PART I

Introduction

1
CHAPTER

Introduction to JavaScript

JavaScript is the premier client-side scripting language used today on the Web. It's widely used in tasks ranging from the validation of form data to the creation of complex user interfaces. Yet the language has capabilities that many of its users have yet to discover. JavaScript can be used to manipulate the very markup in the documents in which it is contained. As more developers discover its true power, JavaScript is becoming a first class client-side Web technology, ranking alongside (X)HTML, CSS, and XML. As such, it will be a language that any Web designer would be remiss not to master. This chapter serves as a brief introduction to the language and how it is included in Web pages.

NOTE *JavaScript can also be used outside of Web pages, for example, in Windows Script Host or for application development with Mozilla or Jscript.NET. We primarily focus on client-side JavaScript embedded in Web pages, but the core language is the same no matter where it is used; only the runtime environment (e.g., the browser objects discussed in Part 2) is different.*

First Look at JavaScript

Our first look at JavaScript is the ever-popular "Hello World" example. In this version, we will use JavaScript to write the string "Hello World from JavaScript!" into a simple XHTML transitional document to be displayed.

NOTE *XHTML is the most recent version of HTML. It reformulates HTML in terms of XML, bringing greater regularity to the language as well as an increased separation of logical structure from the presentational aspects of documents.*

L 1-1

```
<!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" lang="en">
<head>
<title>JavaScript Hello World</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
</head>
<body>
<h1 align="center">First JavaScript</h1>
```

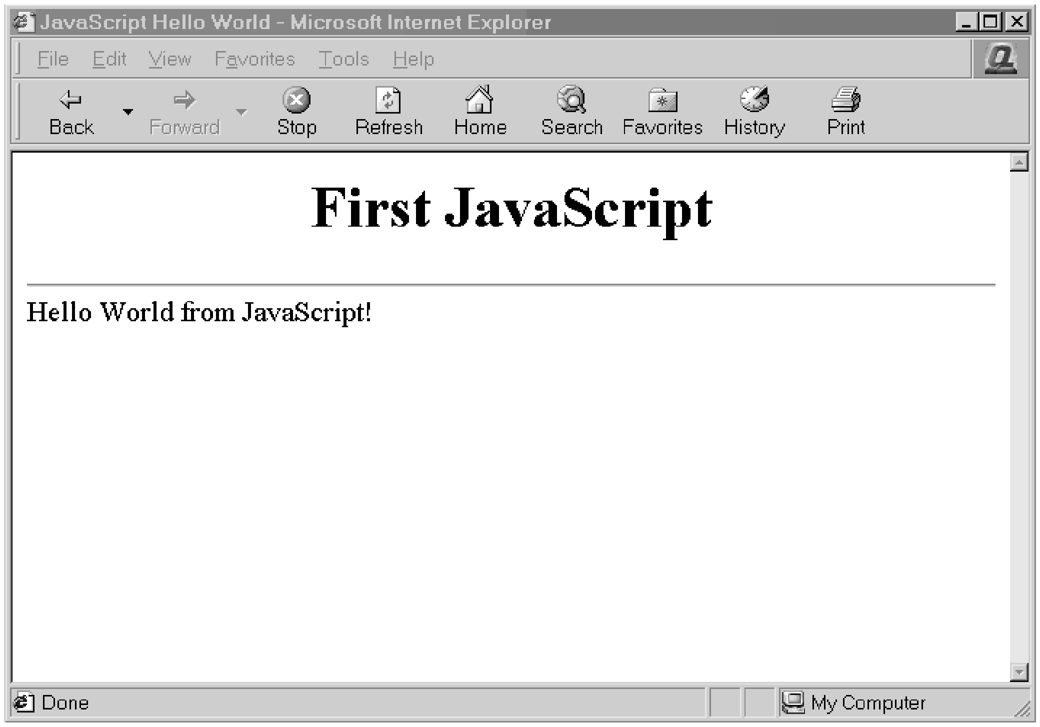
```
<hr />
<script type="text/javascript">
  document.write("Hello World from JavaScript!");
</script>
</body>
</html>
```

Notice how the script is included directly in the markup using the **<script>** element that encloses the simple one line script:

L 1-2
```
document.write("Hello World from JavaScript!");
```

Using the **<script>** element allows the browser to differentiate between what is JavaScript and what is regular text or (X)HTML. If we type this example in using any standard text editor, we can load it into a JavaScript-aware Web browser such as Internet Explorer, Netscape, Mozilla, Opera, or many others and we should see the result shown in Figure 1-1.

If we wanted to bold the text we could modify the script to output not only some text but also some markup. However, we need to be careful when the world of JavaScript and the world of markup in XHTML, or HTML, intersect—they are two different technologies. For example, consider if we substituted the following **<script>** block in the preceding document, hoping that it would emphasize the text:


JavaScript Hello World - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Print
Done
My Computer

First JavaScript

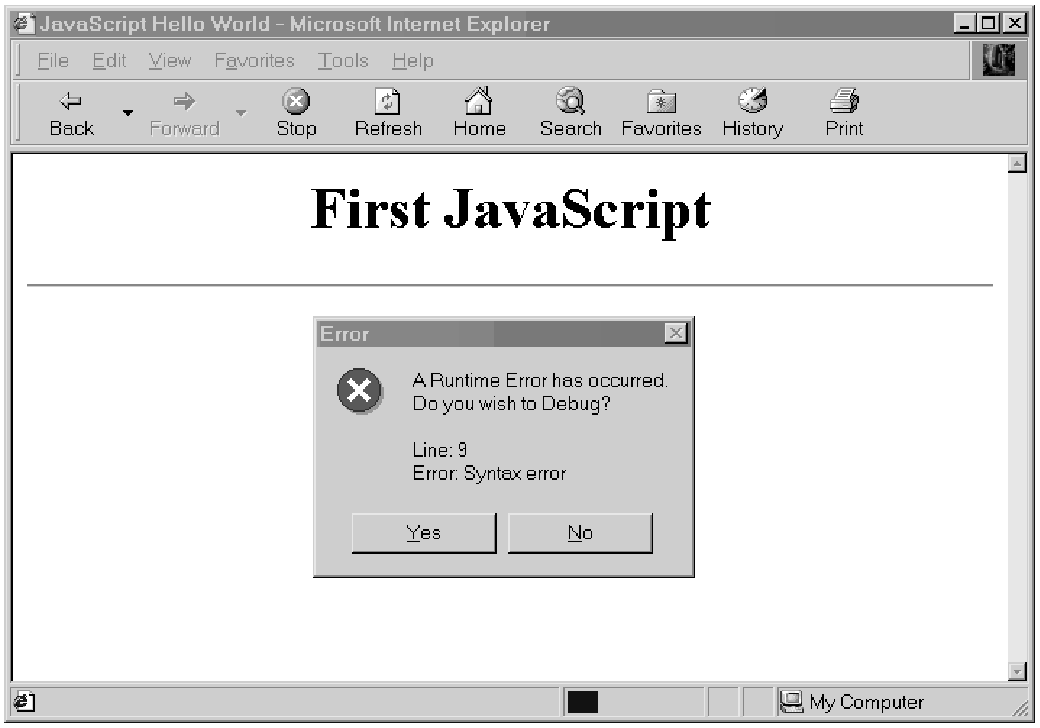
Hello World from JavaScript!

