The Complete Reference

JavaScript

Chapter 24

JavaScript Practices

In the previous two chapters we covered some of the proprietary issues involving the Netscape and Internet Explorer browsers. While an awareness of the features and quirks of each major browser and version is required to write well-behaved code, it is not a sufficient condition. You need to be able to apply what you know in an effective manner in order to accommodate the needs of your users, fellow programmers, and future script maintainers.

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 applications. The wide variety of errors that can occur during the execution of a script can be roughly placed into three categories: syntax errors, semantic errors, and runtime errors.

Of the three types of errors, s*yntax errors* are the most obvious and are the result of code that somehow violates the rules of the language itself. For example, writing the following:

```
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:

```
var myString = "This string doesn't terminate
```

because the string literal never terminates.

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 an unterminated 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 "dead," 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:

```
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 or compilation 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.

Errors of the second type, *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 type of semantic errors are those caused by JavaScript's weak typing; for example:

```
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 assignment 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:

```
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()** function.

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).

In the final category are the *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,

```
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,

```
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,

```
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.

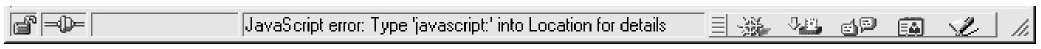
Debugging

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 an approach that works for him or her. It is because of the individual nature of debugging that most software development groups have a language expert, whether formally or informally designated, to whom elusive problems and anomalies can be taken for resolution. This individual is often a veteran developer with an ability for tracking down problems honed by years of software development experience. Experience is indeed the greatest teacher when it comes to debugging.

Turning on Error Messages

The most basic way to track down errors is by turning on error information in your browser. This is accomplished in Internet Explorer by using the Tools menu, selecting Internet Options, and activating the Advanced tab. Make sure that the "Disable script debugging" box is unchecked and that the "Display a notification about every script error" box is checked, as shown in Figure 24-1.

Netscape 3 shows all JavaScript errors to the user by default, but Netscape 4+ sends them to the JavaScript Console. To view the Console in Netscape 4.x, type **javascript:** in the browser's Location bar. In Netscape 6, use the Tasks menu to open the Tools submenu, and then select "JavaScript Console." By default, the only indication that an error has occurred in Netscape 4 is a message on the status bar:

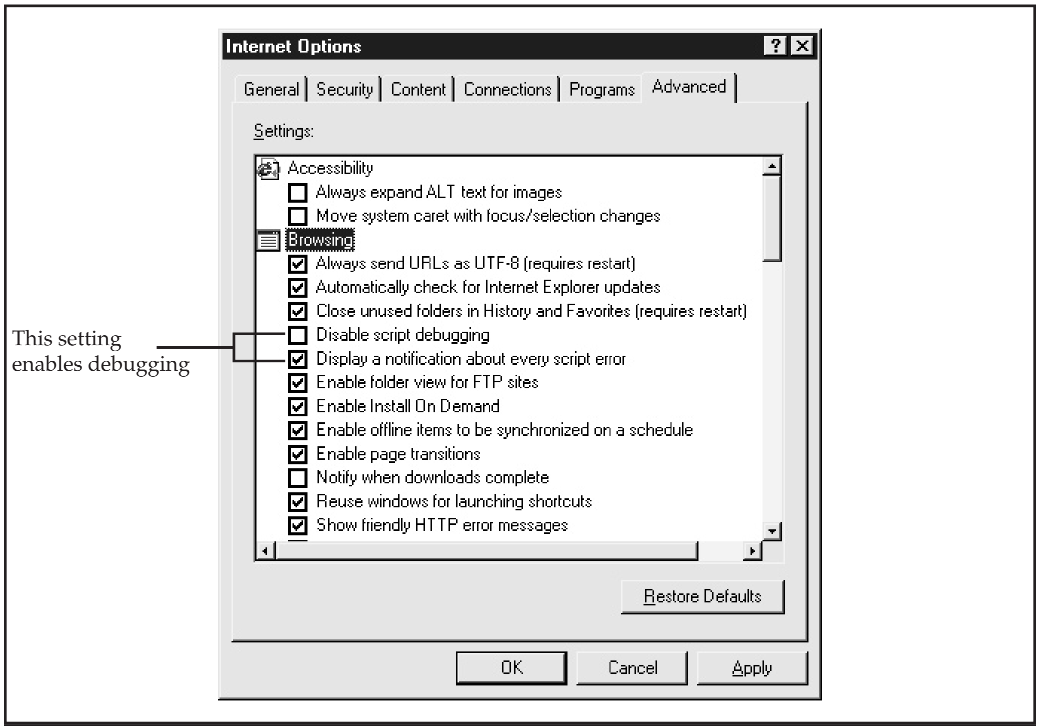


