chapter **two**

Writing Scripts with JavaScript

To do his work well, a workman must first sharpen his tools.

—Chinese proverb

Before we continue our adventure into the world of JavaScript, we need to gather some tools and learn how to integrate our scripts with HTML.

In this chapter, we'll first look at the tools necessary for writing, testing, and running scripts. I'll even make some recommendations. Then we'll discuss the script development process and the various ways in which we can meld JavaScript with HTML, write non-executing comments within our scripts to describe what our code does, and hide JavaScript from browsers that don't understand it. We'll finish off this chapter by creating a script that dynamically writes content to a Web page about the browser currently being used. In a nutshell, we'll

- explore the tools of the trade: editors and browsers.
- look at the script development process.
- describe the scripting workflow.
- scrutinize the <script> tag.
- learn how to integrate JavaScript into Web documents.
- determine where to place scripts and why.
- discover how to hide scripts from old browsers.

- study JavaScript's case sensitivity.
- see how to add comments to our code and learn why we should comment our code.
- play with the properties of the `navigator` object.

Let's get started.

Tools of the Trade

JavaScript programmers don't require much in the way of tools. The most basic JavaScript programmer's toolkit need contain only two things:

- A text editor for writing programs.
- A Web browser in which to run and test programs.

However, a third tool, while optional, can be particularly useful:

- A debugger to help identify and correct programming errors.

Let's take a closer look at text editors.

Text Editors

Every major operating system comes with some sort of text editor. The Windows operating system comes with WordPad and Notepad; Linux operating system installations usually include vi, Pico, or another editor; and Macintosh systems are equipped with Apple's NotePad.

While any text editor will do for writing JavaScript, which is good to know when you're in a pinch and don't have your favorite tools with you, I recommend using a good, non-WYSIWYG (what you see is what you get) HTML editor or an editor designed specifically for programming. Here are some features you should look for in an editor:

- line numbering
- color coding
- search and replace tools
- HTML coding help
- HTML validator
- debugging capabilities

Line numbering is an essential feature you should look for in any editor you consider. When a browser encounters an error in a script it always reports a line number. Being able to turn on line numbers in your editor can help you quickly locate the culprit line and, hopefully, eliminate the pesky bug quickly.

Color coding helps you identify errors as you type. For instance, in HomeSite when you type a quotation mark ("), any text that follows the quotation mark displays in red, reminding you that the quote has not yet been closed. Once you type the closing quotation mark, the text settles to a cool blue, indicating that all is well. This is a very useful feature, because one of the most common errors programmers regularly make is to forget to close a quote.

Search and replace tools can speed up the process of making global changes to a document, for instance, changing a variable name from Visitor to realname.

HTML coding help is a particularly useful feature when you're programming with JavaScript because JavaScript works hand in hand with HTML. You'll also often use JavaScript to write HTML statements. HTML coding help provided by an editor can be in the form of one-click tag insertion, auto completion, tag insight, link checking, or all of the above.

HTML validators are wonderful tools that may save your hair! Sometimes a JavaScript program will not run correctly due to an error in your HTML. A good HTML validator can help you locate and eliminate HTML editors quickly and painlessly, saving you from pulling your hair out in frustration.

Debuggers are sweet little tools that help you find and eradicate—you guessed it—bugs in your programs. A good debugger will have trace features that allow you to step through your script line by line and watch windows that let you see how variable and property values change as the script executes. Debuggers can save you loads of time and energy, once you learn how to use them. Debugging is discussed in detail in Appendix D.

Now that we've discussed the key features to look for when choosing an editor, let's look at some particular editors and their features.

Notepad

Many Web developers still use Notepad to create their Web documents. I don't understand why they continue to put themselves to so much work, typing in all the code themselves, when better tools are available, but that's their choice. One good thing about Notepad: it comes with every Windows operating system. You can always count on a Windows machine to have Notepad installed.

