The Complete Reference

Part 1

Introduction

The Complete Reference

Chapter 1

Introduction to JavaScript

JavaScript is the premier client-side scripting language used today on the Web. Its use is widespread 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. Soon JavaScript will be increasingly used to manipulate the very HTML and even XML documents in which it is contained. When it finally achieves this role, it will become a first class client-side Web technology, ranking alongside HTML, CSS, and, eventually, 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.

A 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 the HTML document to be displayed:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>JavaScript Hello World</title>
</head>
<body>
<h1 align="center">First JavaScript</h1>
<hr>
<script language="JavaScript" type="text/javascript">
   document.write("Hello World from JavaScript!");
</script>
</body>
</html>
```

Notice how the script is intermixed into the HTML document using the **<script>** element that encloses the simple one line script:

```
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 HTML. If we type this example in using a standard text editor, we can load it into a JavaScript-aware Web browser—such as Internet Explorer 3, Netscape 2, Opera 3, or any later version of these browsers—and we should see the result shown in Figure 1-1.

If we wanted to embold the text we could modify the script to output not only some text but also some HTML. However, we need to be careful when the world of

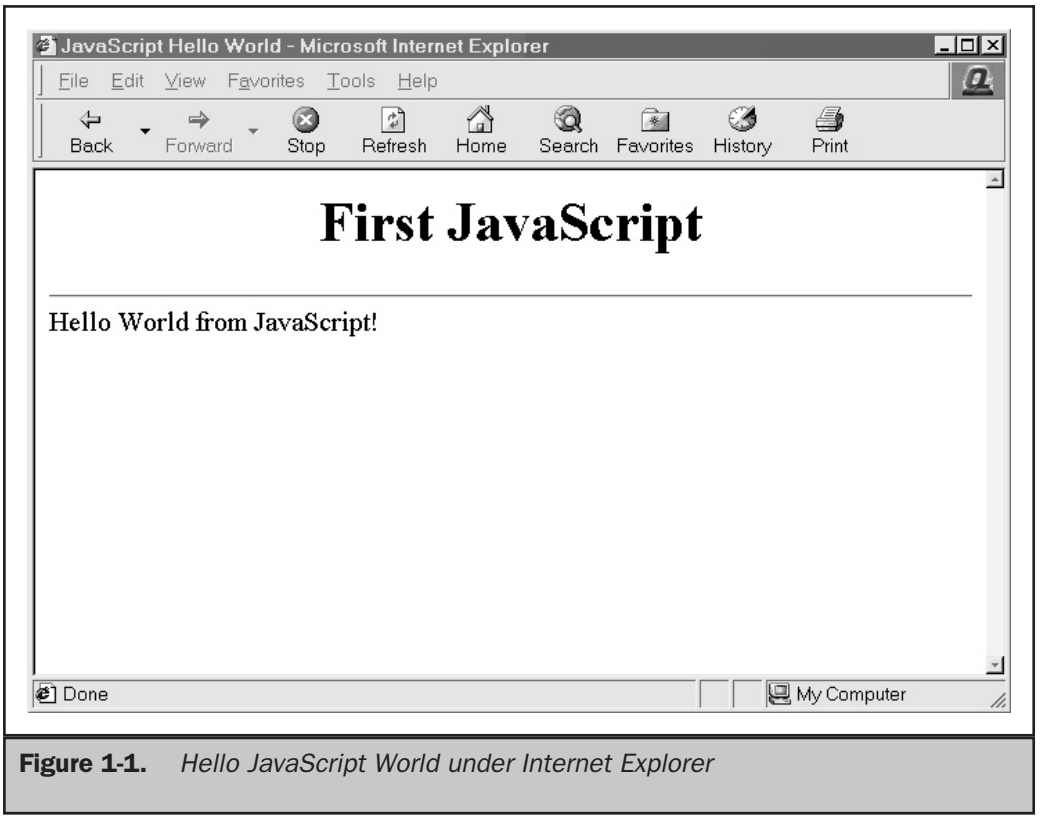


Figure 1-1. *Hello JavaScript World under Internet Explorer*

JavaScript and the world of HTML intersect—they are two different technologies. For example, consider what happens if we substitute the following **<script>** block in the document, hoping that it would embold the text:

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

Doing so would probably throw an error in our browser window as shown in Figure 1-2. The reason for this is that **<b>** tags are part of HTML, 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.

Figure 1-2. *JavaScript error dialog*

Some browsers unfortunately may not show errors directly on the screen. The reason for this is that JavaScript errors are so commonplace on the Web that error dialogs became a real nuisance for many users, thus leading the browser vendors to suppress errors by default. In the case of Netscape 4.x generation browsers, you can type **javascript:** in the URL bar to view the JavaScript console. In the case of Netscape 6, choose the Tasks menu, then the Tools menu, and enable the JavaScript console. With Internet Explorer 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. There may be different ways for achieving this in other browsers.

