what you were doing when the error occurred. Submitting this form helps us
improve the quality of our site, especially for users with your browser.
<form id="errorForm" name="errorForm" action="/cgi-bin/submitError.cgi">
The following information will be submitted:<br />
URL: <input type="text" name="url" id="url" size="80" /><br />
Line: <input type="text" name="line" id="line" size="4" /><br />
Error: <input type="text" name="message" id="message" size="80" /><br />
Your browser: <input type="text" name="usergent" id="usergent" size="60" />
<br />
```

```
Additional Comments:<br />
<textarea name="comments" cols="40" rows="5"></textarea><br />
<input type="submit" value="Submit to webmaster" />
</form>
</body>
</html>
```

The other part of the script is placed in each of the pages on your site and provides the information that *fillValues()* requires. It does so by setting a handler for **onerror** that stores the error data and opens "errorReport.html" automatically when a runtime error occurs:

L 23-17
```
var lastErrorMessage, lastErrorURL, lastErrorLine;
// variables to store error data
function reportError(message, url, lineNumber)
{
   if (message && url && lineNumber)
    {
      lastErrorMessage = message;
      lastErrorURL = url;
      lastErrorLine = lineNumber;
      window.open("errorReport.html");
    }
   return true;
}
window.onerror = reportError;
```

When "errorReport.html" is opened as a result of an error, it retrieves the relevant data from the window that opened it (the window with the error) and presents the data to the user in a form. Figure 23-9 shows the window opened as the result of the following runtime error:

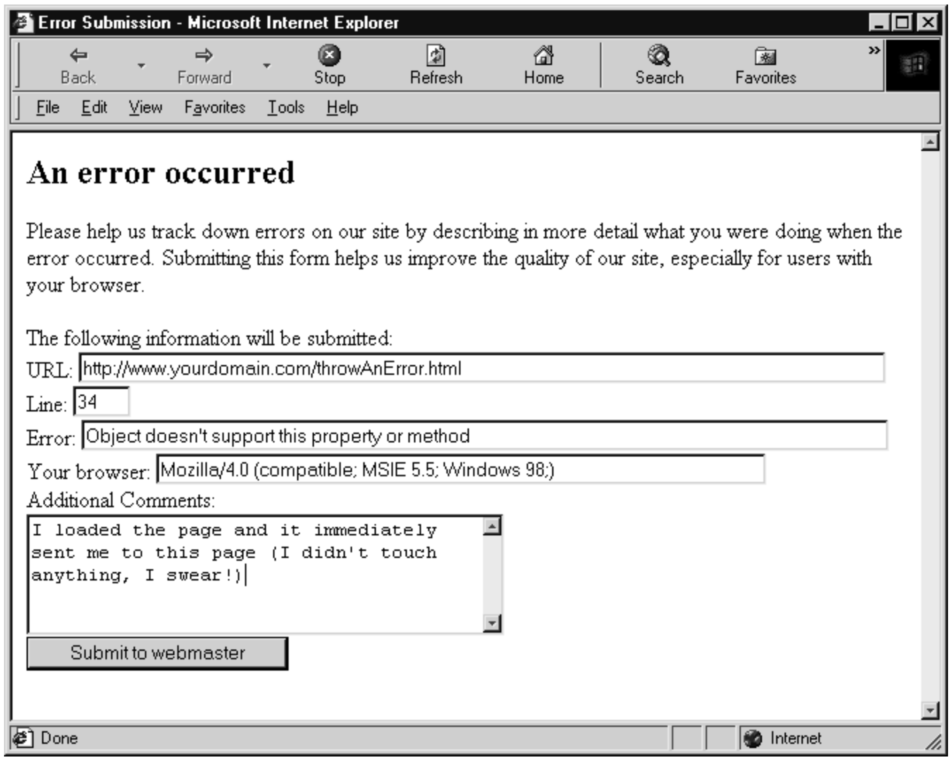
L 23-18
```
window.noSuchMethod();
```

The first four form values are automatically filled in by *fillValues()*, and the **<textarea>** shows a hypothetical description entered by the user. Of course, the presentation of this page needs some work (especially under Netscape 4), but the concept is solid.