Perhaps the best feature of Notepad lies in its simplicity. Because it supports only very basic formatting, you cannot accidentally save special characters or formatting in documents that need to remain pure text. When working with Web documents, this can be especially useful; some special characters or other formatting may not be visible, but still, just by being present, can cause errors.

Notepad has a simple search and replace capability, but it only works within a single open document. It also has **Go to line**, which allows you to jump directly to a specific line in your document. This can be very useful when debugging. A major limitation of Notepad is that it cannot open large documents.

programmer's tip

Notepad can be very useful for removing special characters and formatting not visible to the naked eye. Simply open the Web document in Notepad that you suspect has a hidden character or hidden formatting in it, and then save it. Notepad will remove any special characters or formatting contained in your document.

WordPad

Another basic text editor that ships with Windows, WordPad includes many word-processing features. It can also open large documents that are too big for Notepad. Otherwise, for

our purposes, WordPad is little better than Notepad for editing Web documents and writing scripts.

An important thing to keep in mind about WordPad is that, by default, it does not save documents in .txt format. This makes it especially easy to accidentally save special formatting in your Web documents. To avoid this, when saving a Web document, choose *Text Document* or *Text Document - MS DOS Format* in the Save As Type box.

HomeSite

This is my personal favorite. While not developed specifically for writing JavaScript programs, HomeSite was created with the myriad tasks of Web developers in mind. HomeSite has everything on our list of key features, except debugging capabilities, including

- Line numbers. Turn them on and off as needed.
- Color coding. This allows you to easily distinguish between text, HTML code, and JavaScript. It indicates when you've forgotten to close a quote or parenthesis. It is completely customizable.
- Search and replace within a document and across multiple documents in a directory and its subdirectories.
- Loads of HTML help, including
  - The ability to insert tags with a click of a mouse button.
  - Tag insight.
  - A link checker.
  - The automatic completion of HTML tags.
  - The automatic insertion of closing HTML tags.
- HTML validator. You can specify which HTML version your code should conform to and set other restrictions according to your own needs and desires.

In addition to most of the key items on our list, HomeSite also features

- The ability to save snippets of code. This is of particular use to JavaScript programmers. With the click of a mouse button, you can insert your tried and proven scripts into new HTML documents instantly. This is one of my favorite HomeSite features.
- A customizable user interface.
- The ability to preview your document in a variety of resolutions with HomeSite's internal browser. You can even set up HomeSite to preview a document in any browser installed on your computer with a click of a button. This makes switching between editing and browsing fast and simple.
- A Cascading Style Sheets editor, TopStyle Lite.
- Project management tools.
- A built-in deployment system. It allows you to quickly and easily upload your entire site or only those documents that have changed.

I've programmed for years in JavaScript using only HomeSite and a Web browser. It works like a charm. HomeSite is available from Macromedia for a 30-day trial (http://www.macromedia.com/software/homesite/). It is shareware, not freeware. It often comes bundled with Macromedia's Dreamweaver. The two make an awesome team for Web development.

BBEdit

If you're using a Macintosh computer, my students recommend BBEdit. They claim it is the best general editor for the Mac hands down. As I don't own a Mac (no groans from the apple bin), I cannot comment on it other than to tell you it was created by Bare Bones Software, hence the BB in the name. A demonstration version is available that works for a limited number of uses (http://www.barebones.com/).

Freeware and Shareware Text Editors

There are, of course, many other shareware and freeware text editors available for downloading on the Internet. Here are a few of the most popular ones:

- TextPad (shareware). A powerful, general-purpose text editor for Windows. You can get the details and download a trial copy at http://www.textpad.com/.
- EditPad (freeware and shareware). Another general-purpose text editor for Windows; this one's by JGsoft. The Lite version is free for non-commercial use. The Pro version has many more features; an evaluation version is also available. You can learn more about both and download them at http://www.editpadpro.com/.
- ConTEXT (freeware). ConTEXT bills itself as the "programmer's editor." The author says, "After years and years searching for [a] suitable Windows text editor, I haven't found any of them to completely satisfy my needs, so I wrote my own." One of the features listed is powerful syntax highlighting for JavaScript and HTML, among other languages. You can learn all about it yourself and download it at http://fixedsys.com/context/.