Regardless of whether or not the error was displayed, to output the string properly we either include the **<b>** element directly within the output string, as in:

```
document.write("<b>Hello World</b> from <font
```

or surround **<script>** element itself in a **<b>** element, like this:

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

In this case the <**b**> tag surrounds the output from the JavaScript, which is displayed as boldface text by the browser. This example suggests the importance of understanding the intersection of HTML and JavaScript. In fact, before learning JavaScript, readers should fully understand the subtleties of correct HTML markup, especially given that most scripts will be used to produce HTML markup. Any JavaScript used within malformed HTML documents may act unpredictably, particularly if the script tries to manipulate HTML that is not well formed. A firm understanding of HTML is essential to writing effective scripts.

Tip *Readers looking for more information on correct HTML usage should consult the companion book,* HTML: The Complete Reference, *3rd Edition, by Thomas Powell (Osborne/McGraw-Hill, 2001).*

Adding JavaScript to HTML 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 JavaScript in an HTML document:

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

There are other nonstandard ways to include scripts in your page, most commonly with the entity JavaScript. It is also possible to execute JavaScript on the server side by embedding it in an application or even using it to automate common operating system tasks. The use of the language outside the Web browser and HTML documents is extremely uncommon and is not significantly discussed in this book. However, Chapter 19 presents a brief discussion of JavaScript's server-side role. The following section presents the various methods for combining with HTML 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 of including JavaScript within HTML is by using 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,

```
<script language="JavaScript">

</script>
```

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

```
<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 "text/javascript," so you use:

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

Practically speaking, the **type** attribute is not as common as the **language** attribute, which has some other useful characteristics, particularly to conditionally set code according to the version of JavaScript supported by the browser (this technique will be discussed in Chapter 24 and illustrated throughout the book). To harness the usefulness of the **language** attribute while respecting the standards of the **<script>** element, using the following would be a good idea:

```
<script language="JavaScript" type="text/javascript">

</script>
```

Note *Besides using the **type** attribute for **<script>**, 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 avoid having to put the type attribute on each **<script>** element. However, poor browser support for this approach argues for the continued use of the **language** and **type** 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.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>JavaScript and Script Tag</title>
</head>
<body>
<h1>Ready start</h1>
<script language="Javascript" type="text/javascript">
      alert("First Script Ran");
</script>
<h2>Running…</h2>
<script language="Javascript" type="text/javascript">
      alert("Second Script Ran");
</script>
<h2>Keep running</h2>
<script language="Javascript" 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. With some browsers the HTML is written out as the script progresses, with others not. This difference is an example of how the execution model of JavaScript varies from browser to browser.

Script in the <head>

A special location for the **<script>** element is within the **<head>** tag of an 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 here shows how the script in the **<head>** defines a function that is later called by script within the **<script>** block in the **<body>** of the document.

```
<!DOCTYPE html public "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<script language="JavaScript" type="text/javascript">
function alertTest()
{
  alert("Danger! Danger! JavaScript Ahead");
}
</script>
</head>
<body>
<h2 align="center">Script in the Head</h2>
<hr>
<script language="JavaScript" type="text/javascript">
 alertTest();
</script>
</body>
</html>
```

Script Hiding

With HTML, browsers tend to print out anything they don't understand on the screen, so it is important to mask code from browsers. One way to do this is to use comments around the script code; for example:

```
<script language="JavaScript" type="text/javascript">
<!--
  put your JavaScript here
//-->
</script>
```

Figure 1-3 shows a Web page viewed by non-JavaScript–supporting browsers without masking.

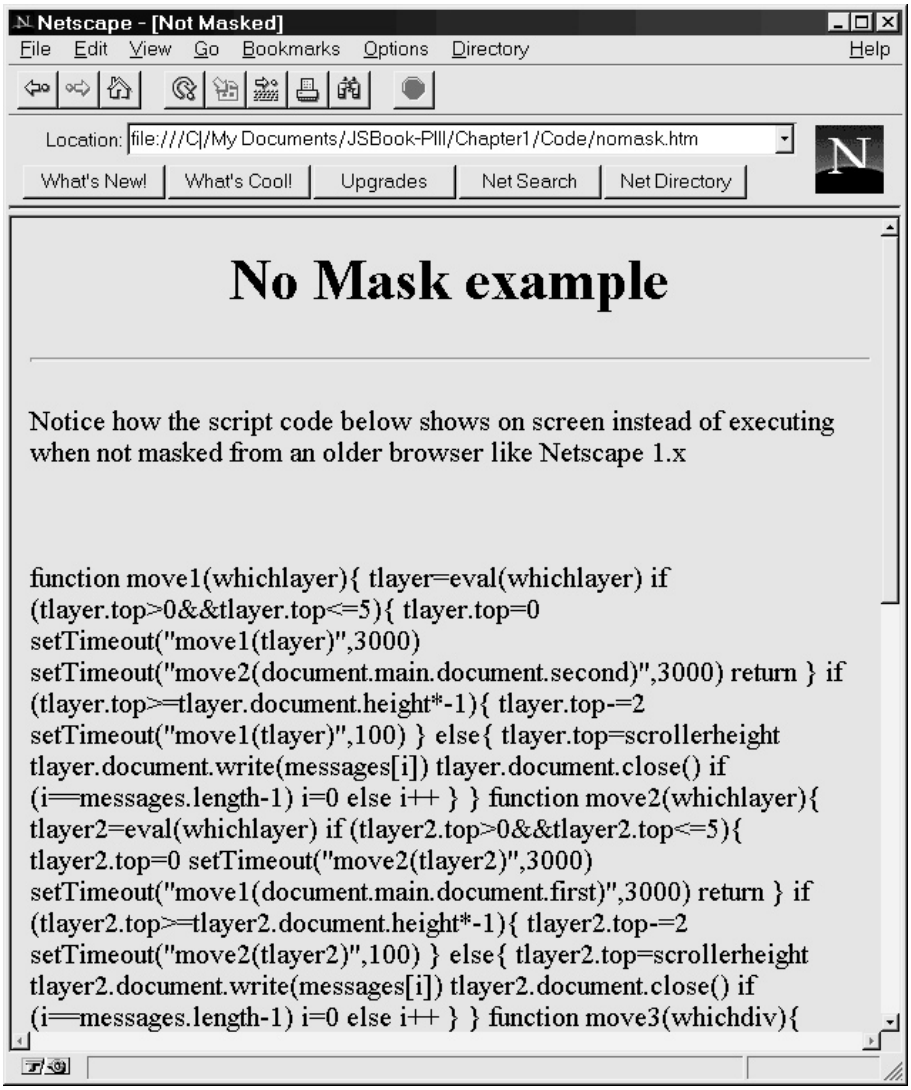


Figure 1-3. *JavaScript code, not masked, but printed on screen*

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.*

The <noscript> Element

When a browser does not support JavaScript or when 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. The following example illustrates a simple example of this versatile element's use:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>noscript Demo</title>
</head>
<body>
<script language="JavaScript" type="text/javascript">
<!--
      alert("Your JavaScript is on!");
//-->
</script>
<noscript>
      <i>Either your browser does not support JavaScript or it
         is currently disabled.</i>
</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 is to automatically redirect users to a special error page if they do not have scripting enabled in the browser or are using a very old browser. This example shows how it might be done:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>Needs JavaScript</title>
<noscript>
      <meta http-equiv="Refresh" content="0;URL=noscript.htm">