Exceptions

An *exception* is a generalization of the concept of an error to include any unexpected condition encountered during execution. While errors are usually associated with some unrecoverable condition, exceptions can be generated in more benign problematic situations and are not usually fatal. JavaScript 1.4+ and JScript 5.0+ support exception handling as the result of their movement towards ECMAScript conformance.

When An Exception Is Generated, It Is Said To Be *Thrown* (Or, In Some Cases, *Raised*). The Browser May Throw Exceptions In Response To Various Tasks, Such As Incorrect Dom Manipulation, But Exceptions Can Also Be Thrown By The Programmer Or Even An Embedded Java Applet. Handling An Exception Is Known As *Catching* An Exception. Exceptions Are Often Explicitly Caught By The Programmer When Performing Operations That He Or She Knows Could Be Problematic. Exceptions That Are *Uncaught* Are Usually Presented To The User As Runtime Errors.



An error occurred

Please help us track down errors on our site by describing in more detail what you were doing when the error occurred. Submitting this form helps us improve the quality of our site, especially for users with your browser.

The following information will be submitted:
URL: http://www.yourdomain.com/throwAnError.html
Line: 34
Error: Object doesn't support this property or method
Your browser: Mozilla/4.0 (compatible; MSIE 5.5; Windows 98;)
Additional Comments:
I loaded the page and it immediately sent me to this page (I didn't touch anything, I swear!)

Submit to webmaster

FIGURE 23-9 Automatic error reporting with the **onerror handler**

The Error Object

When an exception is thrown, information about the exception is stored in an **Error** object. The structure of this object varies from browser to browser, but its most interesting properties and their support are described in Table 23-4.

The **Error()** constructor can be used to create an exception of a particular type. The syntax is

var *variableName* = new Error(*message*);

where *message* is a string indicating the *message* property that the exception should have. Unfortunately, support for the argument to the **Error()** constructor in Internet Explorer 5 and some early versions of 5.5 is particularly bad, so you might have to set the **message** property manually, such as:

L 23-19
```
var myException = new Error("Invalid data entry");
myException.message = "Invalid data entry";
```

Property	IE5?	IE5.5+?	Mozilla/NS6+?	ECMA?	Description
description	Yes	Yes	No	No	String describing the nature of the exception.
fileName	No	No	Yes	No	String indicating the URL of the document that threw the exception.
lineNumber	No	No	Yes	No	Numeric value indicating the line number of the statement that generated the exception.
message	No	Yes	Yes	Yes	String describing the nature of the exception.
Name	No	Yes	Yes	Yes	String indicating the type of the exception. ECMAScript values for this property are "EvalError," "RangeError," "ReferenceError," "SyntaxError," "TypeError," and "URIError."
Number	Yes	Yes	No	No	Number indicating the Microsoft-specific error number of the exception. This value can deviate wildly from documentation and from version to version.
Stack	No	No	Yes	No	String containing the call stack at the point the exception occurred.

TABLE 23-4 Properties of the **Error Object Vary from Browser to Browser**

You can also create instances of the specific ECMAScript exceptions given in the *name* row of Table 23-4. For example, to create a syntax error exception you might write

L 23-20
```
var myException = new SyntaxError("The syntax of the statement was invalid");
```

However, in order to keep user-created exceptions separate from those generated by the interpreter; it is generally a good idea to stick with **Error** objects unless you have a specific reason to do otherwise.