FIGURE 1-1 "Hello World from JavaScript" under Internet Explorer

L 1-3
```
<script type="text/javascript">
<strong>
   document.write("Hello World from JavaScript!");
</strong>
</script>
```

Doing so should throw an error in our browser window, as shown in Figure 1-2. The reason is that **<strong>** tags are markup, not JavaScript. Because the browser treats everything enclosed in **<script>** tags as JavaScript, it naturally throws an error when it encounters something that is out of place.

Note that some browsers unfortunately may not show errors directly on the screen. This is due to the fact that JavaScript errors are so commonplace on the Web that error dialogs became a real nuisance for many users, thus forcing the browser vendors to suppress errors by default. In the case of many Netscape browsers you can type **javascript:** in the URL bar to view the JavaScript console. In the case of Mozilla browsers, choose Tools | Web Development, and enable the JavaScript console. Under Internet Explorer, by default the only indication an error has occurred is a small error icon (yellow with an exclamation point) in the lower left-hand corner of the browser's status bar. Clicking this icon shows a dialog box with error information. In order to have this information displayed automatically, you may have to check "Display a notification about every script error," which can be found under the Advanced tab of the dialog displayed when selecting Internet Options.


JavaScript Hello World - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Forward
Stop
Refresh
Home
Search
Favorites
History
Print
My Computer
Error
Yes
No

First JavaScript

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

Line: 9
Error: Syntax error

FIGURE 1-2 JavaScript error dialog

Regardless of whether or not the error was displayed, to output the string properly we could either include the **<strong>** element directly within the output string, like so,

L 1-4
```
document.write("<strong>Hello World</strong> from
<font color='red'>JavaScript</font>!");
```

or we could surround the output of the **<script>** element in a **<strong>** element like this:

L 1-5
```
<strong>
<script type="text/javascript">
   document.write("Hello World from JavaScript!");
</script>
</strong>
```

In this case the **<strong>** tag happens to surround the output from the JavaScript so it then gets read and is generally bolded by the browser. This example suggests the importance of understanding the intersection of markup and JavaScript. In fact, before learning JavaScript, readers should fully understand the subtleties of correct HTML or more importantly XHTML markup. This is not a casual suggestion. Consider first that any JavaScript used within malformed (X)HTML documents may act unpredictably, particularly if the script tries to manipulate markup that is not well formed. Second, consider that many, if not most, scripts will be used to produce markup, so you need to know what you are outputting. In short, a firm understanding of (X)HTML is essential to writing effective scripts. In this book we present all examples in validated XHTML 1.0 Transitional unless otherwise noted. We chose this variant of markup because it balances the strictness of XHTML with the common practices of today's Web developers.

TIP *Readers looking for more information on correct HTML and XHTML usage should consult the companion book HTML & XHTML: The Complete Reference, Fourth Edition by Thomas Powell (McGraw-Hill/Osborne, 2003).*

Adding JavaScript to XHTML Documents

As suggested by the previous example, the **<script>** element is commonly used to add script to a document. However, there are four standard ways to include script in an XHTML document:

- Within the **<script>** element
- As a linked file via the **src** attribute of the **<script>** element
- Within an XHTML event handler attribute such as **onclick**
- Via the pseudo-URL **javascript:** syntax referenced by a link

Note that some older browser versions support other non-standard ways to include scripts in your page, such as Netscape 4's entity inclusion. However, we avoid discussing these in this edition since today these methods are interesting only as historical footnotes and are not used. The following section presents the four common methods for combining markup and JavaScript, and should be studied carefully by all readers before tackling the examples in the rest of the book.

The <script> Element

The primary method to include JavaScript within HTML or XHTML is the **<script>** element. A script-aware browser assumes that all text within the **<script>** tag is to be interpreted as some form of scripting language; by default this is generally JavaScript. However, it is possible for the browser to support other scripting languages such as VBScript, which is supported by the Internet Explorer family of browsers. Traditionally, the way to indicate the scripting language in use is to specify the **language** attribute for the tag. For example,

L 1-6
```
<script language="JavaScript">

</script>
```

is used to indicate the enclosed content is to be interpreted as JavaScript. Other values are possible; for example,

L 1-7
```
<script language="VBS">

</script>
```

would be used to indicate VBScript is in use. A browser should ignore the contents of the **<script>** element when it does not understand the value of its **language** attribute.

***TIP** Be very careful setting the **language** attribute for **<script>**. A simple typo in the value will usually cause the browser to ignore any content within.*

According to the W3C HTML syntax, however, the **language** attribute should not be used. Instead the **type** attribute should be set to indicate the MIME type of the language in use. JavaScript's MIME type is generally agreed upon to be "text/javascript", so you use

L 1-8
```
<script type="text/javascript">
</script>
```

***NOTE** The "W3C" is the World Wide Web Consortium, the international body responsible for standardizing Web-related technologies such as HTML, XML, and CSS. The W3C Web site is **www.w3.org**, and is the canonical place to look for Web standards information.*

Practically speaking, the **type** attribute is not as common in markup as the **language** attribute, which has some other useful characteristics, particularly to conditionally set code depending on the version of JavaScript supported by the browser. This technique will be discussed in Chapter 22 and illustrated throughout the book. To harness the usefulness of the **language** attribute while respecting the standards of the **<script>** element, you might consider using both:

L 1-9
```
<script language="JavaScript" type="text/javascript">

</script>
```

Unfortunately, this doesn't work well in some cases. First off, your browser will likely respect the **type** attribute over **language** so you will lose any of the latter attribute. Secondly, the page will not validate as conforming to the XHTML standard because, as we've said, the **language** attribute is non-standard. Following the standard, using the **type** attribute is the best bet unless you have a specific reason to use the non-standard **language** attribute.

NOTE *Besides using the type attribute for* ***<script>****, according to HTML specifications you could also specify the script language in use document-wide via the <****meta****> element, as in* ***<meta http-equiv="Content-Script-Type" content="text/javascript" />****. Inclusion of this statement within the* ***<head>*** *element of a document would alleviate any requirement of putting the* ***type*** *attribute on each* ***<script>*** *element. However, poor browser support for this approach suggests continued use of the* ***language*** *and* ***content*** *attributes together to limit script execution.*

Using the <script> Element

You can use as many **<script>** elements as you like. Documents will be read and possibly executed as they are encountered, unless the execution of the script is deferred for later. (The reasons for deferring script execution will be discussed in a later section.) The next example shows the use of three simple printing scripts that run one after another.