Web Browsers

A **Web browser** is an essential tool in your JavaScript programmer's toolkit. You need some way to preview your progress and run your scripts. As I discuss each browser, it is not my intent to recommend any one browser over another, but rather to describe the strengths of each. Browser preferences are like computer preferences; there are people who swear by PCs and those who believe Macintosh systems are the only way to compute. While I do have my own browser preference, I think it is important that Web developers have a *variety* of Web browsers at their disposal for testing and previewing their Web documents and JavaScript programs. For instance, on my computer I have a copy of every generation of Netscape Navigator, the latest version of Internet Explorer, and three generations of Opera. This allows me to test my programs and Web pages pretty thoroughly before deploying them to the Web. Ideally, I would also have Macintosh and Unix machines, each with a

variety of browsers installed. Someday. For most people, however, multiple computers and operating systems are just not financially feasible. Regardless of which browser you prefer for your own personal use, testing is a crucial part of any programming or Web development undertaking. So load your computer up with as many as you can. Here's a list of the major browsers in use today.

Netscape Navigator

You would think that the browser that first supported JavaScript would have the best support for JavaScript and contain features for working with JavaScript. You would be right. The latest version of Netscape has, by far, the best and most complete support for the JavaScript programming language. This makes total sense because Netscape, after all, is the company that developed JavaScript in the first place.

As far as special features for working with the language go, Netscape has a built-in JavaScript Console that allows you to test lines of code live and view a list of errors encountered after running a script. For many years, JavaScript Console was the only JavaScript debugging tool available.

You can access JavaScript Console by typing **javascript:** in the location bar of any Netscape browser, version 2 or after.

Figure 2.1 Netscape's JavaScript Console

The Netscape browser changed dramatically when it abandoned its 5.0 browser update in order to focus its efforts on Gecko, a Web page "layout engine" that forms the basis of Netscape's 6.0+ browsers. The browser's layout engine handles the display of Web pages. It parses the HTML and applies stylesheet rules.

Mozilla, a Netscape-sponsored, open-source, Web browser project, also uses Gecko as the layout engine. According to The Mozilla Organization, the Mozilla browser was "designed for standards compliance, performance and portability." Gecko itself was designed to support leaner, faster browsers with good support for existing Web standards.

The latest version of Navigator is available at http://www.netscape.com/. You can learn more about Mozilla and download a copy at http://www.mozilla.org/.

Microsoft Internet Explorer

Microsoft was a latecomer to the Web, but it now has the most widely used browser. Internet Explorer (IE) does a good job of supporting JavaScript, but tends to lag behind Navigator in its support of the *latest* JavaScript features.

If you have a Windows system, you probably already have a copy of Internet Explorer installed on your computer since Microsoft bundles it with the operating system. You can get the latest version at http://www.microsoft.com/windows/ie/default.asp.

Opera

As the newest major entry to the browser market, Opera is quickly gaining market share. It truly is the fastest browser of the three listed here. Opera's makers seem to be dedicated to supporting and sticking to established Web standards. As such, Opera's support for JavaScript began primarily with support for Core JavaScript only; Opera followed the ECMAScript specification. With each new generation, Opera has increased its support for JavaScript.

Opera comes in both adware (not too obtrusive) and commercial versions. Adware is software you can use freely as long as you're willing to put up with little advertisements displaying as you work. Opera displays the ads across the top right of the browser window. Adware is becoming quite popular. Qualcomm now lets everyone use its Eudora Pro email program as adware for free. Eudora displays its ads in the bottom left corner of the program. If you don't like the ads, you can pay a small fee (like with shareware), receive a registration code, and voilà, no more ads! You can get a copy of Opera at http://www.opera.com/.