</noscript>
```

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

More information about defensive programming techniques like this one is found in Chapter 24.

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 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 HTML element to reference a script. We refer to these HTML attributes collectively as *event handlers*. All of these attributes start with the word "on"—for example, **onclick**, **ondblclick**, and **onmouseover**. The simple example here shows how a form button would react to a click:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>JavaScript and HTML Events Example</title>
</head>
<body>
<form>
<input type="button" value="press me"
       onclick="alert('Hello from JavaScript!');">
</form>
</body>
</html>
```

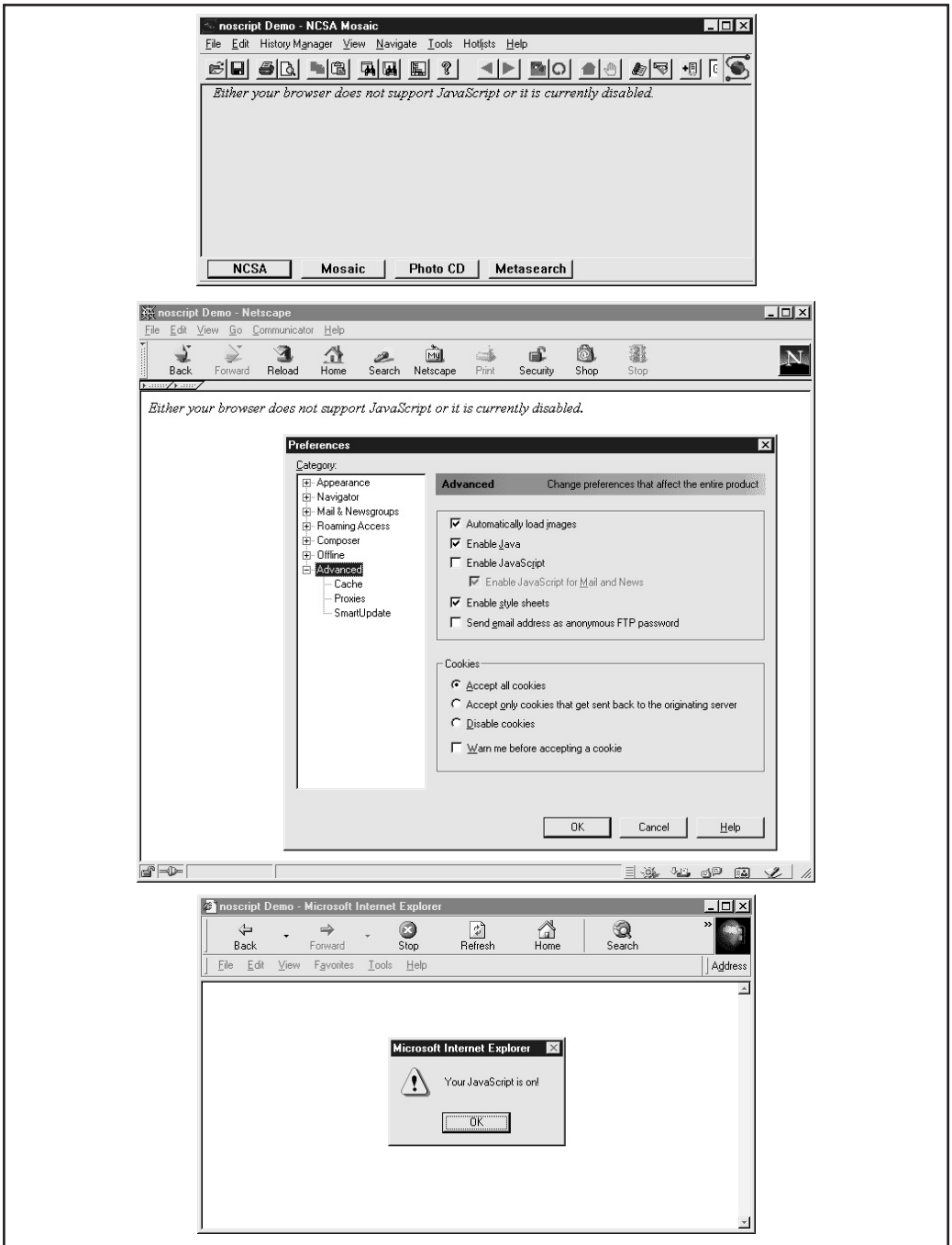


Figure 1-4. *Use of <noscript> to handle browsers with no JavaScript*

Note *When writing traditional HTML markup, developers would often mix-case 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 are not case sensitive, so onClick, ONCLICK, onclick, or even oNcLiCK are all valid. However, because the eventual movement towards XHTML will require all lowercase, 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.