L 1-10
```
<!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" lang="en">
<head>
<title>JavaScript and the Script Tag</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
</head>
<body>
<h1>Ready start</h1>
<script type="text/javascript">
      alert("First Script Ran");
</script>
<h2>Running...</h2>
<script type="text/javascript">
      alert("Second Script Ran");
</script>
<h2>Keep running</h2>
<script type="text/javascript">
      alert("Third Script Ran");
</script>
<h1>Stop!</h1>
</body>
</html>
```

Try this example in various browsers to see how the script runs. You may notice that with some browsers the HTML is written out as the script progresses, with others not. This shows that the execution model of JavaScript does vary from browser to browser.

Script in the <head>

A special location for the **<script>** element is within the **<head>** tag of an (X)HTML document. Because of the sequential nature of Web documents, the **<head>** is always read in first, so scripts located here are often referenced later on by scripts in the **<body>** of the document. Very often scripts within the **<head>** of a document are used to define variables or functions that may be used later on in the document. The example below shows how the script in the **<head>** defines a function that is later called by script within the **<script>** block later in the **<body>** of the document.

L 1-11

```
<!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" lang="en">
<head>
<title>JavaScript in the Head</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<script type="text/javascript">
function alertTest()
{
  alert("Danger! Danger! JavaScript Ahead");
}
</script>
</head>
<body>
<h2 align="center">Script in the Head</h2>
<hr />
<script type="text/javascript">
 alertTest();
</script>
</body>
</html>
```

Script Hiding

Most browsers tend to display the content enclosed by any tags they don't understand, so it is important to mask code from browsers that do not understand JavaScript. Otherwise, the JavaScript would show up as text in the page for these browsers. Figure 1-3 shows an example Web page viewed by non-JavaScript supporting browsers without masking. One easy way to mask JavaScript is to use HTML comments around the script code. For example:

L 1-12

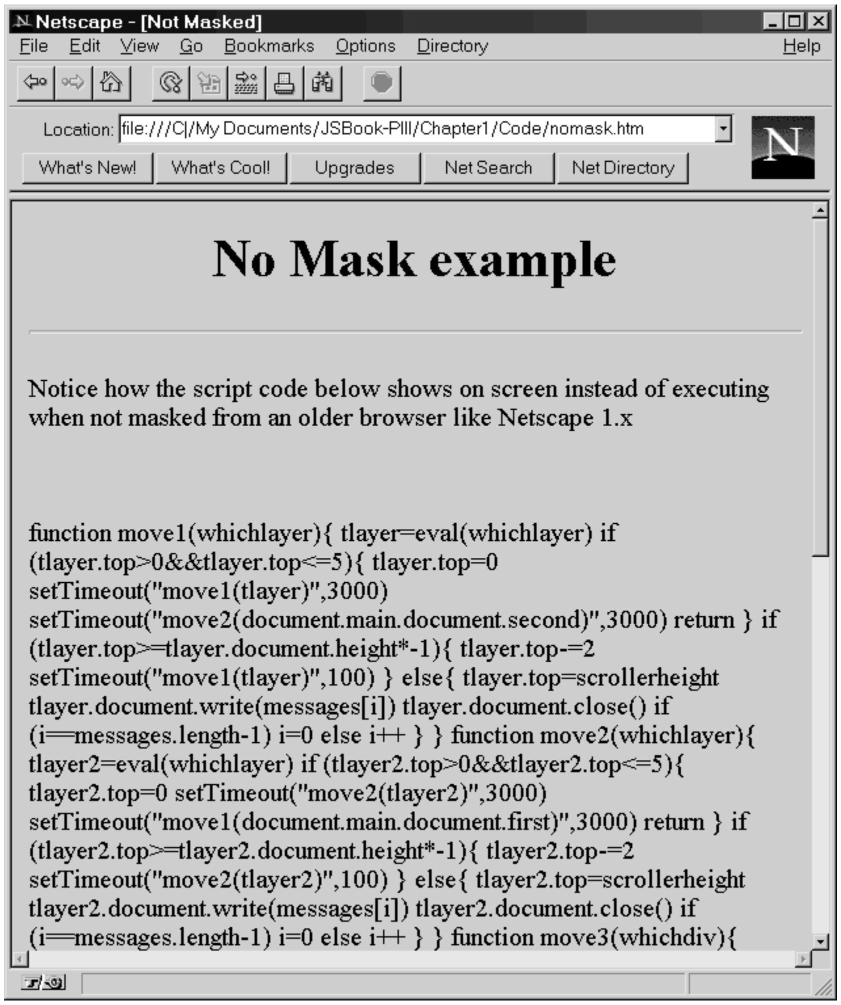
```
<script type="text/javascript">
<!--

  put your JavaScript here

//-->
</script>
```

NOTE *This masking technique is similar to the method used to hide CSS markup, except that the final line must include a JavaScript comment to mask out the HTML close comment. The reason for this is that the characters – and > have special meaning within JavaScript.*

While the comment mask is very common on the Web, it is actually not the appropriate way to do it in strict XHTML. Given that XHTML is an XML-based language, many of the characters found in JavaScript, such as > or &, have special meaning, so there could be trouble with the


Netscape - [Not Masked]
File Edit View Go Bookmarks Options Directory Help
Location: file:///C|/My Documents/JSBook-PIII/Chapter1/Code/nomask.htm
What's New! What's Cool! Upgrades Net Search Net Directory

No Mask example

Notice how the script code below shows on screen instead of executing when not masked from an older browser like Netscape 1.x

```
function move1(whichlayer){ tlayer=eval(whichlayer) if
(tlayer.top>0&&tlayer.top<=5){ tlayer.top=0
setTimeout("move1(tlayer)",3000)
setTimeout("move2(document.main.document.second)",3000) return } if
(tlayer.top>=tlayer.document.height*-1){ tlayer.top-=2
setTimeout("move1(tlayer)",100) } else{ tlayer.top=scrollerheight
tlayer.document.write(messages[i]) tlayer.document.close() if
(i==messages.length-1) i=0 else i++ } } function move2(whichlayer){
tlayer2=eval(whichlayer) if (tlayer2.top>0&&tlayer2.top<=5){
tlayer2.top=0 setTimeout("move2(tlayer2)",3000)
setTimeout("move1(document.main.document.first)",3000) return } if
(tlayer2.top>=tlayer2.document.height*-1){ tlayer2.top-=2
setTimeout("move2(tlayer2)",100) } else{ tlayer2.top=scrollerheight
tlayer2.document.write(messages[i]) tlayer2.document.close() if
(i==messages.length-1) i=0 else i++ } } function move3(whichdiv){
```

FIGURE 1-3 JavaScript code may print on the screen if not masked.

previous approach. According to the strict XHTML specification, you are supposed to hide the contents of the script from the XHTML-enforcing browser using the following technique:

L 1-13
```
<script type="text/javascript">
<![CDATA[
..script here ..
]]>
</script>
```

This approach does not work in any but the strictest XML-enforcing browsers. It generally causes the browser to ignore the script entirely or throw errors, so authors have the option of using linked scripts or traditional comment blocks, or simply ignoring the problem of down-level browsers. Most Web developers interested in strict XHTML conformance use linked scripts; developers only interested in HTML (or not interested in standards at all) generally use the traditional comment-masking approach. We've chosen the latter approach as it is the most widely used on the Web today.