The Script Development Process

The **script development process** is very similar to the program development process:

1. **Requirements Analysis Phase**: determine needs and requirements.
2. **Design Phase**: formulate a plan that will meet the needs and requirements.
3. **Implementation Phase**: write the code, testing regularly as you work; deploy to the server; and test again.
4. **Support and Maintenance Phase**: maintain and support the finished project.

Entire books have been written on the program development process. We can't possibly cover all of the details and approaches to program development here. Instead, we'll just give a brief overview of each phase.

The Requirements Analysis Phase

Before beginning any project, it's a good idea to first determine the project's requirements. Some good questions to ask during this phase are

- What is the goal? Is there more than one goal?

- What needs must be addressed or answered? What problems need to be solved?
- Who is the Web site's audience? What browser(s) will visitors be using? What size monitors do they have?
- What constraints must you work under? What technologies does the Web site's Internet service provider (ISP) support or not support?

The Design Phase

This is where you plan how to solve the problem, answer the needs, and reach the goals determined during the requirements analysis phase. When planning Web sites, you might sketch what the site will look like, create a storyboard, diagram the site architecture with a site map, outline the content, or all of the above. For scripting, you may need to perform some of those same activities, but you'll also definitely want to consider performing activities specific to program design such as flowcharting or writing pseudocode.

Flowcharts let you visually diagram the flow of your program. Programmers have used them for years to great success. Pseudocode is another useful design tool and the method I most often use to design scripts. Pseudocoding is the process of describing the steps of your program in simple English. One really nice thing about pseudocode is that you can type real code in right next to it, commenting out the pseudocode as you go. When you're done, you have a complete program with detailed comments; your pseudocode becomes your comments and program documentation.

The Implementation Phase

This is where you perform the actual coding, writing JavaScript and HTML as necessary and testing and debugging as you go. When testing, it's a good idea to run your scripts in various browsers and view your pages at different screen resolutions. You'll want to periodically deploy the site and test it online as well.

The Support and Maintenance Phase

Once your project is complete, tested, deployed, and tested again, your project falls into the support and maintenance phase. You may be asked to make minor modifications, add or change content, etc. to keep the Web site fresh and interesting. This may also be when your part in the project ends, depending on how the project was set up. An important thing to note is that when new needs and goals are found, it is time to begin the cycle again with the requirements analysis phase.

OK, we've looked at the various stages of the script development process, but what about the nitty-gritty details of the day-to-day work during the implementation phase? After all, we're here to learn how to code JavaScript. To address this, let's look at the scripting workflow.

The Scripting Workflow

The workflow of writing JavaScript is very similar to the workflow of writing plain old HTML: code it, test it, code it, and test it; in other words, regularly switching back and forth between editor and browser.

This process may already be quite familiar to you. However, for the sake of completeness, I think it's a good idea if we review it anyway.

Code It

Open your text editor. For this example, we'll use Notepad. Create a basic Web page by typing in the following. Do not type the line numbers, just type the code listed in the second column.

```
1   <html>
2   <head>
3         <title>A Basic Web Page</title>
4   </head>
5   <body>
6         <h1>My Web Page</h1>
7   </body>
8   </html>
```

Script 2.1 A Basic Web Page

Save your document as index.html and note what directory you saved it in. Leave Notepad open.

Test It

Do not close Notepad. Start Netscape Navigator and open the document you just created. To do so, choose File from the menu bar and choose Open Page. Browse to index.html. Choose Open. Your Web page should look similar to this:

Figure 2.2 Results of Script 2.1

OK, so far so good.

Code It

Now let's add a little JavaScript. Leave your browser open, and switch to Notepad. You can do this in one of two ways:

- Press Alt + Tab until you get to the Notepad icon.
- Click on Notepad in the taskbar.

Modify your code so it looks like the following:

```
1   <html>
2   <head>
3         <title>A Basic Web Page</title>
4   </head>
5   <body bgcolor="white" text="black">
6         <h1>My Web Page</h1>
7   <script language="JavaScript" type="text/javascript"><!--
8         document.write("Hello, World!")
9   // -->
10  </script>
11  </body>
12  </html>
```

Script 2.2 Adding Some JavaScript

Save your file.