```
<!DOCTYPE html public "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<script language="JavaScript" type="text/javascript">
<!--
function alertTest( )
{
  alert("Danger! Danger!");
}
//-->
</script>
</head>
<body>
<div align="center">
<form>
<input type="button" name="TestButton"
       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 what HTML elements have event handler attributes. Under the HTML 4.0 specification nearly every tag should have most of the core events, such as **onclick**, **ondblclick**, **onkeydown**, **onkeypress**, **onkeyup**, **onmousedown**, **onmousemove**, **onmouseout**, and **onmouseover**, 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:

```
<p onclick="alert('Under HTML 4 you can!')">Can you click me</p>
```

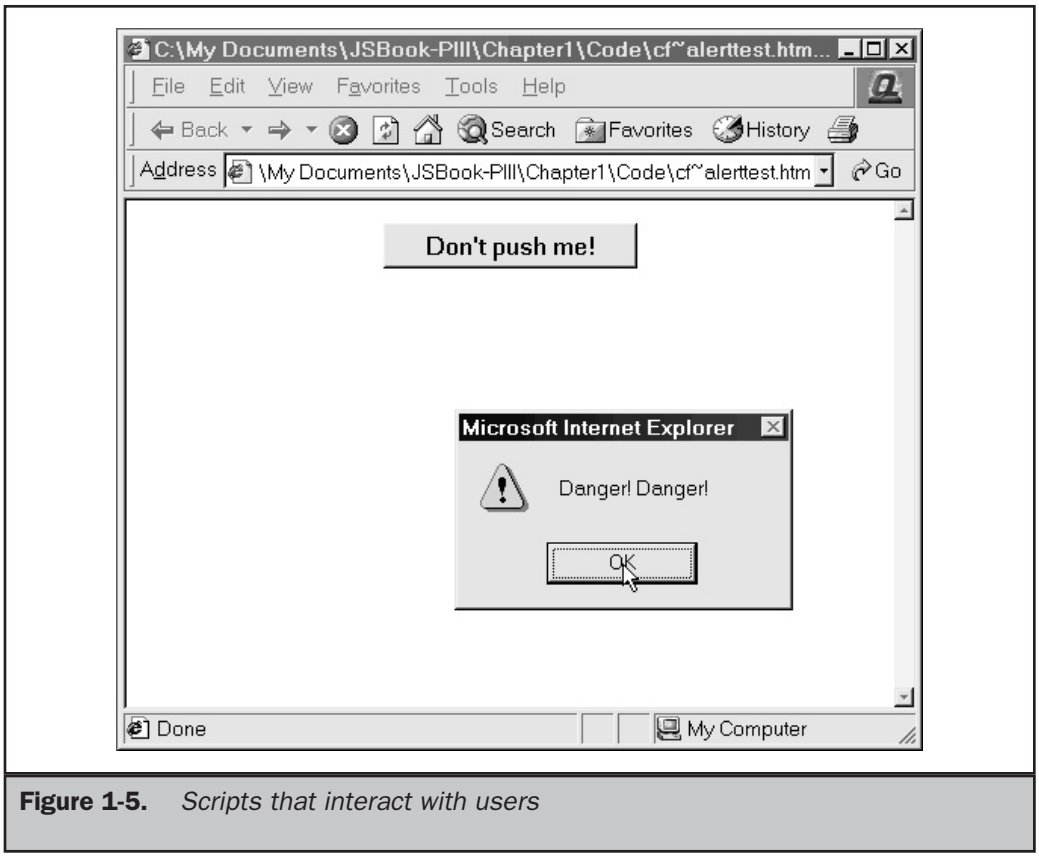


Figure 1-5. *Scripts that interact with users*

Of course many older browsers, even those from the 4.x generation, won't recognize event handlers for many HTML elements, such as a paragraph. Most browsers, however, should understand events such as the page loading and unloading, link presses, form fill-in, and mouse movement. Unfortunately, the degree to which each browser supports events and how they are handled varies significantly. There will be many examples throughout the book examining how events are handled, and an in-depth discussion on browser differences for event handling can be found in Chapter 11.

Linked Scripts

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

```
<!doctype html public "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<script language="JavaScript" type="text/javascript"
        src="danger.js">
</script>
</head>
<body>
<div align="center">
<form>
<input type="button" name="TestButton"
       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 just have 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 should contain the following script:

```
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 and to update only one file to affect many others. Further, a browser can cache external scripts so their use effectively speeds up Web site access by avoiding extra download time spent refetching the same script.

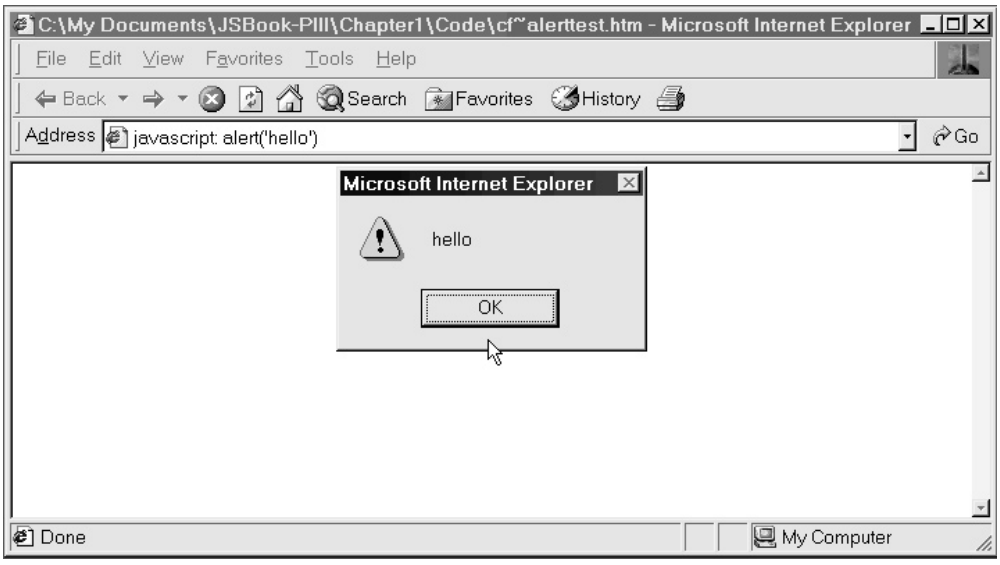
Tip *Consider putting all the scripts used in a site in a common script directory similar to how images are stored in an images directory. Doing this will ensure proper caching, keep scripts separated from content, and start a library of common code for use in a site.*

While external scripts have many benefits, they are often not used because of potential downsides. First, not all JavaScript-aware browsers support linked scripts. Fortunately, this problem is related mostly to older browsers, specifically Netscape 2 and some Internet Explorer 3 releases. Another 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; otherwise error dialogs may be displayed. Lastly, there are just plain and simple bugs when using external scripts. Fortunately, most of the problems with external scripts can be alleviated with the good defensive programming styles demonstrated throughout this book. However, if stubborn errors won't seem to go away and external scripts are in use, a good strategy is to move the code into the HTML file itself.

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 statement using a JavaScript pseudo-URL. A pseudo-URL begins with **javascript:** and is followed by the code to execute. For example, typing **javascript: alert('hello')** directly into the browser's address bar invokes the alert box shown here:



Under Netscape 4 and 6, it is possible to gain access to a JavaScript console by typing in the URL **javascript:** by itself in the address bar. Netscape 6 also provides this access via the Tools submenu of the Tasks menu. Other browsers may not provide such direct access to the console, which can be used 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.