The <noscript> Element

In the situation that a browser does not support JavaScript or that JavaScript is turned off, you should provide an alternative version or at least a warning message telling the user what happened. The **<noscript>** element can be used to accomplish this very easily. All JavaScript-aware browsers should ignore the contents of **<noscript>** unless scripting is off. Browsers that aren't JavaScript-aware will show the enclosed message (and they'll ignore the contents of the **<script>** if you've remembered to HTML-comment it out). The following example illustrates a simple example of this versatile element's use.

L 1-14

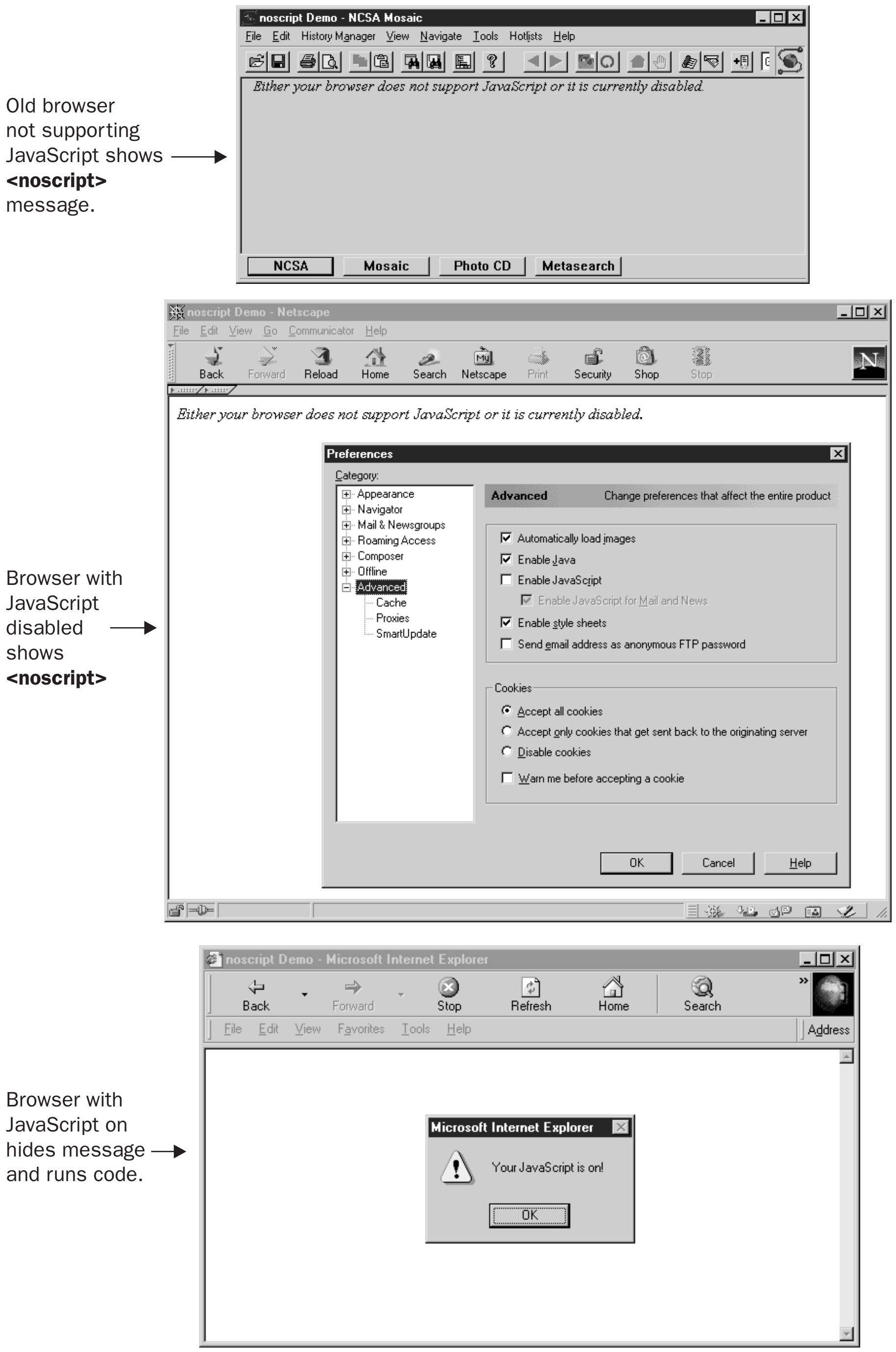
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en">
<head>
<title>noscript Demo</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
</head>
<body>
<script type="text/javascript">
<!--
      alert("Your JavaScript is on!");
//-->
</script>
<noscript>
      <em>Either your browser does not support JavaScript or it
         is currently disabled.</em>
</noscript>
</body>
</html>
```

Figure 1-4 shows a rendering in three situations: first a browser that does not support JavaScript, then a browser that does support it but has JavaScript disabled, and finally a modern browser with JavaScript turned on.

One interesting use of the **<noscript>** element might be to redirect users automatically to a special error page using a **<meta>** refresh if they do not have scripting enabled in the browser or are using a very old browser. The example below shows how this might be done.

L 1-15

```
<!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" lang="en">
<head>
<title>noscript Redirect Demo</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<!--  warning example does not validate -->
<noscript>
      <meta http-equiv="Refresh" content="0;URL=/errors/noscript.html" />
</noscript>
</head>
```


Old browser not supporting JavaScript shows **<noscript>** message.
noscript Demo - NCSA Mosaic
File Edit History Manager View Navigate Tools Hotlists Help
NCSA
Mosaic
Photo CD
Metasearch
Browser with JavaScript disabled shows **<noscript>**
noscript Demo - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Preferences
Category:
Appearance
Navigator
Mail & Newsgroups
Roaming Access
Composer
Offline
Advanced
Cache
Proxies
SmartUpdate
OK Cancel Help
Browser with JavaScript on hides message and runs code.
noscript Demo - Microsoft Internet Explorer
Back Forward Stop Refresh Home Search
File Edit View Favorites Tools Help
Address
Microsoft Internet Explorer
OK

Either your browser does not support JavaScript or it is currently disabled.

Either your browser does not support JavaScript or it is currently disabled.

Advanced Change preferences that affect the entire product

- [✓] Automatically load images
- [✓] Enable Java
- [ ] Enable JavaScript
- [✓] Enable JavaScript for Mail and News
- [✓] Enable style sheets
- [ ] Send email address as anonymous FTP password

Cookies

- (●) Accept all cookies
- ( ) Accept only cookies that get sent back to the originating server
- ( ) Disable cookies
- [ ] Warn me before accepting a cookie

Your JavaScript is on!

FIGURE 1-4 Use <noscript> **to handle browsers with no JavaScript.**