Test It

Leaving Notepad open, switch to your browser by clicking on it in the taskbar or pressing Alt + Tab. Click the Reload button on the browser's toolbar. You should see something similar to this:

Figure 2.3 Results of Script 2.2

Continue the process of coding, testing, coding, and testing. Add some more HTML or try some simple JavaScript statements.

The <script> Tag

When Netscape introduced JavaScript in Netscape Navigator 2, it included support for a new tag: the <script> tag. The <script> tag has several attributes, two of which you should *always* set: `language` and `type`.

The `language` Attribute

The `language` attribute specifies which scripting language you are using. The <script> tag was intended to be programming-language neutral. While JavaScript is the default scripting language supported by most browsers, other languages, including Jscript and VBScript, can also be used in Web documents within the <script> tag.

The language attribute also lets us specify which version of JavaScript we are using. Here are the possibilities and what each indicates:

language=	Result
`JavaScript`	Browsers that support JavaScript 1.0 and later will read and interpret the contents. JavaScript 1.0 support corresponds to Netscape Navigator 2+ and roughly to Internet Explorer 3+ and Opera 3+.
`JavaScript1.1`	Browsers that support JavaScript 1.1 and later will read and interpret the contents. JavaScript 1.1 support corresponds to Netscape Navigator 3+ and roughly to Opera 3.5+.
`JavaScript1.2`	Browsers that support JavaScript 1.2 and later will read and interpret the contents. JavaScript 1.2 support corresponds to Netscape Navigator 4+ and roughly to Internet Explorer 4+.
`JavaScript1.3`	Browsers that support JavaScript 1.3 and later will read and interpret the contents. JavaScript 1.3 support corresponds to Netscape Navigator 4.5+ and roughly to Opera 4+. Core JavaScript corresponds roughly to ECMA Script Version 2.
`JavaScript1.4`	Browsers that support JavaScript 1.4 and later will read and interpret the contents. JavaScript 1.4 support corresponds to Netscape Navigator 5+ (NN 5 was never released). Core JavaScript corresponds roughly to ECMA Script Version 3. Opera 5 supports most of JavaScript 1.4's functionality.
`JavaScript1.5`	Browsers that support JavaScript 1.5 and later will read and interpret the contents. JavaScript 1.5 support corresponds to Netscape Navigator 6+. Opera 6 also supports most of JavaScript 1.5's functionality. Some of JavaScript 1.5 has also been incorporated in the ECMA script version 3 specification.

Table 2.1 `language` Attribute Value Meanings

The `type` Attribute

The `type` attribute specifies the MIME type of the text contained within the <script> tag or the file referenced by the <script> tag. MIME (multipurpose Internet mail extensions) is an extension of the original Internet email protocol that lets people exchange different types of data files on the Internet. Servers insert the MIME header at the beginning of any Web transmission, and browsers use the MIME type listed in the header to choose the appropriate player application for the type of data specified. Some players are built into browsers, such as those needed to display GIFs, JPEGs, and HTML files; others are plug-ins and have to be downloaded, like Adobe Acrobat Reader.

The first part of a MIME type specifies the file type of the data file. The second part, after the slash, indicates the specific type of file the data file represents. For instance, here's a list of common MIME types having to do with Web pages:

Type of File	MIME Type
JavaScript	text/javascript
HTML	text/html
Cascading Style Sheets	text/css
GIF image	image/gif
JPEG image	image/jpeg

Table 2.2 MIME Types for the Web

You should always set the `type` attribute; however, it is particularly important that you set the `type` attribute with the appropriate MIME type whenever you reference an external JavaScript file with the `src` attribute. Otherwise, the browser may not understand what type of file the referenced file is or know what to do with it.

The MIME type and appropriate `type` attribute for JavaScript (any version) is `text/javascript`.