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

When the user clicks the link, the alert is executed, resulting in the dialog box shown here:

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

```
<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.

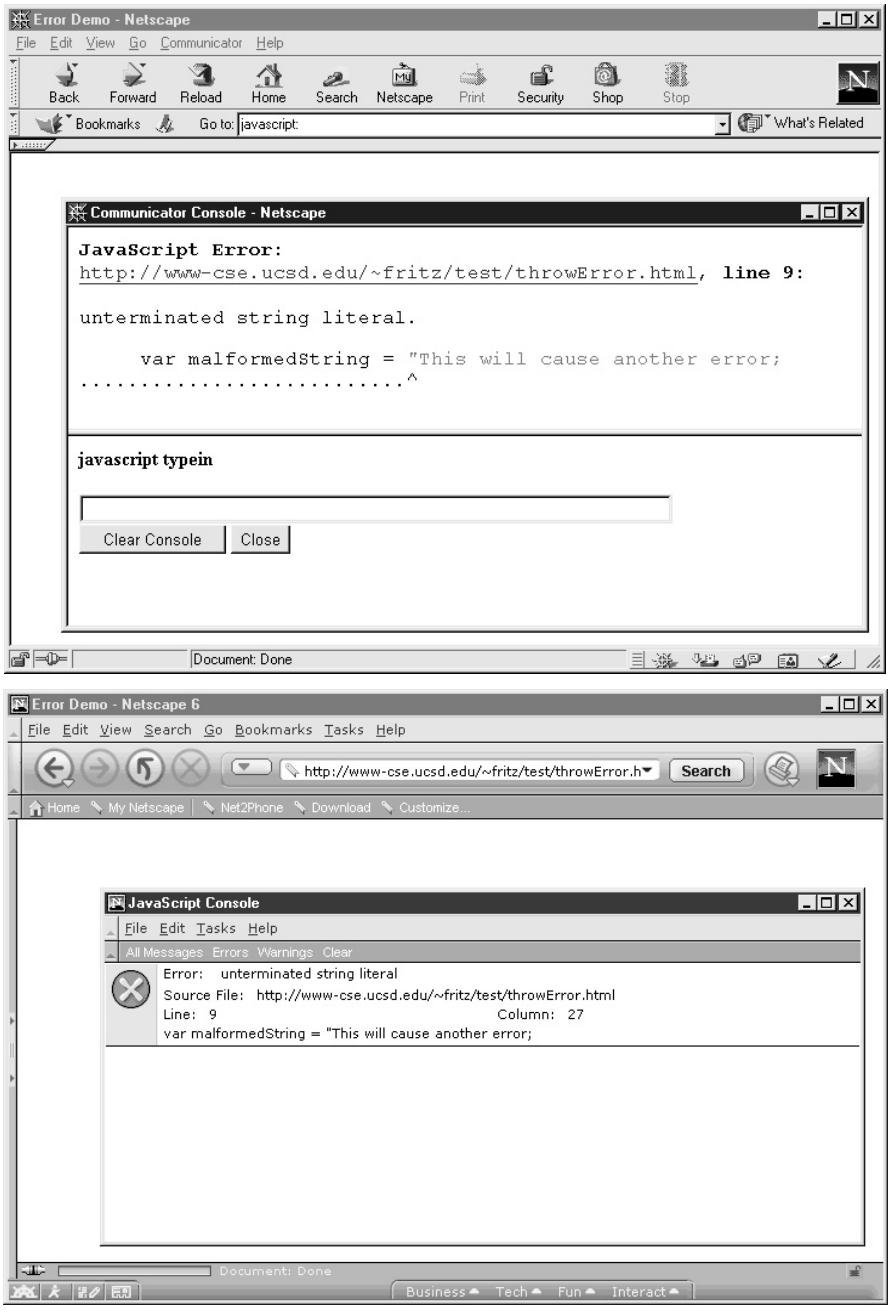


Figure 1-6. *JavaScript console used for debugging and testing*

The JavaScript pseudo-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 by using the **<noscript>** element, as shown here:

```
<noscript>
<b><i>Warning:</i> This page contains links that use JavaScript and your browser
either has JavaScript disabled or does not support this technology.</b>
</noscript>
```

While the JavaScript pseudo-URL does have some limitations, it is commonly found in all major implementations of the language. The final method of including JavaScript in a page (to be discussed next) is not nearly as safe and should generally be avoided.

JavaScript Entities

Netscape 3.x and 4.x generation browsers supported a form of script inclusion called *JavaScript entities*. Under HTML an entity is generally used to insert a special character, such as a copyright symbol, and is indicated using the ***&code;*** notation where *code* is either a keyword or a numeric value corresponding to the character's ASCII value. For example, in HTML we would use either **©** or **©** to insert the copyright symbol. Older Netscape browsers also made it possible to use JavaScript within an entity to simulate a form of a macro. JavaScript entities are indicated using a slightly modified entity style, as in

```
&{script};
```

where *script* is some JavaScript identifier or function call. JavaScript entities can be used only within HTML attributes. For example,

```
<body bgcolor="&{pagebgcolor};" >
```

would set the **bgcolor** attribute of the **<body>** element to the value of "pagebgcolor," which would be some JavaScript variable defined elsewhere in the document. However, do not attempt to use these entities outside of an attribute value; for example,

```
<p>Hello, &{username}; !</p>
```

will only produce the text "&{username};" on the screen instead of the value stored in a JavaScript variable called "username."

Generally, JavaScript entities are used in this simple macro fashion to create attribute values at page load-time, but it is possible to have the entity script call a function or perform multiple statements to compute the final attribute value. The code here illustrates the possible uses of JavaScript entities:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>JavaScript Entities</title>
<script>
var bordersize=5;
var tablecellcolor="#ff0000";
var alignment="center";

function setImage()
 {
  var today = new Date();
  var hours = today.getHours();
  if ((hours > 8) && (hours < 18))
    return 'sun.gif'
  else
    return 'moon.gif';
 }
</script>
</head>
<body>
<table border="&{bordersize};" align="&{alignment};">
<tr>
  <td bgcolor="&{tablecellcolor};">JavaScript Entities!</td>
</tr>
</table>
In the sky now:  <img src="&{setImage()};">
</body>
</html>
```

While JavaScript entities appear useful, they are not supported under standard JavaScript and have never been supported under Internet Explorer. Figure 1-7 shows how different the result of the previous example would be under browsers that support script entities and those that do not.