try, catch, and throw

Exceptions are caught using the **try/catch** construct. The syntax is

```
try {
  statements that might generate an exception
} catch (theException) {
  statements to execute when an exception is caught
} finally {
  statements to execute unconditionally
}
```

If a statement in the **try** block throws an exception, the rest of the block is skipped and the **catch** block is immediately executed. The **Error** object of the exception that was thrown is placed in the "argument" to the **catch** block (*theException* in this case, but any identifier will do). The *theException* instance is accessible only inside the **catch** block and should not be a previously declared identifier. The **finally** block is executed whenever the **try** or **catch** block finishes and is used in other languages to perform clean-up work associated with the statements that were tried. However, because JavaScript performs garbage collection, the **finally** block isn't generally very useful.

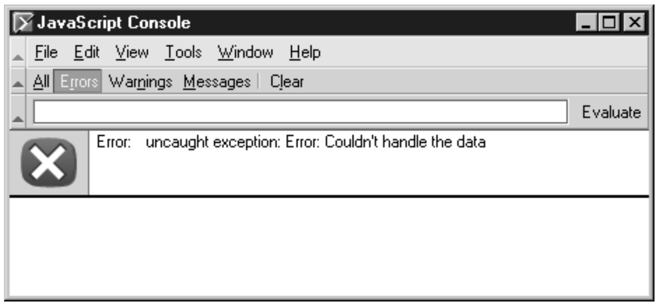
Note that the **try** block must be followed by exactly one **catch** or one **finally** (or one of both), so using **try** by itself or attempting to use multiple **catch** blocks will result in a syntax error. However, it is perfectly legal to have nested **try/catch** constructs, as in the following:

L 23-21
```
try {
   // some statements to try
   try {
      // some statements to try that might throw a different exception
   } catch(theException) {
      // perform exception handling for the inner try
   }
} catch (theException) {
   // perform exception handling for the outer try
}
```

Creating an instance of an **Error** does not cause the exception to be thrown. You must explicitly throw it using the **throw** keyword. For example, with the following,

L 23-22
```
var myException = new Error("Couldn't handle the data");
throw myException;
```

the result in Mozilla's JavaScript Console is

Ill 23-3

JavaScript Console
File Edit View Tools Window Help
All Errors Warnings Messages Clear
Evaluate

```
Error:   uncaught exception: Error: Couldn't handle the data
```

In Internet Explorer with debugging turned on, a similar error is reported.

NOTE *You can **throw** any value you like, including primitive strings or numbers, but creating and then throwing an **Error** instance is the preferable strategy.*

To illustrate the basic use of exceptions, consider the computation of a numeric value as a function of two arguments (mathematically inclined readers will recognize this as an identity for sine(a + b)). Using previously discussed defensive programming techniques, we could explicitly type-check or convert the arguments to numeric values in order to ensure a valid computation. We choose to perform type checking here using exceptions (and assuming, for clarity, that the browser has already been determined to support JavaScript exceptions):

L 23-23
```
function throwMyException(message)
{
  var myException = new Error(message);
  throw myException;
}
function sineOf(a, b)
{
   var result;
   try
   {
      if (typeof(a) != "number" || typeof(b) != "number")
         throwMyException("The arguments to sineOf() must be numeric");
      if (!isFinite(a) || !isFinite(b))
         throwMyException("The arguments to sineOf() must be finite");
      result = Math.sin(a) * Math.cos(b) + Math.cos(a) * Math.sin(b);
      if (isNaN(result))
         throwMyException("The result of the computation was not a number");
      return result;
   } catch (theException) {
      alert("Incorrect invocation of sineOf(): " + theException.message);
   }
}
```

Invoking this function correctly, for example,

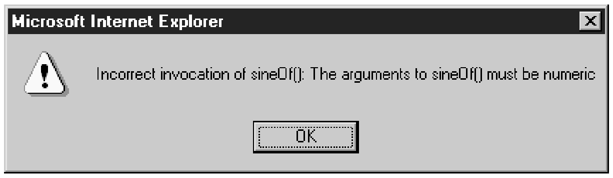
L 23-24
```
var myValue = sineOf(1, .5);
```

returns the correct value, but an incorrect invocation,

L 23-25
```
var myValue = sineOf(1, ".5");
```

results in an exception, in this case:

Ill 23-4


Microsoft Internet Explorer

Incorrect invocation of sineOf(): The arguments to sineOf() must be numeric

Exceptions in the Real World

Exceptions are the method of choice for notification of and recovery from problematic conditions, but the reality is that they are not well supported even in many modern Web browsers. To accommodate the non-ECMAScript **Error** properties of Internet Explorer 5.*x* and Netscape 6, you will probably have to do some sort of browser detection in order to extract useful information. While it might be useful to have simple exception handling, such as

L 23-26

```
try {
   // do something IE or Netscape specific
} catch (theException) {
}
```

that is designed to mask the possible failure of an attempt to access proprietary browser features, the real application of exceptions at the current moment is to Java applets and the DOM.