```
JavaScript error: Type 'javascript:' into Location for details
```

However, it is possible to configure Netscape 4 to bring up the JavaScript Console whenever an error occurs. To do so follow these steps:

1. Quit the browser.
2. Open the file prefs.js for editing. This file is usually found in the Netscape install directory (or in UNIX, it is often called preferences.js under ~/.netscape).
3. Add the following line to the file:
   user_pref("javascript.console.open_on_error", true);.
4. Save the file and close the editor.

Netscape 6 gives no indication when an error occurs, so you must keep the JavaScript Console open and watch for errors as your script executes.


Internet Options

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
☑ Enable offline items to be synchronized on a schedule
☑ Enable page transitions
☐ Notify when downloads complete
☑ Reuse windows for launching shortcuts
☑ Show friendly HTTP error messages

This setting enables debugging

Figure 24-1. *Enabling script errors in Internet Explorer*

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,

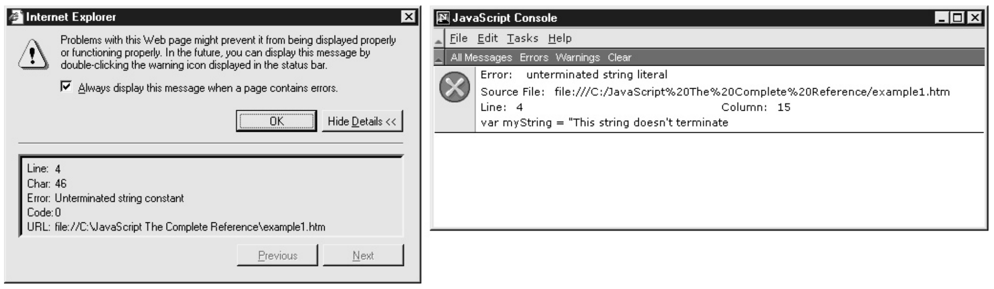
```
var myString = "This string doesn't terminate
```

results in the error dialog and JavaScript Console messages in Figure 24-2. 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 24-3.

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



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.

```
Line: 4
Char: 46
Error: Unterminated string constant
Code: 0
URL: file://C:\JavaScript The Complete Reference\example1.htm
```

```
Error:   unterminated string literal
Source File:  file:///C:/JavaScript%20The%20Complete%20Reference/example1.htm
Line:  4                                    Column:  15
var myString = "This string doesn't terminate
```

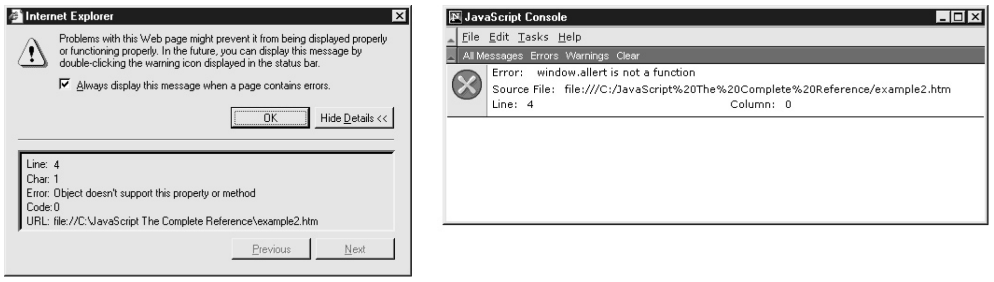
Figure 24-2. *An unterminated string constant syntax error in Internet Explorer and Netscape*

time these error messages are fairly easy to decipher, but some messages are less descriptive than others. It is therefore useful here to explicitly mention some common mistakes.

Common Mistakes

Table 24-1 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.

Using some sort of integrated development environment (IDE) that matches parentheses and that colors your code is often helpful in avoiding syntax errors.



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.

```
Line: 4
Char: 1
Error: Object doesn't support this property or method
Code: 0
URL: file://C:\JavaScript The Complete Reference\example2.htm
```

```
Error:   window.allert is not a function
Source File:  file:///C:/JavaScript%20The%20Complete%20Reference/example2.htm
Line:  4                                    Column:  0
```

Figure 24-3. *A runtime error in Internet Explorer and Netscape*

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.

Mistake	Example	Symptom
Infinite loops	while (*x*<myArray.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

Table 24-1. *Common JavaScript Mistakes*

Mistake	Example	Symptom

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 (as in this example)

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

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
Invoking undefined functions	noSuchFunction();	"object expected" or "*noSuchFunction* is not defined" error messages

Table 24-1. *Common JavaScript Mistakes* (continued)

Mistake	Example	Symptom
Accessing the document before it has finished loading	<head><script>var myElement=document.all.myElement;</script></head>	**undefined** values, broken DHTML, and errors associated with nonexistent properties and methods
Accessing a form element rather than its value	var *x* = document.myform.myfield;	Computation resulting in **NaN**, broken DHTML, 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 24-1. *Common JavaScript Mistakes* (continued)

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 in finding logical errors. There are, however, some widespread practices that many developers employ when trying to find the reason for malfunctioning code.

Outputting Debugging Information

One of the most common techniques is to output verbose status information throughout the script 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 **alert()**s; for example, you might write something like:

```
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);
```

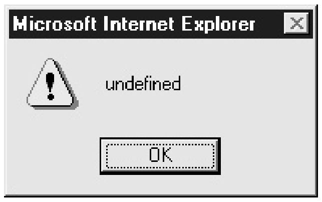
and include **alert()**s marking the flow of execution in *swapImages()*. By examining the content and order of the **alert()**s as they appear, you are granted a window to the internal state of your script.

Because using a large number of **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 is opened at the beginning of the script, and debugging information is written into the window using its **document.write()** method. For example, consider the following (erroneous) implementation of the **extractCookie()** method from Chapter 16 that has been instrumented with debugging statements. Of course, you can omit the HTML from the calls to **output()** below; we included them to ensure correct rendering. Note the **for** loop immediately after the **window.open()**. This loop kills some time before writing to the new window in order to avoid a bug with Internet Explorer that sometimes occurs when you attempt to **write()** to a newly opened empty window before it is done setting up.