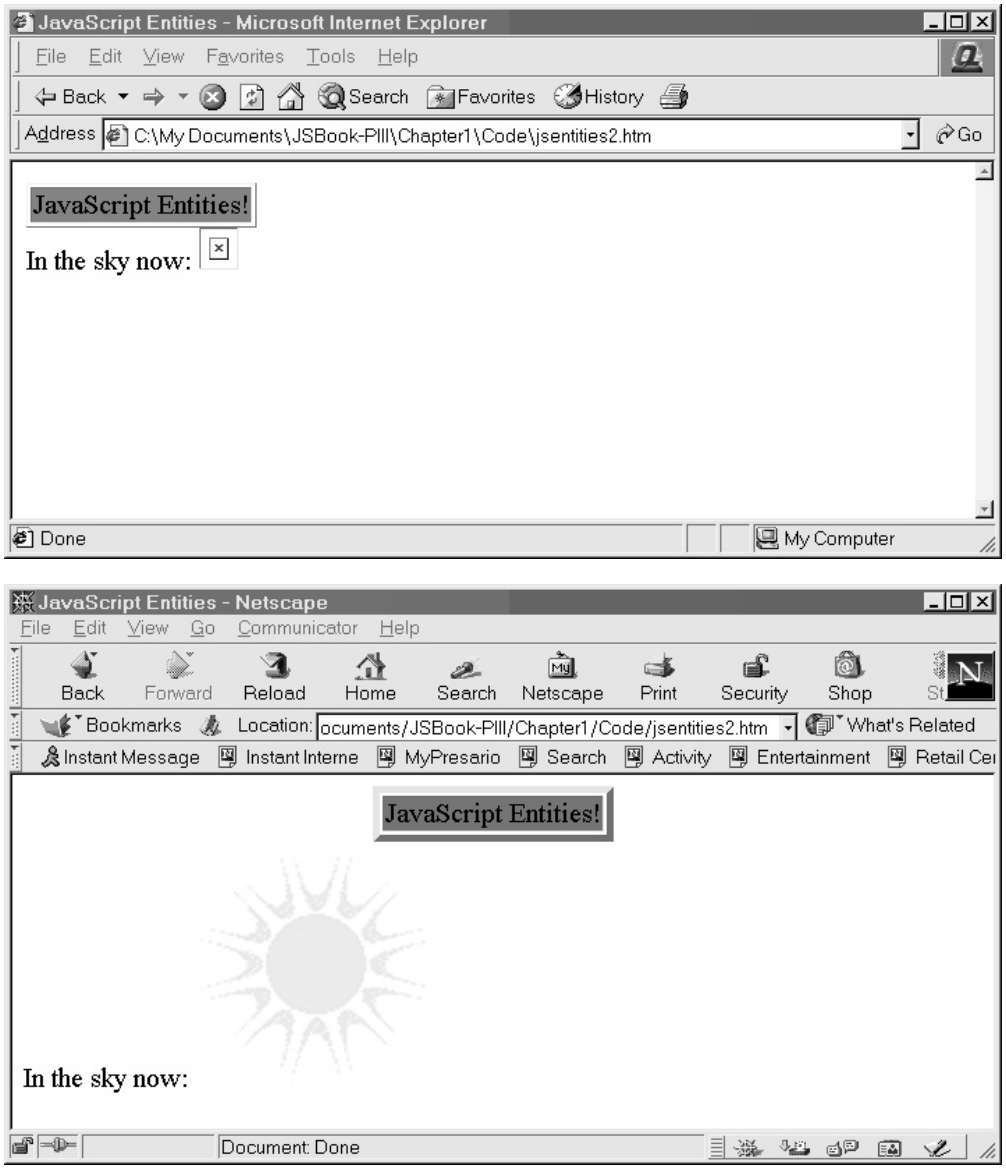


Figure 1-7. *JavaScript Entities are not widely supported*

While entities may appear enticing, they should not be used since they are not supported by newer JavaScript implementations. To accomplish the same types of tasks, use the **document.write()** method to manually output an HTML element complete with attribute values. The code here illustrates how the previous entity example can be made to work under all browsers.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>JavaScript Without Entities</title>
<script>
var bordersize=5;
var tablecellcolor="#ff0000";
var alignment="center";

function setImage()
 {
   var today = new Date();
   var hours = today.getHours();
   if ((hours > 8) && (hours < 18))
     return 'sun.gif'
   else
     return 'moon.gif';
 }
</script>
</head>
<body>
<script>
 document.write('<table border="'+bordersize+'"
align="'+alignment+'">');
</script>
<tr>
<script>
 document.write('<td bgcolor="'+tablecellcolor+'">JavaScript
Without Entities!</td>');
</script>
</tr>
</table>
In the sky now:
<script>
      document.write('<img src="'+setImage()+'">');
```