By enclosing potentially dangerous code such as LiveConnect calls to applets and the invocation of DOM methods in **try/catch** constructs, you can bring some of the robustness of more mature languages to JavaScript. However, using exception handling in typical day-to-day scripting tasks is probably still a few years in the future. For the time being, JavaScript's exception handling features are best used in situations where some guarantee can be made about client capabilities—for example, by applying concepts from the following two sections. Use them if you can guarantee that your users' browsers support them; otherwise, they're best avoided.

Capability and Browser Detection

We've seen some examples of capability and browser detection throughout the book, but there remain a few relevant issues to discuss. To clarify terminology in preparation for this discussion, we define *capability detection* as probing for support for a specific object, property, or method in the user's browser. For example, checking for **document.all** or **document.getElementById** would constitute capability detection. We define *browser detection* as determining which browser, version, and platform is currently in use. For example, parsing the **navigator.userAgent** would constitute browser detection.

Often, capability detection is used to infer browser information. For example, we might probe for **document.layers** and infer from its presence that the browser is Netscape 4.*x*. The other direction holds as well: Often capability assumptions are made based upon browser detection. For example, the presence of "MSIE 6.0" and "Windows" in the **userAgent** string might be used to infer the ability to use JavaScript's exception handling features.

When you step back and think about it, conclusions drawn from capability or browser detection can easily turn out to be false. In the case of capability detection, recall from Chapter 17 that the presence of **navigator.plugins** in no way guarantees that a script can probe for support for a particular plug-in. Internet Explorer does not support plug-in probing, but defines **navigator.plugins[ ]** anyway as a synonym for **document.embeds[ ]**. Drawing conclusions from browser detection can be equally as dangerous. Although Opera has the capability to masquerade as Mozilla or Internet Explorer (by changing its **userAgent** string), both Mozilla and Internet Explorer implement a host of features not found in Opera.

While it is clear that there are some serious issues here that warrant consideration, it is not clear exactly what to make of them. Instead of coming out in favor of one technique over another, we list some of the pros and cons of each technique and suggest that a combination of both capability and browser detection is appropriate for most applications.

The advantages of capability detection include

- You are free from writing tedious case-by-case code for various browser version and platform combinations.
- Users with third-party browsers or otherwise alternative browsers (such as text browsers) will be able to take advantage of functionality that they would otherwise be prevented from using because of an unrecognized **userAgent** (or related) string. Capability detection is "forward safe" in the sense that new browsers emerging in the market will be supported without changing your code, so long as they support the capabilities you utilize.

Disadvantages of capability detection include

- The appearance of a browser to support a particular capability in no way guarantees that that capability functions the way you think it does. For example, consider that **navigator.plugins[ ]** in Internet Explorer is available but does not provide any data.
- The support of one particular capability does not necessarily imply support for related capabilities. For example, it is entirely possible to support **document.getElementById()** but not support **Style** objects. The task of verifying each capability you intend to use can be rather tedious.

The advantage of browser detection includes

- Once you have determined the user's browser correctly, you can infer support for various features with relative confidence, without having to explicitly detect each capability you intend to use.

The disadvantages of browser detection include

- Support for various features often varies widely across platforms, even in the same version of the browser (for example, DHTML Behaviors are not supported in Internet Explorer across platforms as the Mac OS does not implement them).