The `src` Attribute

You'll only need to set this attribute when you attach an external JavaScript file. An external JavaScript file is just a simple text document with a .js extension. However, the external JavaScript file may contain only legal JavaScript statements. <script> tags are unnecessary and moreover not allowed inside an external JavaScript file. The <script> tag is HTML code, not JavaScript. We'll talk more about external JavaScript files later.

You may use either a relative or an absolute path to indicate the location of your external JavaScript file. If you use a relative path, that path is relative to the HTML document to which the external script file is being attached.

Integrating JavaScript into Your Web Documents

There are basically five ways to integrate JavaScript into HTML documents:

- In a <script> tag in the head of an HTML document.
- In a <script> tag in the body of an HTML document.
- Inline with HTML as an event handler.
- In an external JavaScript file.
- Inline using the javascript pseudo-protocol.

Let's look at each in turn and specify reasons for each placement.

When programming in JavaScript, you will place scripts in all five areas according to the needs of the task at hand. Each placement area has its own benefits, and there are instances in which a particular area is the best choice. There is no one particular area that is best for *all* coding needs. Likely, you will use multiple areas for every one of your applications.

During our discussion of JavaScript placement, we'll mention topics and use terms that may not mean much to you at this point. Don't worry about it. We'll cover every aspect in detail in future chapters. The main thing for you to get from this section is an idea of how to integrate JavaScript into your Web documents. The details will become clear later as we cover each topic. You may want to refer back to this chapter from time to time as you work your way through the book.

Placing JavaScript Statements in a <script> Tag within the <head>

Why and when should you place scripts in the <head> of an HTML document? The <head> of an HTML document is the perfect place for any statements that need to be read and executed before the contents of your Web document (in the <body> tag) load. This is also a good place to declare **user-defined functions**. A user-defined function is a set of pre-defined, deferred statements that do not execute until the function is **called**. You call a function by invoking its name, followed by an optional set of parameters in parentheses. If the idea of a function seems a little confusing right now, don't worry about it. We'll define and discuss functions in much greater detail in Chapter 8. For now, just file it away in the back of your mind that a <script> tag in the <head> of a document is one good place to write function definitions. By declaring a function in the <head> of an HTML document, you ensure that it is defined and ready to use by the time the <body> of the document loads. The <head> always loads before the <body>.

Global **variables** are also best declared in the <head> of an HTML document, within a <script> tag, of course. A variable is a temporary holding place in memory in which we can store data for future use. We'll define and discuss variables in detail in Chapter 3.

Statements that preload images for use in rollover effects are also most appropriately placed in the <head> of your HTML document within a <script> tag. Basically, the <head>

is the best place to put statements that must execute before the content of your HTML document loads.

Here's an example:

```
1   <html>
2   <head>
3   <title>Placing JavaScript Statements Appropriately</title>
4   <script language="JavaScript" type="text/javascript"><!--
5         var visitor = prompt("What is your name?", "")
6   // -->
7   </script>
8   </head>
9   <body>
10
11  </body>
12  </html>
```

Script 2.3 Placing JavaScript Statements in the <head>

In the above example, we asked the Web site visitor to enter his or her name, and we stored that value in the variable `visitor`. We used the window's `prompt` method, which pops up a prompt dialog box where the question is displayed and a text box is provided for the visitor to enter an answer. Try it for yourself. Type in the above code in your favorite text editor and view it in your browser. You should see the following prompt:

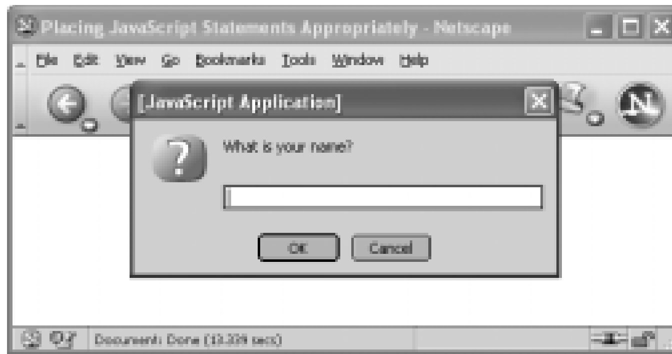

Figure 2.4 Results of Script 2.3

Something to keep in mind when you are writing JavaScript is that you should never place statements that *write* Web page content such as headings, paragraphs of text, tables, lists, etc. in the <head> unless they are part of a function definition. Why? Because Web page content, that is, the text the visitor sees while viewing a Web page, should never be placed in the <head> of an HTML document. Content goes in the <body> of an HTML