```
</script>
</body>
</html>
```

The previous sections have detailed the ways that JavaScript will typically be included within an HTML document. Now, before concluding the chapter, let's take a brief look at what such scripts might do.

JavaScript Applications

JavaScript is quite powerful as a client-side technology, but, like all languages, there are limitations on the types of applications it can be used for. Some common uses of JavaScript include:

- Form validation
- Page embellishments and special effects
- Navigation systems
- Basic mathematical calculations
- Dynamic document generation

Historically, the main reason JavaScript came to be was to perform simple checks of form data before submission to a server-side application, such as a CGI program. This is generally known as *form validation*. Since its beginnings, JavaScript has grown in its usefulness, and for many its primary use now is for page embellishments, such as the ubiquitous "mouseover" or rollover button or various types of page animations. The language actually can be used for much more than merely implementing simple gimmicks, and it has been successfully employed to build sophisticated navigation systems, such as drop-down menus or look-up lists. Also, the language has been employed for basic client-side calculations, such as loan calculators and other applications that may be useful for a Web visitor. However, JavaScript's eventual role will be more significant than all these uses put together. Already, JavaScript is used for simple dynamic documentation manipulation, such as conditional inclusion of code or markup on the basis of browser version, and eventually it will be used to modify a page significantly after load. This concept is often called *Dynamic HTML*, or *DHTML* for short. A large focus of this book, particularly in Chapters 9 and 10, will be to explain the implications of DHTML and the Document Object Model (discussed in the next section) that will power the dynamic Web pages of the future.

While it may be possible to write full-blown applications in JavaScript, most programmers will find the bugs, browser differences, and stability of the language

ill-suited for a large application. Remember, it is a scripting language and as such is supposed to be focused in its use. Chapter 2 will present a more complete overview of the use of JavaScript within Web sites and compare it with other technologies, such as CGI, Java, and so on, that could be used. Before concluding the chapter let's take a brief look at the history of JavaScript as it sheds some insight to why the language is the way it is.

History of JavaScript

A study of the evolution of JavaScript is critical for mastering its use, as it can clarify the design motivations for its changes as well as lead to an understanding of its quirks, challenges, and potential as a first-class Web technology. For example, even the name JavaScript itself can be confusing unless you consider its history because, despite the similarity in name, JavaScript has very little to do with Java. In fact, Netscape initially introduced the language under the name LiveScript in an early Beta release of Navigator 2.0 in 1995. 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 to build Web applications. Unfortunately, because of the inclusion of 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 somewhat similar to Java, and syntactically it often has more in common with languages such as C or Perl.

While the name of the language has lead to some confusion among its users, it has been widely adopted by browser vendors. After Netscape introduced JavaScript in version 2.0 of its browser, Microsoft introduced a clone of JavaScript called JScript in Internet Explorer 3.0. Opera also introduced JavaScript support in the 3.x generation of its browser. Many other browsers, including WebTV, also supported various flavors of JavaScript. As time has gone by, each of the major browser vendors has made its 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 this book will deal with such issues, but a concentrated discussion can be found in Chapter 24.

Because each browser originally implemented its own proprietary version of JavaScript, writing code for cross-browser compatibility was a tedious task. To address this issue, a standard form of JavaScript called ECMAScript was specified. 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.

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, 4.5x, 4.6x, 4.7x	1.3
Netscape 6.x	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*

*Still speculative at time of writing, may be renamed JScript 6.0.

Table 1-1. *Browser Versions and JavaScript Support*

Even with the rise of ECMAScript JavaScript can still be challenging to use. ECMAScript is primarily concerned with defining the common statements of the language (such as "if," "for," "while," and so on) as well as the major data types. JavaScript also generally can access a common set of objects related to the browser, such as the window, navigator, history, screen, and others. This collection of objects is often referred to as the *Browser Object Model*, or *BOM* for short. All the browser versions tend to have a different set of objects that make up the BOM, and every new release of a browser seems to include new objects and properties. The BOM finally reached its worst degree of compatibility with the 4.x generation of browsers.

Fortunately, the World Wide Web Consortium (W3C) stepped in to define various objects and interfaces that scripts can utilize to access and manipulate the components of a page in a standardized way. This specification is called the *Document Object Model*, or *DOM* for short. There is some crossover between what is considered BOM and what is DOM, but fortunately in the newer browsers the differences are starting to be ironed out and the three parts of JavaScript are starting to become more 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 provides 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.

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 get started 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 lead to numerous incompatibilities. However, there is hope in sight. With the rise of ECMAScript and the W3C-specified Document Object Model, many of the coding techniques required to make JavaScript code work in different browsers may no longer be necessary.