- You must write case-by-case code for each browser or class of browsers that you intend to support. As new versions and browsers continue to hit the market, this prospect looks less and less attractive.
- Users with third-party browsers may be locked out of functionality their browsers support simply by virtue of an unrecognized **userAgent**.
- Browser detection is not necessarily "forward safe." That is, if a new version of a browser or an entirely new browser enters the market, you will in all likelihood be required to modify your scripts to accommodate the new **userAgent**.
- There is no guarantee that a valid **userAgent** string will be transmitted.
- There is no guarantee that the **userAgent** value is not falsified.

The advent of the DOM offers hope for a simplification of these issues. At the time of this edition's publication (2004), more than 75 percent of users have browsers that support most if not all commonly used DOM0 and DOM1 features (Internet Explorer 6+, Netscape 6+, Mozilla 1+). While this number will increase, there's no guarantee that your users will be "average." Additionally, if your site must be maximally compatible with your user base (e.g., you're running an e-commerce site), you have no choice but to do some sort of capability or browser detection to accommodate old browsers.

We offer the following guidelines to help you make your decisions:

- Standard features (such as DOM0 and DOM1) are probably best detected using capabilities. This follows from the assumption that support for standards is relatively useless unless the entire standard is implemented. Additionally, it permits users with third-party standards-supporting browsers the use of such features without the browser vendor having to control the market or have their **userAgent** recognized.
- Support for proprietary features is probably best determined with browser detection. This follows from the fact that such features are often difficult to capability-detect properly and from the fact that you can fairly easily determine which versions and platforms of a browser support the features in question.

These guidelines are not meant to be the final word in capability versus browser detection. Careful consideration of your project requirements and prospective user must factor into the equation in a very significant way. Whatever your choice, it is important to bear in mind that there is another tool you can add to your defensive programming arsenal for accomplishing the same task.

Code Hiding

Browsers are supposed to ignore the contents of **<script>** tags with **language** or **type** attributes that they do not recognize. We can use this to our advantage by including a cascade of **<script>**s in the document, each targeting a particular language version. The **<script>** tags found earlier in the markup target browsers with limited capabilities, while those found later in sequence can target increasingly specific, more modern browsers.

The key idea is that there are two kinds of code hiding going on at the same time. By enclosing later scripts with advanced functionality in elements with appropriate **language** attributes (for example, "JavaScript1.5"), their code is hidden from more primitive browsers because these scripts are simply ignored. At the same time, the more primitive code can be hidden from more advanced browsers by replacing the old definitions with new ones found in later tags.

To illustrate the concept more clearly, suppose we wanted to use some DOM code in the page when the DOM is supported, but also want to degrade gracefully to more primitive non-standard "DHTML" functionality when such support is absent. We might use the following code, which redefines a *writePage()* function to include advanced functionality, depending upon which version of the language the browser supports:

L 23-27
```
<script language="Javacript">
<!—
function writePage()
```

```
{
   // code to output primitive HTML and JavaScript for older browsers
}
// -->
</script>
<script language="Javacript1.3">
<!—
function writePage()
{
   // code to output more advanced HTML and JavaScript that utilizes the DOM}
}
// -->
</script>
<script language="Javacript">
<!--
// actually write out the page according to which writePage is defined
writePage();
// -->
</script>
```

Because more modern browsers will parse the second **<script>**, the original definition of *writePage()* is hidden. Similarly, the second **<script>** will not be processed by older browsers, because they do not recognize its **language** attribute.

NOTE *While the **language** attribute is considered non-standard, you can see that it is much more flexible than the standard **type** attribute and thus the attribute continues to be used widely.*

If you keep in mind the guidelines for the **language** attributes given in Table 23-5, you can use this technique to design surprisingly powerful cascades (as will be demonstrated momentarily).