```
<body>
<script type="text/javascript">
<!--
 document.write("Congratulations! If you see this you have JavaScript.");
//-->
</script>
<noscript>
   <h2>Error: JavaScript required</h2>
   <p>Read how to <a href="/errors/noscript.html">rectify this problem</a>.</p>
</noscript>
</body>
</html>
```

Unfortunately, according to the XHTML specification the **<noscript>** tag is not supposed to be found in the **<head>**, so this example will not validate. This seems more an oversight than an error considering that the **<script>** tag is allowed in the **<head>**. However, for those looking for strict markup, this useful technique is not appropriate, despite the fact that it could allow for robust error handling of down-level browsers. More information about defensive programming techniques like this one is found in Chapter 23.

Event Handlers

To make a page more interactive you can add JavaScript commands that wait for a user to perform a certain action. Typically, these scripts are executed in response to form actions and mouse movements. To specify these scripts we set up various event handlers, generally by setting an attribute of an (X)HTML element to reference a script. We refer to these attributes collectively as *event handlers*—they perform some action in response to a user interface event. All of these attributes start with the word "on," indicating the event in response to which they're executed, for example, **onclick**, **ondblclick**, and **onmouseover**. This simple example shows how a form button would react to a click:

L 1-16
```
<!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" lang="en">
<head>
<title>JavaScript and HTML Events Example</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
</head>
<body>
<form action="#" method="get">
<input type="button" value="press me"
       onclick="alert('Hello from JavaScript!');" />
</form>
</body>
</html>
```

NOTE *When writing traditional HTML markup, developers would often mix case in the event handlers, for example onClick="". This mixed casing made it easy to pick them out from other markup and had no effect other than improving readability. Remember, these event handlers are part of HTML and would not be case sensitive, so onClick, ONCLICK, onclick, or even oNcLiCK are all valid. However, XHTML requires all lowercase, so you should lowercase event handlers regardless of the tradition.*

By putting together a few **<script>** tags and event handlers you can start to see how scripts can be constructed. The following example shows how a user event on a form element can be used to trigger a JavaScript defined in the **<head>** of a document.

L 1-17
```
<!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” lang=“en”>
<head>
<title>Event Trigger Example</title>
<meta http-equiv=“content-type” content=“text/html; charset=ISO-8859-1” />
<script type=“text/javascript”>
<!--
function alertTest( )
{
  alert(“Danger! Danger!”);
}
//-->
</script>
</head>
<body>
<div align=“center”>
<form action=“#” method=“get”>
<input type=“button” value=“Don’t push me!”
      onclick=“alertTest();” />
</form>
</div>
</body>
</html>
```

A rendering of the previous example is shown in Figure 1-5.

You may wonder which (X)HTML elements have event handler attributes. Beginning with the HTML 4.0 specification nearly every tag (generally, all that have a visual display) should have one of the core events, such as **onclick**, **ondblclick**, **onkeydown**, **onkeypress**, **onkeyup**, **onmousedown**, **onmousemove**, **onmouseout**, **onmouseover**, and **onmouseout** associated with it. For example, even though it might not make much sense, you should be able to specify that a paragraph can be clicked using markup and script like this:

L 1-18
```
<p onclick=“alert(‘Under HTML 4 you can!’)”>Can you click me</p>
```

Of course many older browsers, even from the 4.*x* generation, won’t recognize event handlers for many HTML elements, such as paragraphs. Most browsers, however, should understand events such as the page loading and unloading, link presses, form fill-in, and mouse movement. The degree to which each browser supports events and how they are handled varies significantly, but the core events are widely supported among modern browsers. Many examples throughout the book will examine how events are handled and an in-depth discussion on browser differences for event handling can be found in Chapter 11.

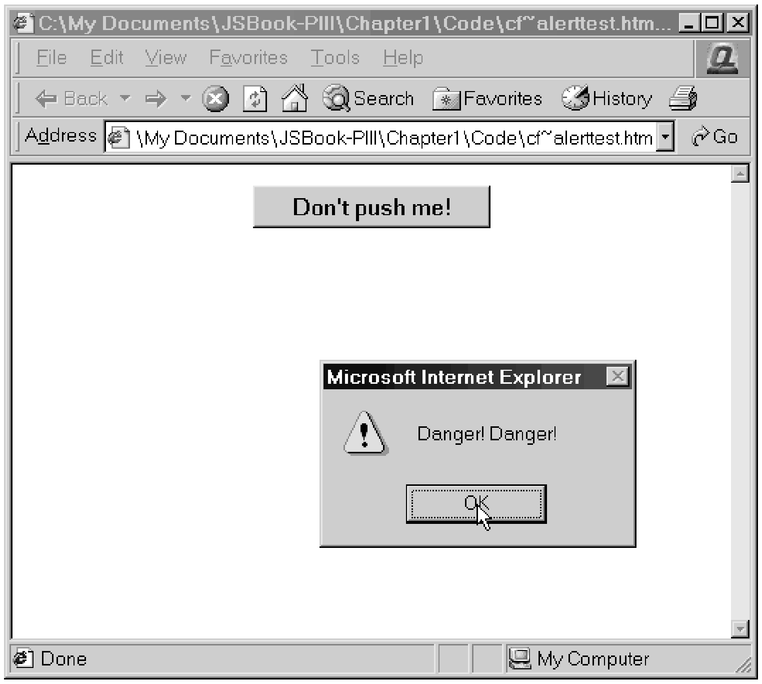

C:\My Documents\JSBook-PIII\Chapter1\Code\cf~alerttest.htm...
File Edit View Favorites Tools Help
Back Search Favorites History
Address \My Documents\JSBook-PIII\Chapter1\Code\cf~alerttest.htm
Go
Don't push me!
Microsoft Internet Explorer
Danger! Danger!
OK
Done
My Computer

FIGURE 1-5 Scripts can interact with users.

Linked Scripts