```
var debugging = true;
if (debugging)
{
   var debuggingWindow = window.open("");

   // omitting the name causes a new window to open
   for (var x=0; x<1500000; x++);      // give IE time to open window
   output = debuggingWindow.document.writeln;
   output("<html><head><title>Debugging
Window</title></head><body><pre>");
   // we're going to be writing preformatted info, so include the <pre>
}
var cookies = new Object();    // associative array indexed as cookies["name"] = "value"
function extractCookies()
{  // extract current cookies, destroying old value of cookies array
   if (debugging) output("Beginning extractCookies on:\n" + document.cookie + "\n");
   var name, value, beginning, middle, end;
   beginning = 0;
   while (beginning < document.cookie.length)
    {
     if (debugging)
        output("Top of the loop (beginning = " + beginning + ")");
      middle = document.cookie.indexOf('=', beginning);
      end = document.cookie.indexOf(';', beginning);
      if (end == -1)
         end = document.cookie.length;
      if ( (middle > end) || (middle == -1) )
       {  // if no equal sign in this cookie...
         if (debugging) output("\tNo value for this cookie");
         name = document.cookie.substring(beginning, end);
         value = "";
       }
      else
       {
         name = document.cookie.substring(beginning, middle);
         value = document.cookie.substring(middle, end);
         if (debugging)
           output("\tExtracted cookie with name='"+name+"' and value='"+value+"'");
       }
      cookies[name] = unescape(value);
      beginning = end + 1;
    }
   if (debugging) output("\nExiting extractCookies()</pre></body></html>")
}
document.cookie = "username=fritz";
document.cookie = "favoritecolor=green";
document.cookie = "debuggingisfun=false";
extractCookies();
alert(cookies["favoritecolor"]);
```

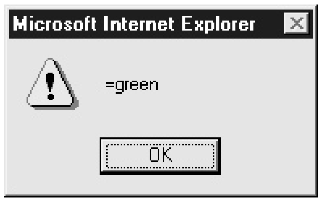
Running this code, it is apparent that there is some sort of error. The following alert is shown:



undefined

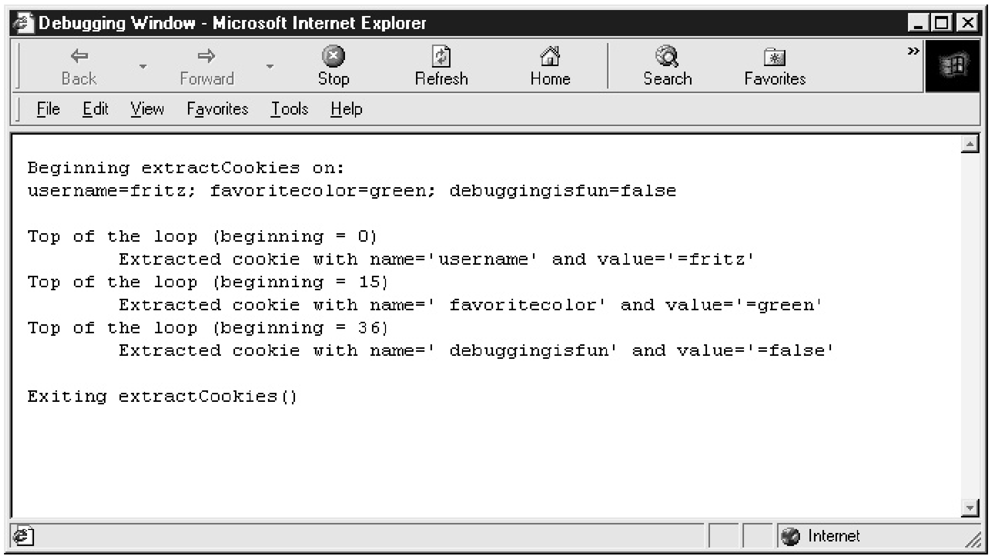
An examination of the output to the debugging window shown in Figure 24-4 reveals that the reason for this behavior is an extra space in the name of the cookies. Also, it appears as if the equal sign is being included in the value.

Tracing through the source code it looks like the line "beginning = end + 1" is the culprit. It appears as if *beginning* is being set to the character before the name of the next cookie (which is a space) rather than being set to the first character of the cookie name. So we change "beginning = end + 1" to "beginning = end + 2" in hopes that this will solve the problem. Doing so should point *beginning* past the space to the beginning of the cookie name. Running the script with this change results in:



```
=green
```

This is definitely progress, but is still incorrect. The debugging output is shown in Figure 24-5.

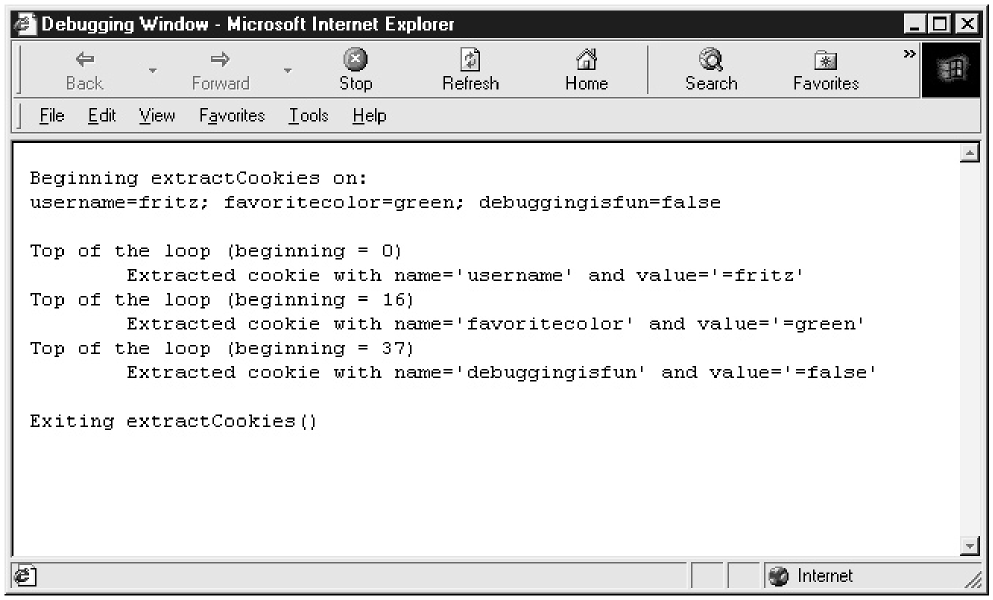