NOTE *Opera 3 parses any **<script>** with its **language** attribute beginning with "JavaScript."*

language Attribute	Supported By
Jscript	All scriptable versions of Internet Explorer and Opera 5+
JavaScript	All scriptable versions of Internet Explorer, Opera, and Netscape
JavaScript1.1	Internet Explorer 4+, Opera 3+, Mozilla, and Netscape 3+
JavaScript1.2	Internet Explorer 4+, Opera 3+, Mozilla, and Netscape 4+
JavaScript1.3	Internet Explorer 5+, Opera 4+, Mozilla, and Netscape 4.06+
JavaScript1.5	Opera 5+, Mozilla, and Netscape 6+

TABLE 23-5 The **language Attributes Recognized by Major Browsers**

To glimpse the power that the **language** attribute affords us, suppose that you wanted to include separate code for ancient browsers, Netscape 4, Mozilla, and Internet Explorer 4+. You could do so with the following:

L 23-28

```
<script language="JScript">
<!--
// set a flag so we can differentiate between Netscape and IE later on

var isIE = true;
// -->
</script>
<script language="Javacript">
<!--
function myFunction()
{
   // code to do something for ancient browsers
}
// -->
</script>
<script language="Javacript1.2">
<!--
if (window.isIE)
{
   function myFunction()
   {
      // code to do something specific for Internet Explorer 4+
   }
}
else
{
  function myFunction()
   {
      // code to do something specific for Netscape 4
   }
}
// -->
</script>
<script language="Javacript1.5">
<!--
function myFunction()
{
   // code to do something specific for Mozilla and Opera 5+
}
// -->
</script>
<noscript>
  <strong>Error:</strong>JavaScript not supported
</noscript>
```

We've managed to define a cross-browser function, *myFunction()*, for four different browsers using only the **language** attribute and a little ingenuity! Combined with some simple browser detection, this technique can be very powerful indeed.

NOTE *Always remember the **language** attribute is deprecated under HTML 4, so don't expect your pages to validate as strict HTML 4 or XHTML when using this trick. The upside is that all modern browsers continue to support the attribute even though it is no longer officially a part of the language.*

Remember that it is always good style to include **<noscript>**s for older browsers or browsers in which JavaScript has been disabled. We provided a very basic example of **<noscript>** here, but if we followed very defensive programming styles each piece of code in this book should properly have been followed by a **<noscript>** indicating that JavaScript is required or giving alternative functionality for the page or indicating that a significant error has occurred. We omitted such **<noscript>**s in most cases for the sake of brevity and clarity, but we would always include them in a document that was live on the Web. See Chapter 1 for a quick **<noscript>** refresher. We now turn our attention towards general practices that are considered good coding style.

Coding Style

Because of the ease with which JavaScript can be used for a variety of tasks, developers often neglect good coding style in the rush to implement. Doing so often comes back to haunt them when later they are faced with mysterious bugs or code maintenance tasks and cannot easily decipher the meaning or intent of their own code. Practicing good coding habits can reduce such problems by bringing clarity and consistency to your scripts.

While we have emphasized what constitutes good coding style throughout the book, we summarize some of the key aspects in Table 23-6. We cannot stress enough how important good style is when undertaking a large development project, but even for smaller projects good style can make a serious difference. The only (possible) time you might wish to take liberties with coding style is when compressing your scripts for speed, but then again you might want to let tools do that for you and write nice descriptive code for yourself.

Aspect of JavaScript	Recommendation
Variable identifiers	Use camel-back capitalization and descriptive names that give an indication of what value the variable might be expected to hold. Appropriate variable names are most often made up of one or more nouns.
Function identifiers	Use the camel-back capitalization and descriptive names that indicate what operation they carry out. Appropriate function names are most often made up of one or more verbs.
Variable declarations	Avoid implicitly declared variables as they clutter the global namespace and lead to confusion. Always use **var** to declare your variables in the most specific scope possible. Avoid global variables whenever possible.