A very important way to include a script in an HTML document is by linking it via the **src** attribute of a **<script>** tag. The example here shows how we might put the function from the previous example in a linked JavaScript file.

L 1-19
```
<!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" lang="en">
<head>
<title>Event Trigger Example using Linked Script</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1" />
<script type="text/javascript" src="danger.js"></script>
</head>
<body>
<div align="center">
<form action="#" method="get">
<input type="button" value="Don't push me!" onclick="alertTest();" />
</form>
</div>
</body>
</html>
```

Notice that the **src** attribute is set to the value "danger.js." This value is a URL path to the external script. In this case, it is in the same directory, but it could have just as easily been an absolute URL such as http://www.javascriptref.com/scripts/danger.js. Regardless of the location of the file, all it will contain is the JavaScript code to run—no HTML or other Web technologies. So in this example the file danger.js could contain the following script:

L 1-20
```
function alertTest( )
{
  alert("Danger! Danger!");
}
```

The benefit of script files that are external is that they separate the logic, structure, and presentation of a page. With an external script it is possible to easily reference the script from many pages in a site easily. This makes maintenance of your code easier because you only have to update code common to many pages in one place (the external script file) rather than on every page. Furthermore, a browser can cache external scripts so their use effectively speeds up Web site access by avoiding extra download time retrieving the same script.

TIP *Consider putting all the scripts used in a site in a common script directory similar to how images are stored in an images directory. This will ensure proper caching, keep scripts separated from content, and start a library of common code for use in a site.*

While there are many benefits to using external scripts, they are often not used because of some of their potential downsides. An uncommon reason is that not all JavaScript-aware browsers support linked scripts. Fortunately, this problem is mostly related to extremely old browsers, specifically Netscape 2 and some Internet Explorer 3 releases. These are extremely uncommon browsers these days, so this isn't much of a concern unless you're hyper-conscious of backward-compatibility.

The primary challenge with external scripts has to do with browser loading. If an external script contains certain functions referenced later on, particularly those invoked by user activities, programmers must be careful not to allow them to be invoked until they have been downloaded or error dialogs may be displayed. That is, there's no guarantee as to when an externally linked script will be loaded by the browser. Usually, they're loaded very quickly, in time for any JavaScript in the page to reference them properly. But if the user is connecting via a very slow connection, or if script calling functions defined in the external script is executed immediately, they might not have loaded yet.

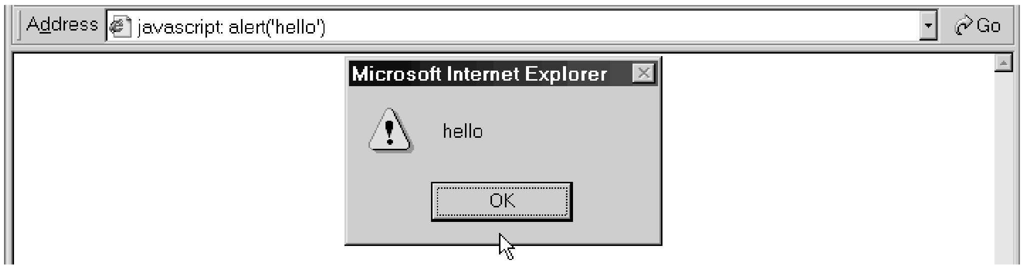
Fortunately, most of the problems with external scripts can be alleviated with good defensive programming styles as demonstrated throughout the book. Chapter 23 covers specific techniques in detail. However, if stubborn errors won't seem to go away and external scripts are in use, a good suggestion is to move the code to be included directly within the HTML file.

TIP *When using external .js files make sure that your Web server is set up to map the file extension .js to the MIME type text/javascript. Most Web servers have this MIME type set by default but if you are experiencing problems with linked scripts this could be the cause.*

JavaScript Pseudo-URL

In most JavaScript-aware browsers it is possible to invoke a script using the JavaScript pseudo-URL. A pseudo-URL like **javascript: alert('hello')** would invoke a simple alert displaying "hello" when typed directly in the browser's address bar, as shown here:

Ill 1-1



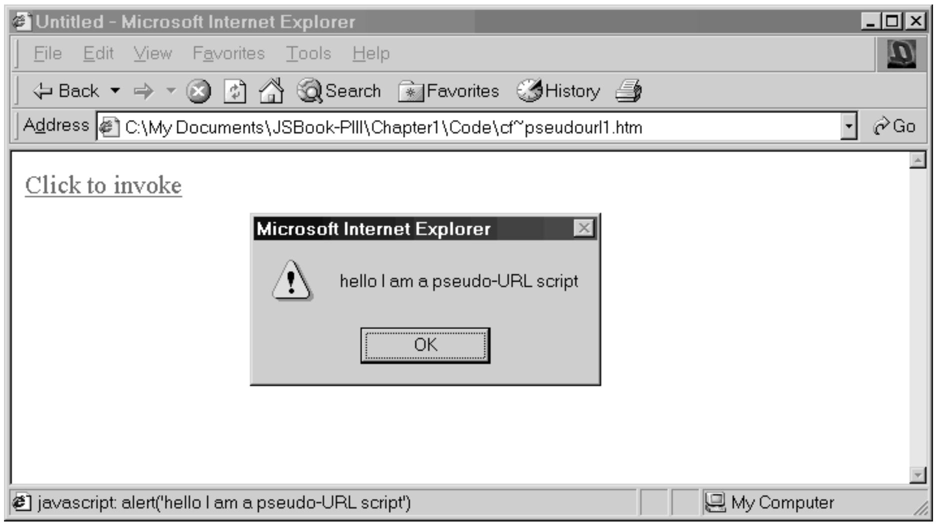
NOTE *Under some browsers, notably versions 4 and above of Netscape, it is possible to gain access to a JavaScript console when typing in the URL **javascript:** by itself. Other browsers have a console that can be accessed to view errors and test code. However, Internet Explorer does not provide such direct access to the console, which can be used both for debugging as well as for testing the values of scripts. Examples of the JavaScript console are shown in Figure 1-6.*

One very important way to use the JavaScript pseudo-URL is within a link, as demonstrated here:

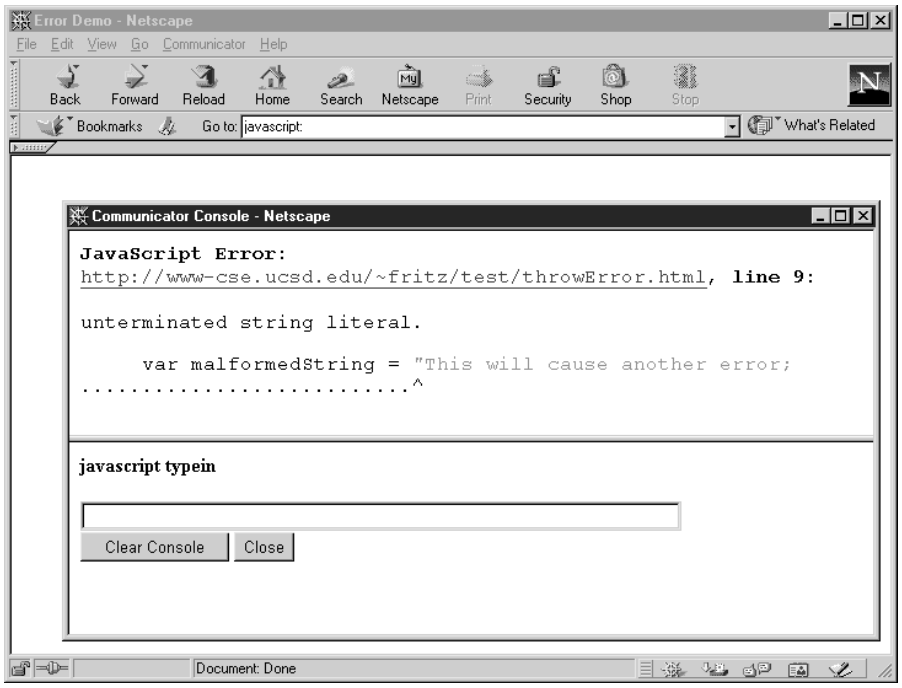
L 1-21