```
Beginning extractCookies on:
username=fritz; favoritecolor=green; debuggingisfun=false

Top of the loop (beginning = 0)
        Extracted cookie with name='username' and value='=fritz'
Top of the loop (beginning = 15)
        Extracted cookie with name=' favoritecolor' and value='=green'
Top of the loop (beginning = 36)
        Extracted cookie with name=' debuggingisfun' and value='=false'

Exiting extractCookies()
```

Figure 24-4. *Debugging output for the extractCookies() example*



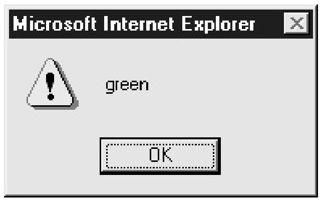
```
Beginning extractCookies on:
username=fritz; favoritecolor=green; debuggingisfun=false

Top of the loop (beginning = 0)
        Extracted cookie with name='username' and value='=fritz'
Top of the loop (beginning = 16)
        Extracted cookie with name='favoritecolor' and value='=green'
Top of the loop (beginning = 37)
        Extracted cookie with name='debuggingisfun' and value='=false'

Exiting extractCookies()
```

Figure 24-5. *Debugging out for the extractCookies() example after making the first correction*

It looks as though we have a similar problem with the cookie value. Tracing through the code yet again we find that the variable *middle* is set to the equal sign in the current cookie (if one exists). When we extract the cookie value into *value*, *middle* is passed as the start of the value substring. Changing "value = document.cookie.substring(middle, end)" to "value = document.cookie.substring(middle + 1, end)" should start the substring at the correct position. Making this change and running the script again results in:



```
green
```

This is correct. An examination of the debugging window shows that the script appears to be functioning correctly, so we can disable debugging. To do so, either the debugging statements can be removed from the script or *debugging* can be set to **false**.

While this example may seem somewhat contrived, it reflects the actual debugging process that went on when **extractCookies()** was written. Using **alert()**s or **write()**s to assist in tracking down problems is a very common technique, but as you can see it requires a bit of work. A better solution is to use a tool designed specifically for the task.

Using a Debugger

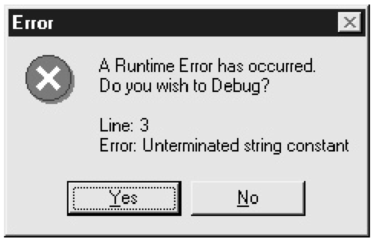
A *debugger* is an application that places all aspects of script execution under the control of the programmer. Debuggers provide fine-grain control over the state of the script through an interface that allows you to examine and set values as well as control the flow of execution.

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 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 a *trace* of the program, the call tree representing the flow of execution through various pieces of code. 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 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.

Note *Because you can type in **javascript:** pseudo-URLs in the Location bar of Netscape 4 and 6, you can use the browser itself as a primitive debugger. Netscape 4's JavaScript Console also allows you to type in arbitrary statements that are executed within the context of the current document. You can use these capabilities to examine or set values on a page—for example, with **alert()** or assignment statements.*

Clearly, debuggers sound like extremely useful tools. But where can they be found? You have three primary options. The first is the Netscape JavaScript debugger, a Java application that runs on top of Netscape 4.x and is implemented specifically with Netscape JavaScript and the Netscape family of browsers in mind. This debugger has the advantage of accommodating Netscape-centric technologies, like signed scripts and LiveConnect, but has the disadvantage of being a couple of generations out of date (no new version has appeared in the last several years). You can find more information about this free tool at **http://developer.netscape.com/software/jsdebug.html**.

Your second option is Microsoft Script Debugger, a free utility that integrates with Internet Explorer 4+ and is available from **http://msdn.microsoft.com/scripting**. Whenever debugging is turned on and you load a page that has errors, the following dialog is shown in place of the normal error message, allowing you to load the page into the debugger.



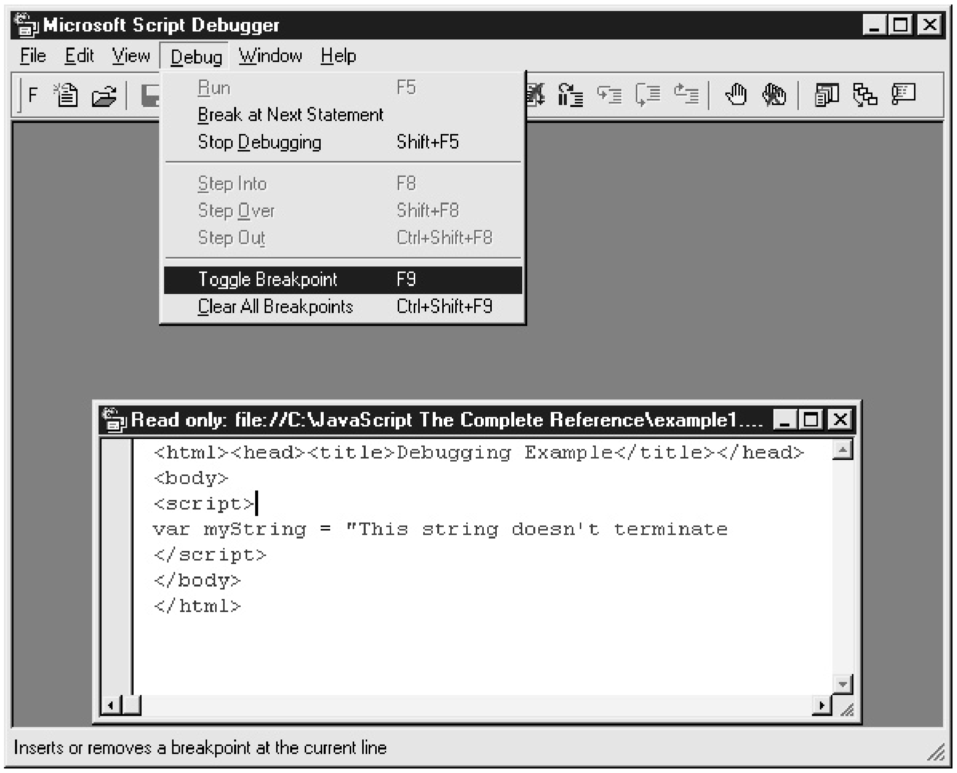
Error

A Runtime Error has occurred.
Do you wish to Debug?

Line: 3
Error: Unterminated string constant

Of course, you can also load a document directly into the debugger without having an error occur. This debugger has the advantage of close coupling with Microsoft's JScript and document object model and is much more current (though still a bit out of date) than Netscape's debugger. A screenshot of Microsoft Script Debugger is shown in Figure 24-6.

The third option you have is to use a commercial development environment. A JavaScript debugger is usually just one small part of such development tools, which



Read only: file://C:\JavaScript The Complete Reference\example1....