Functions	Pass values that need to be modified by reference by wrapping them in a composite type. Or, alternatively, return the new value that the variable should take on. Avoid changing global variables from inside functions. Declare functions in the document **<head>** or in a linked .js library.

TABLE 23-6 Good Coding Style Guidelines

Aspect of JavaScript	Recommendation
Constructors	Indicate that object constructors are such by capitalizing the first letter of their identifier.
Comments	Use comments liberally. Complex conditionals should always be commented and so should functions.
Indentation	Indent each block two to five spaces further than the enclosing block. Doing so gives visual cues as to nesting depth and the relationship between constructs like **if/else**.
Modularization	Whenever possible, break your scripts up into externally linked libraries. Doing so facilitates code reuse and eases maintenance tasks.
Semicolons	Use them. Do not rely on implicit semicolon insertion.

TABLE 23-6 Good Coding Style Guidelines *(continued)*

Speeding up Your Code

There are a variety of ways in which developers try to decrease the time it takes to download and render their pages. The most obvious is *crunching*, which is the process of removing excess whitespace in files (since it is collapsed or ignored by the browser anyway) and replacing long identifiers with shorter ones. The assumption is that there will be fewer characters to transfer from the server to the client, so download speed should increase proportionally. There are many tools available on the Web that perform crunching, and the capability may be packaged with commercial development systems as well.

Some tools such as the W3Compiler (**www.w3compiler.com**) take crunching to the next level. Not only do they perform whitespace removal, but they apply code transformations to JavaScript, CSS, and HTML while preserving the logic and functionality of the page. Special optimization tools like this one may even rearrange your code and combine scripts into external .js files or even inline it as one large **<script>** block depending on the performance considerations of the page. All these types of techniques attempt to reduce code size to improve download time, but don't forget about runtime optimizations. If your script performs lots of manipulation of objects or the page's DOM, consider firing up the Venkman debugger and profiling your code to look for ways to improve runtime execution.

Protecting Your Code

If you are concerned with people stealing your scripts for use on their own sites, then you probably should not be implementing in JavaScript. Because of JavaScript's nature as an interpreted language included directly in (X)HTML documents, your users have unfettered access to your source code, at least in the current Web paradigm. While you might be able to hide code from naïve users by placing it in externally linked .js files, doing so will certainly not deter someone intent upon examining or "borrowing" your code. Just because the JavaScript is not included inline in the page does not mean that it is inaccessible. It is very easy to load an external .js library into a debugger, retrieve from your browser's cache, or download it using your browser using a direct URL.

A partial solution to protecting your JavaScript code is offered by code *obfuscators*. Obfuscators read in JavaScript (or a Web page) and output a functionally equivalent version of the code that is scrambled (presumably) beyond recognition. Obfuscators are often included with crunchers, but there are numerous stand-alone obfuscators available on the Web. Be careful though: good obfuscation often comes at the expense of good crunching. Really hard-to-decipher code might even be bigger than the original code! To illustrate the idea, we use an obfuscator on the following snippet of HTML and JavaScript:

L 23-29
```
<a href="#" onclick="alert('No one must know this secret!')">This is a
secret link!</a>
```

The result from an obfuscator might be

L 23-30
```
<script type="text/javascript">var
enkripsi="$2B`$31isdg$2E$33$32$33$31nobmhbj$2E$33`mdsu$39$36On$31nod$31ltru$31jonv
$31uihr$31rdbsdu$30$36$38$33$2DUihr$31hr$31`$31rdbsdu$31mhoj$30$2B.`$2D"; teks="";
teksasli="";var panjang;panjang=enkripsi.length;for (i=0;i<panjang;i++)
teks+tring.fromCharCode(enkripsi.charCodet(i)1)
teksasliunescape(teks);document.write(teksasli);</script>
```

This obfuscated code replaces the original code in your document and, believe it or not, works entirely properly, as shown in Figure 23-10.