```
<a href="javascript: alert('hello I am a pseudo-URL script');">Click
to invoke</a>
```

Ill 1-2


Untitled - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
History
Address
C:\My Documents\JSBook-PIII\Chapter1\Code\cf~pseudourl1.htm
Go
Click to invoke
Microsoft Internet Explorer
hello I am a pseudo-URL script
OK
javascript: alert('hello I am a pseudo-URL script')
My Computer

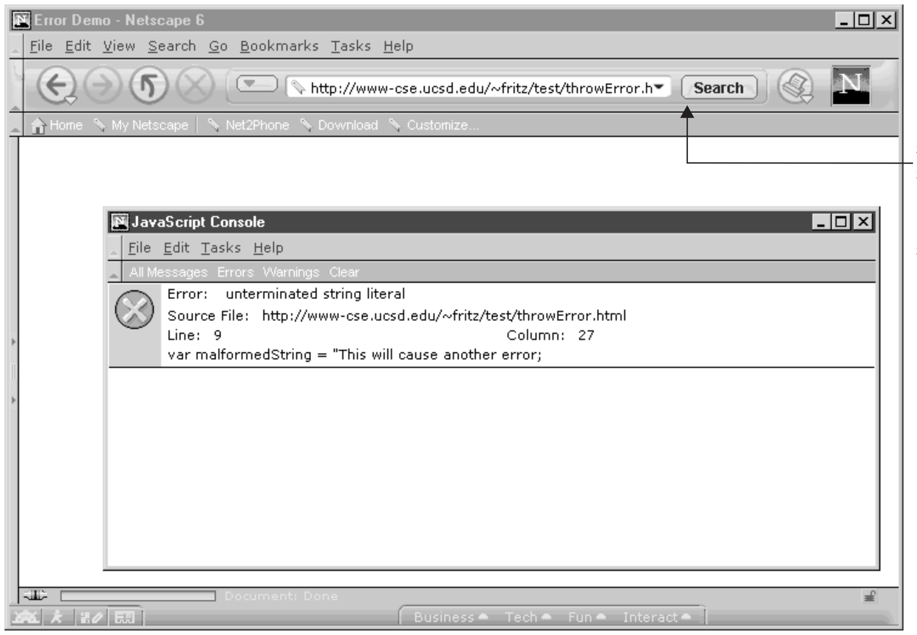
The pseudo-URL inclusion can be used to trigger any arbitrary amount of JavaScript, so


Error Demo - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Go to: javascript: What's Related
Communicator Console - Netscape
javascript typein
Clear Console Close
Document: Done