```
<html><head><title>Debugging Example</title></head>
<body>
<script>
var myString = "This string doesn't terminate
</script>
</body>
</html>
```

Inserts or removes a breakpoint at the current line

Figure 24-6. *Microsoft Script Debugger*

can offer sophisticated HTML and CSS layout capabilities and can even automate certain aspects of site generation. This option is probably 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/** (for purchase or free trial). 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.

Defensive Programming

Now that we have covered how you can recognize and track 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.

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 DHTML.
- 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 third-party, text-based, or aural browsers.
- Users on non-Windows platforms. This is especially important in light of the fact that Internet Explorer for MacOS lags significantly behind Internet Explorer for Windows in terms of functionality.
- 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 DHTML or 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.

In this section we discuss a variety of techniques you can use for defensive programming. While no single set of ideas or approaches will solve every problem that might be encountered, applying the following principles to your scripts can dramatically reduce the number of errors your clients encounter. Additionally, doing so can also help you solve those errors that are encountered in a more timely fashion. 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 24-2.

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

Value of window.onerror	Effect
null	Suppresses reporting of runtime errors in Netscape 3+.
A function which returns **true**	Executes the function whenever a runtime error occurs and suppresses the normal reporting of runtime errors.
A function which returns **false**	Executes the function whenever a runtime error occurs and reports the error as usual.

Table 24-2. *Effect of Setting window.onerror to Different Values*

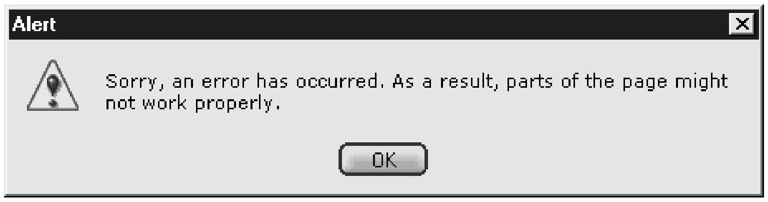
Because the display of JavaScript error messages can be unsettling to non-technical users, their display is often suppressed:

```
function doNothing() { return true; }
window.onerror = doNothing;
```

or replaced with a more benign message:

```
function reportError()
{
  alert("Sorry, an error has occurred. As a result, parts of the page might not work properly.");
  return true;
}
window.onerror = reportError;     // replace the default error functionality
window.noSuchProperty();          // throw a runtime error
```

The result in Netscape 6 is shown here:



Sorry, an error has occurred. As a result, parts of the page might not work properly.

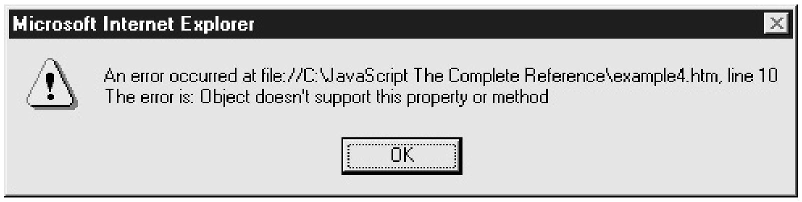
A useful feature of **onerror** handlers is that they are automatically passed three values by the browser. The first argument is a string containing an error message describing the error that occurred. The second is a string containing the URL of the page that generated the error, which might be different 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 *Netscape 6 does not currently pass these values to **onerror** handlers. This could be a bug, but it more likely represents a movement towards the exception-handling features described in the next section.*

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

```
function reportError(message, url, lineNumber)
{
 if (message && url && lineNumber)       // avoid netscape 6
   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:



An error occurred at file://C:\JavaScript The Complete Reference\example4.htm, line 10
The error is: Object doesn't support this property or method

However, a better 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:

```
<html>
<head>
<title>Error Submission</title>
<script language="JavaScript" 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="fillValues()">
<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="userAgent" id="userAgent" size="60"><br>
Additional Comments:<br>
<textarea name="comments" value="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:

```
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 24-7 shows the window opened as the result of the following runtime error:

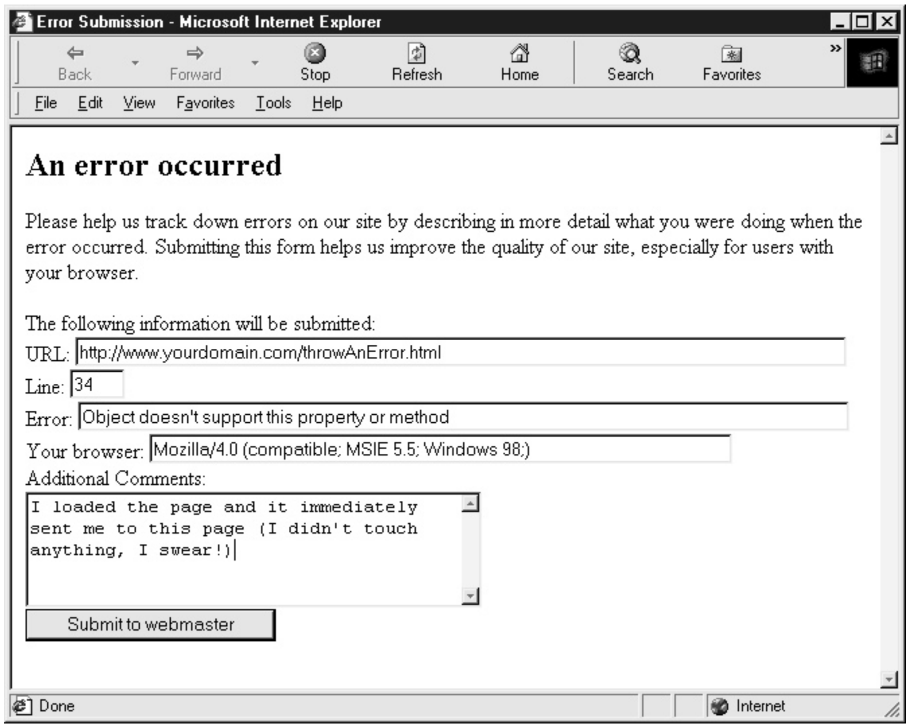
```
window.noSuchMethod();
```

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.

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 IE4.x and 5.5 appear to have complete support, some versions of IE5.0 might have problems.

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.



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 24-7. *errorReport.html opened as the result of an error*

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 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.

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 24-3.

Property	IE5?	IE5.5+?	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.

Table 24-3. *Properties of the Error Object*

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 very early versions of 5.5 is particularly bad, so you might have to set the property manually, such as:

```
var myException = new Error("Invalid data entry");
myException.message = "Invalid data entry";
```

You can also create instances of the specific ECMAScript exceptions given in the *name* row of Table 24-3. For example, to create a syntax error exception you might write

```
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 is essentially useless.

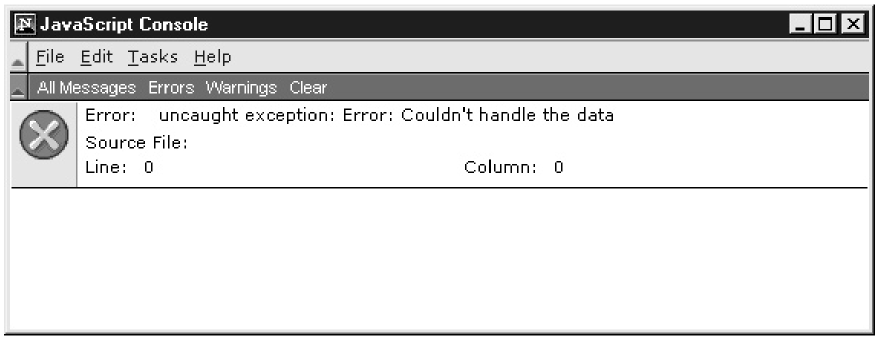
Note that the **try** block must be followed by exactly one **catch** or one **finally** (or one of both), so using **try** by itself or attempting to use multiple **catch** blocks will result in a syntax error. However, it is perfectly legal to have nested **try/catch** constructs, as in the following:

```
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,

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

the result in Netscape 6's JavaScript Console is



```
Error:   uncaught exception: Error: Couldn't handle the data
Source File:
Line:  0                                  Column:  0
```

In Internet Explorer 5.5 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):

```
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);
   }
}
```

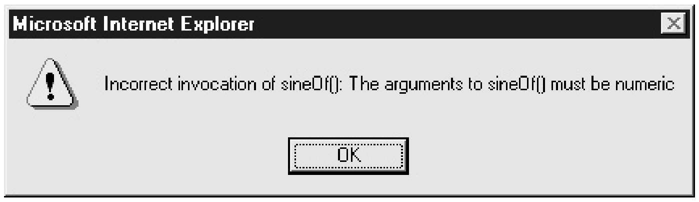
Invoking this function correctly, for example,

```
var myValue = sineOf(1, .5);
```

returns the correct value, but an incorrect invocation:

```
var myValue = sineOf(1, ".5");
```

results in an exception, in this case:



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

Exceptions in the real world

While exceptions will hopefully be the method of choice for notification of and recovery from problematic conditions in the future, the reality is that they are not well supported by today's Web browsers. To accommodate the non-ECMAScript **Error** properties of Internet Explorer 5 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:

```
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 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.

Capability and Browser Detection

We've seen some examples of capability and browser detection throughout the book, particularly in Chapter 17, and while we continue to assert that utilizing these techniques is good defensive programming, 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 5.5" 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 18 that the presence of **navigator.plugins** in no way guarantees that a script can probe for support for a particular plugin. Internet Explorer does not support plugin 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 IE 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. (Recall **navigator.plugins[ ]** in Internet Explorer).
- 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 advantages of browser detection include:

- 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 MacOS, even in Internet Explorer 5.5).

- 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.

While the advent of the DOM offers some hope for a simplification of these issues, it will not solve all of these problems. First, DOM-compliant browsers are currently being used by only a minority of the population. Second, DOM support in "DOM-compliant" browsers can be rife with errors and inconsistencies. Third, the vast majority of scripts in existence today are not written for the DOM, and it is unlikely that developer focus will wholeheartedly shift to the DOM in the near future. And, finally, browser vendors such as Microsoft bundle a very large number of proprietary features with the browser, necessitating some sort of detection if they are to be used.

Although this outlook may seem pretty bleak, there are some general guidelines you can follow. 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. Standard features 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.

These guidelines are not meant to be the final word in capability versus browser detection. There are some obvious exceptions to the rules, notably that the proprietary **document.layers[ ]** capability is an almost airtight guarantee that Netscape 4 is in use, and therefore that layers are properly supported. Careful consideration of your project requirements and prospective user base also 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** 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>**

elements found earlier in the cascade 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 DHTML code in the page when DHTML features are supported, but also want to degrade gracefully to more primitive 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:

```
<script language="JavaScript">
<!--
function writePage()
{
   // code to output primitive HTML and JavaScript for older browsers
}
// -->
</script>
<script language="JavaScript1.3">
<!--
function writePage()
{
   // code to output more advanced HTML and JavaScript that utilizes DHTML
   // or even the DOM
}
// -->
</script>
<script language="JavaScript">
<!--
   writePage();
   // write out the page according to which writePage is defined
// -->
</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.

If you keep in mind the guidelines for the **language** attributes given in Table 24-4, you can use this technique to design surprisingly powerful cascades (as will be

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+, and Netscape 3+
JavaScript1.2	Internet Explorer 4+, Opera 3+, and Netscape 4+
JavaScript1.3	Internet Explorer 5+, Opera 4+, and Netscape 4.06+
JavaScript1.5	Opera 5+ and Netscape 6+

Table 24-4. *Support for Value of the language attribute*

demonstrated momentarily). Note that Opera 3 parses any **<script>** with its **language** attribute beginning with "JavaScript."

To glimpse the power that the **language** attribute affords us, suppose that you wanted to include separate code for older browsers—Netscape 4, Netscape 6, and Internet Explorer 4+. You could do so with the following:

```
<script language="JScript">
<!--
   var isIE = true;
   // set a flag so we can differentiate between Netscape and IE later on
// -->
</script>
<script language="JavaScript">
<!--
function myFunction()
{
   // code to do something for older browsers
}
// --></script>
<script language="JavaScript1.2">
<!--
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 and others
    }
}
// -->
</script>
<script language="JavaScript1.5">
<!--
function myFunction()
{
   // code to do something specific for Netscape 6 and Opera 5+
}
// -->
</script>
```

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 *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.*

Accommodating Old Browsers

In our exploration of code hiding so far, we have glossed over several important issues that merit attention. First, you should keep in mind that the original purpose of the **language** attribute was to hide code utilizing new language features from older browsers. Therefore, you should make use of this feature whenever appropriate (and not just for cross-browser scripting tricks).

Another important thing to remember is that browsers that are not script-aware will display the contents of **<script>** tags as if they were HTML. Therefore, you should always hide your JavaScript inside HTML comments. Doing so suppresses the messy pages that can result from the JavaScript-as-HTML treatment that **<script>**s receive in very old or text-mode browsers.

And, finally, it is always good style to include **<noscript>**s for older browsers or browsers in which JavaScript has been disabled. Each piece of code in this book should properly have been followed by a **<noscript>** indicating that JavaScript is required or giving alternative HTML functionality for the page. 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. We turn our attention now 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 24-5. 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.

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 like the plague—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.

Table 24-5. *Recommended Good Coding Habits*

Aspect of JavaScript	Recommendation
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.
constructors	Indicate that object constructors are such by capitalizing the first letter of their identifier.
comments	Use them. 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 24-5. *Recommended Good Coding Habits* (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 is often packaged with commercial development studios as well.

Another, better approach is to move the bulk of your code into external .js libraries. Doing so permits the code to be cached by the browser, obviating the need to re-fetch the same JavaScript that would otherwise be included inline in each page.

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 them in JavaScript. Because of JavaScript's nature as an interpreted language included directly in 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 or to download it using a text-mode browser like Lynx. You can even telnet to port 80 of the Web server and issue an HTTP request for the file manually.

A partial solution is offered by code *obfuscators*, programs that 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 standalone obfuscators available on the Web.

To illustrate the technique, we use an obfuscator on the following snippet of HTML and JavaScript:

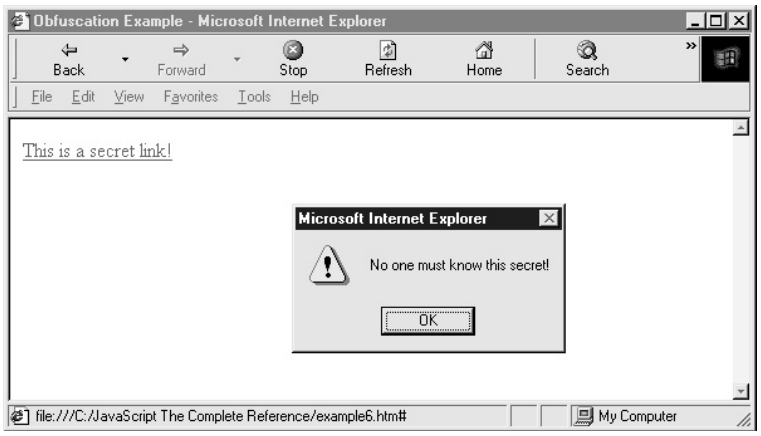
```
<a href="#" onclick="alert('No one must know this secret!')">This is a secret link!</a>
```