document. Only titles, meta tags, and other special data such as scripts and stylesheets are appropriately placed in the <head> tag.

To demonstrate this, create a new document and type in the following:

```
1   <html>
2   <head>
3   <title>Placing JavaScript Statements Illegally</title>
4   <script language="JavaScript" type="text/javascript"><!--
5       document.write("<h1>Hello, I am in the wrong place.</h1>")
6   // -->
7   </script>
8   </head>
9   <body>
10
11  </body>
12  </html>
```

Script 2.4 Placing write Statements in the <head> Illegally

You'll need an old copy of Netscape Navigator (4.8x or prior) to see the problem with this script. When you view the source in an old Netscape Navigator browser (pre-6.0), it shows you the *results* of your `document.write` statements, instead of the `write` statements themselves. This makes it easy to tell if your script is generating the correct HTML statements, stylesheet code, or any other type of string you might be trying to write with a script.

View your file. You should see a blank page or something like this:

Figure 2.5 Results of Script 2.4

Now view the source of the document in Navigator. Navigator shows you the result of your `write` statements, inappropriately written in the <head> of the document. We'll show you the correct place to put `write` statements in the next section.

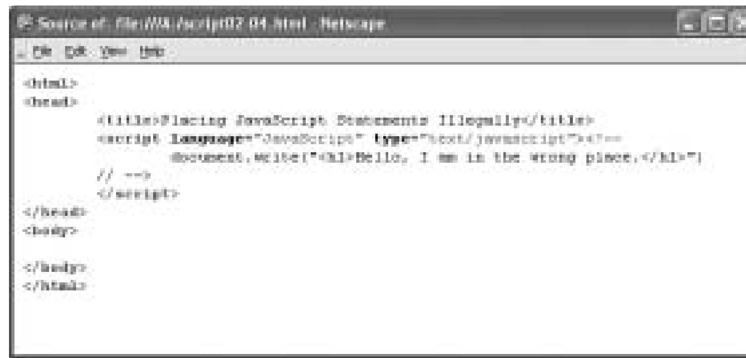

Figure 2.6 Viewing the Source of Script 2.4

Placing JavaScript Statements in a <script> Tag within the <body>

Another place you can write JavaScript statements is in a <script> tag within the <body> of an HTML document. This is the best, and only, place to write statements that actually produce content for inclusion in an HTML document. Calls to functions that write content are also best placed here. We'll explain function calls in Chapter 8.

Let's expand on our earlier example, the one where we prompted the visitor for his or her name. This time, we'll make use of the information we gathered by writing a custom greeting.

```
1   <html>
2   <head>
3   <title>Placing JavaScript Statements Appropriately</title>
4   <script language="JavaScript" type="text/javascript"><!--
5         var visitor = prompt("What is your name?", "")
6   // -->
7   </script>
8   </head>
9   <body>
11  <script language="JavaScript" type="text/javascript"><!--
12        document.write("<h1>Welcome, ", visitor, "</h1>")
13  // -->
14  </script>
```

```
15  </body>
16  </html>
```

Script 2.5 Placing JavaScript Statements in the <body>

Try it. You should see something like this:

Figure 2.7 Prompt for Script 2.5

Go ahead and enter your name. Now a custom welcome should appear, similar to this:

Figure 2.8 Final Results of Script 2.5

Writing JavaScript Statements Inline as Event Handlers

Recall from Chapter 1 that an event handler is one or more JavaScript statements called in response to a particular event. Here's an example that pops up an alert box with the message "Welcome" when the Web document loads:

```
1  <html>
2  <head><title>Writing JavaScript Inline with HTML</title></head>
3  <body onLoad="alert('Welcome!')">
4  </body>
5  </html>
```

Script 2.6 Writing JavaScript Statements Inline

Try it.