There are a few downsides with using obfuscated code. The first is that often the obfuscation increases the size of the code substantially, so obscurity comes at the price of download speed. Second, although code obfuscation might seem like an attractive route, you should be aware that reversing obfuscation is always possible. A dedicated and clever adversary will eventually be able to "undo" the obfuscation to obtain the original code (or a more tidy functional equivalent) no matter what scrambling techniques you might apply.

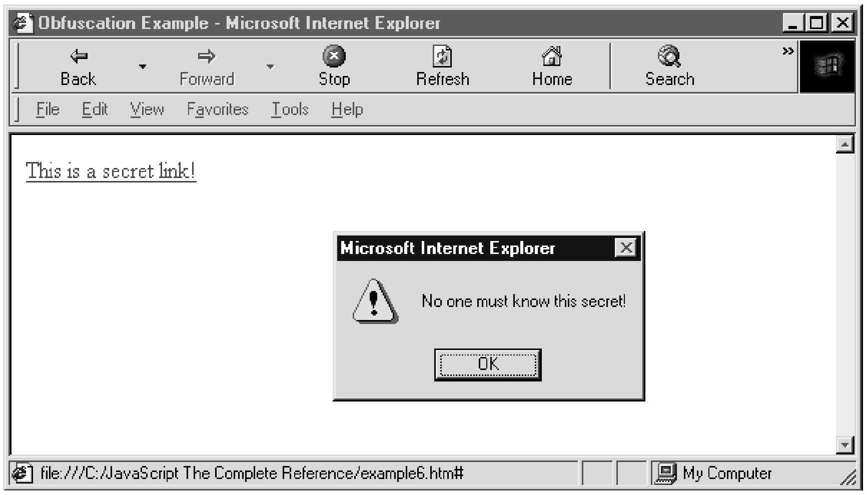

Obfuscation Example - Microsoft Internet Explorer
Back
Forward
Stop
Refresh
Home
Search
File
Edit
View
Favorites
Tools
Help
This is a secret link!
Microsoft Internet Explorer
No one must know this secret!
OK
file:///C:/JavaScript The Complete Reference/example6.htm#
My Computer

FIGURE 23-4 Obfuscated code is functionally equivalent to the original.

Still, obfuscation can be a useful tool when you need to hide functionality from naïve or unmotivated snoopers. It certainly is better than relying on external .js files alone.

NOTE *Many developers refer to obfuscation as "encryption." While doing so is likely to make a cryptographer cringe, the term is in widespread use. It is often helpful to use "encryption" instead of "obfuscation" when searching the Web for these kinds of tools.*

NOTE *Microsoft Script Engine 5+ comes with a feature that allows you to encrypt your scripts. Encrypted scripts can be automatically decrypted and used by Internet Explorer 5+. However, this technology is available only for Internet Explorer, so using it is not a recommendable practice.*

Paranoid developers might wish to move functionality that must be protected at all costs into a more appropriate technology, perhaps a plug-in, ActiveX control, or Java applet. However, doing so doesn't really solve the problem either, because both binaries and bytecode are successfully reverse-engineered on a regular basis. It does, however, put the code out of reach for the vast majority of potential thieves.

Summary

JavaScript errors come in many flavors from simple *syntax errors* that might be simple types to intermittent errors related to download or even semantic errors that produce results unintended by the programmer. To catch errors, JavaScript programmers employ typical debugging techniques such as turning on error messages and outputting verbose status information to track down logical errors, but a better approach is to use a program designed specifically for the task, a *debugger*. Yet like any programmer, JavaScript professionals should always assume errors will occur and employ defensive programming to address errors that may occur. Code hiding, exception handling, and simple ideas like the **<noscript>** tag should be part of every JavaScript developer's arsenal. Yet all the while that JavaScript programmers try to employ good coding practices to improve the quality and maintainability of their code, they may find these practices often fly in the face of performance and security. Tools to "crunch" code to improve download or to obfuscate source to protect from casual snoops are certainly a good idea for complex scripts, but developers need to remember that the determined thief can thwart just about any effort they make. As JavaScript matures certainly programming practices will as well.