```
JavaScript Error:
http://www-cse.ucsd.edu/~fritz/test/throwError.html, line 9:

unterminated string literal.

     var malformedString = "This will cause another error;
...........................^
```


Error Demo - Netscape 6
File Edit View Search Go Bookmarks Tasks Help
http://www-cse.ucsd.edu/~fritz/test/throwError.h
Search
Home My Netscape Net2Phone Download Customize...
JavaScript Console
File Edit Tasks Help
All Messages Errors Warnings Clear
Document: Done
Business Tech Fun Interact
Notice the status bar shows the URL to invoke is script code.

```
Error:    unterminated string literal
Source File:   http://www-cse.ucsd.edu/~fritz/test/throwError.html
Line:  9                                     Column:  27
var malformedString = "This will cause another error;
```

FIGURE 1-6 JavaScript console used for debugging and testing

L 1-22
```
<a href="javascript: x=5;y=7;alert('The sum = '+(x+y));">Click to invoke</a>
```

is just as acceptable as invoking a single function or method. Some developers have found this quite useful and have designed functions to be executed on pages and saved as bookmarks. When these **javascript:** links are added as "Favorites" or "Bookmarks" in your browser they can be clicked in order to carry out a specific task. These scripts, typically dubbed *bookmarklets* or *favlets,* are used to resize windows, validate pages, and perform a variety of useful developer-related tasks.

NOTE *Running JavaScript via the URL in the form of a bookmark does have some security considerations. Since bookmarklets stored in your browser execute in the context of the current page, a malicious bookmarklet could be used to steal cookies for the current site. For this reason, only install bookmarklets from sites you trust, or only after examining their code.*

The **javascript:** URL does have a problem, of course, when used in a browser that does not support JavaScript. In such cases, the browser will display the link appropriately but the user will not be able to cause the link to do anything, which would certainly be very frustrating. Designers relying on pseudo-URLs should make sure to warn users using the **<noscript>** element, as shown here:

L 1-23
```
Fixed<noscript>
<strong><em>Warning:</em> This page contains links that use JavaScript
and your browser either has JavaScript disabled or does not support this
 technology.</strong>
</noscript>
```

However, this assumes that the user sees the message. A more defensive coding style might be to recode the initial pseudo-URL link as follows.

L 1-24
```
<a href="/errors/noscript.html" onclick=" alert('hello I am a pseudo-URL
script');return false;">Click to invoke</a>
```

In this case, with the script on the **onclick,** the JavaScript is run when the link is clicked and return false kills the page load. However, with script off, the code will not run and instead the user will be sent to the error page specified by the **href** attribute. While the **javascript:** pseudo-URL does have some limitations, it is commonly found in all major implementations of the language and used by many developers. It is definitely better, however, to avoid using the pseudo-URL technique and replace it with the defensive onclick code presented. Now before concluding the chapter, let's take a brief look at what JavaScript is used for, where it came from and where it is likely going.

History and Use of JavaScript

Knowledge of JavaScript's past actually leads to a great deal of understanding about its quirks, challenges, and even its potential role as a first class Web technology. For example, even the name JavaScript itself can be confusing unless you consider history since, despite the similarity in name, JavaScript has nothing to do with Java. Netscape initially introduced the language under the name LiveScript in an early beta release of Navigator 2.0 in 1995, and the focus of the language was initially for form validation. Most likely the language was renamed JavaScript because of the industry's fascination with all things Java at the time as well as the potential for the two languages to be integrated together to build Web applications. Unfortunately, because of including the word "Java" in its name, JavaScript is often thought of as some reduced scripting form of Java. In reality the language as it stands today is only vaguely similar to Java, and syntactically often shares more in common with languages such as C, Perl, and Python.

While the name of the language has led to some confusion by some of its users, it has been widely adopted by browser vendors. After Netscape introduced JavaScript in version 2.0 of their browser, Microsoft introduced a clone of JavaScript called JScript in Internet Explorer 3.0. Opera also introduced JavaScript support during the 3.*x* generation of its browser. Many other browsers also support various flavors of JavaScript. As time has gone by, each of the major browser vendors has made their own extensions to the language and the browsers have each supported various versions of JavaScript or JScript. Table 1-1 details the common browsers that support a JavaScript language. The various features of each version of JavaScript are discussed throughout the book, and Appendix B provides information on the support of various features in each version of the language.

Because the specification of JavaScript is changing rapidly and cross-platform support is not consistent, you should be very careful with your use of JavaScript with browsers. Since different levels of JavaScript support different constructs, programmers should be careful to create conditional code to handle browser and language variations. Much of the book will deal with such issues, but a concentrated discussion can be found in Chapter 23.

Because of the cross-browser JavaScript nightmare inflicted on programmers, eventually a standard form of JavaScript called ECMAScript (pronounced *eck-ma-script*) was specified. Version 3 is the latest edition of ECMAScript. While most of the latest browsers have full or close to full support for ECMAScript, the name itself has really yet to catch on with the public, and most programmers tend to refer to the language, regardless of flavor, as simply JavaScript.

TABLE 1-1 Browser Versions and JavaScript Support

Browser Version	JavaScript Support
Netscape 2.*x*	1.0
Netscape 3.*x*	1.1
Netscape 4.0–4.05	1.2
Netscape 4.06–4.08, 4.5*x*, 4.6*x*, 4.7*x*	1.3
Netscape 6.*x*,7.*x*	1.5
Mozilla variants	1.5
Internet Explorer 3.0	Jscript 1.0
Internet Explorer 4.0	Jscript 3.0
Internet Explorer 5.0	Jscript 5.0
Internet Explorer 5.5	Jscript 5.5
Internet Explorer 6	Jscript 5.6

NOTE *JavaScript 2.0 and ECMAScript version 4 are both being slowly pushed through the standards process. Given the fall of Netscape, it is unclear what is going to happen to these versions of the language, and so far the browser vendors are far from implementing the language. However, brief mentions of important differences will be presented throughout the book where appropriate.*

Even with the rise of ECMAScript, JavaScript can still be challenging to use. ECMAScript primarily is concerned with defining core language features such as flow control statements (e.g., **if**, **for**, **while**, and so on) and data types. But JavaScript also generally can access a common set of objects related to its execution environment, most commonly a browser. These objects—such as the window, navigator, history, screen—are not a part of the ECMAScript specification, and are collectively referred to as the traditional *Browser Object Model* or *BOM*. The fact that all the browser versions tend to have similar but subtly different sets of objects making up their BOMs causes mass confusion and widespread browser incompatibility in Web pages. The BOM finally reached its worst degree of incompatibility with the 4.*x* generation of browsers introducing the idea of Dynamic HTML, or DHTML. In reality there is no such thing, technically, as DHTML. The idea came from marketing terms for the 4.*x* generation browsers and was used to characterize the dynamic effects that arise from using HTML, CSS, and JavaScript on a page. If you are talking about DHTML you are talking about the intersection of these technologies and not some all-new technology separate from JavaScript.

Fortunately, the W3C has defined standard objects with which to access Web page components such as HTML elements and their enclosed text fragments, CSS properties, and even XML elements. In doing so, they've tried to end the nightmare of DHTML incompatibilities. Their specification is called the *Document Object Model,* or *DOM* for short. It defines a standard way to manipulate page elements in markup languages and style sheets providing for all the effects possible with DHTML without the major incompatibilities. However, there is some cross-over between what is part of the traditional object model and what is DOM, and differences in DOM implementations abound. Fortunately, the newer browsers have begun to iron out many incompatibilities and the interaction between JavaScript and page objects is finally starting to become well defined. More information on the DOM can be found at **http://www.w3.org/DOM** as well as in Chapter 10.

When taken together, core JavaScript as specified by ECMAScript, browser objects, and document objects will provide all the facilities generally required by a JavaScript programmer. Unfortunately, save the core language, all the various objects available seem to vary from browser to browser and version to version, making correct cross-browser coding a real challenge! A good portion of this book will be spent trying to iron out these difficulties.

As we have seen, study of the evolution of JavaScript can be critical for mastering its use, as it explains some of the design motivations behind its changes. While JavaScript is quite powerful as a client-side technology, like all languages, it is better at some types of applications than others. Some of these common uses of JavaScript include

- Form validation
- Page embellishments and special effects
- Navigation systems

- Basic mathematical calculations
- Dynamic document generation
- Manipulation of structured documents

JavaScript does have its limits. It does not support robust error-handling features, strong typing, or facilities useful for building large-scale applications. Yet despite its flaws and many of the misunderstandings surrounding the language, it has succeeded wildly. Some might say, if you consider all Web developers who have touched the language at one point or another, it is one of the most popular and widely used—though misunderstood—languages on the planet. JavaScript's popularity is growing even beyond the Web, and we see its core in the form of ECMAScript being used in embedded systems and within applications such as Dreamweaver as an internal automation and scripting language. ECMAScript has also spawned numerous related languages, most notably ActionScript in Flash. Much of the user interface of the Mozilla and modern Netscape Web browsers is implemented with JavaScript. JavaScript is no longer relegated to trivial simple rollover effects and form checking; it is a powerful and widely used language. As such, JavaScript should be studied rigorously, just like any programming language, and that is what we will do starting in the next chapter.

Summary

JavaScript has quickly become the premier client-side scripting language used within Web pages. Much of the language's success has to do with the ease with which developers can start using it. The **<script>** element makes it easy to include bits of JavaScript directly within HTML documents; however, some browsers may need to use comments and the **<noscript>** element to avoid errors. A linked script can further be employed to separate the markup of a page from the script that may manipulate it. While including scripts can be easy, the challenges of JavaScript are numerous. The language is inconsistently supported in browsers and its tumultuous history has led to numerous incompatibilities. However, there is hope in sight. With the rise of ECMAScript and the W3C specified Document Object Model, many of the various coding techniques required to make JavaScript code work in different browsers may no longer be necessary.