The result of obfuscation is

```
<script>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+=String.fromCharCode(enkripsi.charCodeAt(i)^1)
}teksasli=unescape(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 24-8.

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 size. 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. 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.


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
OK
file:///C:/JavaScript The Complete Reference/example6.htm#
My Computer

No one must know this secret!

Figure 24-8. *Obfuscated code works just like the original*

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 plugin, 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 Web users.

Probably the best solution is to accept the fact that people will probably peruse, play with, and occasionally reuse parts of your code from time to time. If you feel that you've been seriously burned, for example by someone who sells your code or publishes it for profit, you are most likely protected by the copyright laws in your state, province, or country. You should, however, obtain appropriate legal advice on this issue. At the very least, you might wish to include a copyright notice in your scripts and indicate the acceptable terms under which others can use your code.

Note *Most JavaScript programmers take a very liberal attitude towards code sharing. This might be the result of the fact that many people learn the language by reading others' code, but, whatever the case, it is definitely true that the majority of JavaScript programmers are happy to share code snippets and tips.*

JavaScript's Place on the Web

We began this book by listing some of the common uses for JavaScript on the Web. Now that you know a lot more about the language's capabilities and limitations, we return to this topic to discuss JavaScript's place on the Web in a bit more detail.

The rise of mobile code such as Java and JavaScript means that developers now have a choice about where they wish the computation in Web applications to take place. Functionality that was once the domain of CGI scripts can now be effectively carried out at either the client or the server side of the Web. There are, however, some natural guidelines for determining where functionality should be placed in order to maximize the strengths of various technologies.

Server-side technology is most often appropriate for data-intensive tasks such as content customization or database interaction. The reasons for this are numerous, but they boil down to security issues, computational power, and bandwidth concerns. Server machines typically have greater processing power than client machines and can be more closely coupled to data sources than can arbitrary clients. Additionally, well-founded security and privacy concerns dictate that access to a content database is probably best achieved by the server rather than the client. Sensitive calculations (especially in financial matters) should always be carried out on the server in order to minimize the risks posed by subversive clients. It would be all too easy to save an "e-commerce" JavaScript application to the local disk and modify its functionality to induce unintended behavior. For example, the price of items or the calculation of sales tax could be modified to give a malicious client the deal of the century. If you must carry out e-commerce–related tasks in client-side JavaScript, it is absolutely essential to include "sanity checking" on the server side to ensure that such malicious behavior cannot occur.

Client-side technologies are most appropriate for non-critical enhancements, such as form validation, minor document or presentation customizations, and creation of navigational aids. The reason that these applications are most often found on the client side is the close coupling of client-side technologies to Web documents themselves. Client-side features are always available to the user and document (once it's loaded in the browser). Server-side features, on the other hand, are unavailable once the document is sent from the server to the client (or, at the least, until the next request). It makes sense that tasks involving a high degree of user interaction are best placed on the client side. Form validation, presentation customization, and use of navigational aids are all tasks that involve significant user interaction. Keeping them on the client speeds things up by obviating the need for time-consuming communication with a server.

JavaScript fits into the server side of the Web by providing the means to automate document generation prior to transmission to the user. It faces a number of related, competing technologies such as Java servlets and other database-driven scripting languages like PHP and ColdFusion. One major advantage of server-side JavaScript over related technologies is the degree to which it is integrated with Windows and Windows Internet Information Server.

JavaScript fits into the client side of the Web as the language of choice for dynamic examination, generation, and manipulation of Web pages. Tremendous browser support and easy access to a page's document object model make any other choice for such tasks downright silly. While related technologies like plugins and ActiveX controls can provide more power than JavaScript scripts, they do so at the cost of increased separation from the browser and document objects and of decreased support.

JavaScript's capabilities continue to increase at a pace more rapid than most developers can keep up with. The W3C DOM facilities provide JavaScript with the unique position of being the most easily used language for manipulation and interaction with XML. JavaScript is also easy to learn and without a doubt, will continue to dominate the client-side scripting market for a long time to come. There is simply no competition even on the farthest horizon.

Summary

JavaScript *syntax errors* are those errors that result from code that does not conform to the rules of the language. Scripts generally can't recover from errors that introduce such ambiguity. On the other hand, *runtime* errors are those errors that occur as the result of attempting to utilize an unavailable resource—for example, the document object model before the page has loaded or an undefined object, property, method, or variable. Runtime errors can be caught on a **Window**-wide level in most modern browser by utilizing that object's **onerror** event handler. Runtime errors can also be caught as a part of JavaScript's more advanced exception-handling features.

Semantic errors occur when a JavaScript statement has an effect unintended by the programmer. Typical debugging techniques such as turning on error messages and outputting verbose status information can be used to track down logical errors, but a better approach is to use a program designed specifically for the task, a *debugger*. Defensive programming can help reduce both runtime and semantic errors.

Code hiding is an important aspect of defensive programming. Code hiding occurs when JavaScript is hidden from a browser that is not able to handle it. Although often carried out by utilizing the **language** attribute of the **<script>** element, it also occurs in more straightforward ways—for example, by commenting out the contents of a **<script>** to prevent old browsers from outputting the script as HTML.

An *exception* is a generalization of an error to include more benign and unexpected conditions, such as those arising from incorrect data types or malfunctioning embedded Java applets. Programmers can create and *throw* their own exceptions by instantiated **Error** objects used in conjunction with the **throw** keyword. Exceptions are *caught* (handled) using the **try/catch** construction, which permits the execution of a specific block of code if an exception occurs while executing the **try** statements.