We'll discuss events and event handlers in detail in Chapter 5.

Placing JavaScript Statements in an External JavaScript File

An external JavaScript file is a simple text file containing only JavaScript statements whose name has a .js extension. External JavaScript files are another excellent place to declare functions, especially functions you plan to use again and again with a variety of HTML documents.

By placing functions used repeatedly in Web pages throughout your Web site in an external JavaScript file and linking that file to those documents, you can reduce the overall loading time of your Web site. The external JavaScript file will have to transfer only once, the first time the visitor requests a page that uses it. Any future pages that use that file can access it from the cache, eliminating the need to transfer it again.

External JavaScript files also help to hide your code from would-be pilferers. While a savvy Web developer can still view its contents, many do not know how and others will

programmer's tip

To view the source code of an external JavaScript file, follow these steps:

- View the document's source.
- Note the path to the external JavaScript file in the src attribute of the <script> tag, usually found in the <head> of the document.
- Modify the current URL in your location bar to point to the path and file you noted from the `src` attribute.
- Some browsers may require you to add the JavaScript MIME type before allowing you to view the .js file in your Web browser. Internet Explorer will usually let you choose what application to open the file with. Choose Notepad, WordPad, or your favorite HTML editor.
- You can also get the file from your cache. Look for a new file with a .js extension.

simply not go to the trouble. Like how a steering wheel lock deters would-be car thieves, but doesn't completely prevent the determined thief from stealing your car, an external JavaScript file provides one more layer of protection against novice, would-be code stealers.

Using an external JavaScript file, you can begin building a library of frequently used functions and routines. For instance, you might want to gather all of your methods related to form validation into a single formValidation.js library file. Then whenever you need to validate a form, you attach your form-validation library and more than half the battle's won.

Writing JavaScript Statements Inline Using the JavaScript Pseudo-Protocol

We can also write JavaScript statements using the javascript: pseudo-protocol in the `href` attribute of an anchor (<a>) or area (<area>) tag. The idea is that instead of going right to a document or resource, the JavaScript pseudo-protocol will instead execute one or more JavaScript statements, which may or may not return a URL for the anchor tag to follow.

In the Netscape Navigator browser, typing **javascript:** into the location bar will cause JavaScript Console to pop up. JavaScript Console displays the latest error messages the browser encountered while executing scripts. It can be a big help for light debugging. We can also use it to test and experiment with JavaScript statements.

We'll discuss the JavaScript pseudo-protocol in more detail when we cover the click event in Chapter 5. We won't use this method of JavaScript integration right away. In fact, it isn't often used. But keep it in mind for future reference.

While we've put a lot of details off for future discussion, at this point you should understand that there are reasons to place JavaScripts in particular locations according to need and task. As a JavaScript programmer, you will use each of the five ways of integrating JavaScript at one time or another.

Hiding Scripts from Old Browsers

If you've ever looked at other people's scripts, you've probably noticed an odd thing: an HTML comment surrounding the contents of the <script> tag. The closing comment tag is usually preceded by two forward slashes. What's with that?

The truth is, the comment is there for a good reason. To discover that reason, you need first to answer a simple question: if you use a tag that a browser does not recognize or support, what happens to the text contained within the unrecognizable or unsupported tag? For instance, say you place some text in a <blink> tag, then view the document in Internet Explorer. But IE doesn't support the <blink> tag. So what happens to the text? Does it even display in the browser? Sure it does. It just doesn't blink.

OK, so now let's say you write a bunch of JavaScript statements inside a <script> tag. Some browser doesn't recognize your script tag. What does it do? It displays all the JavaScript code contained in the <script> tag in the browser window. Uh oh! Not exactly what you intended to have happen.

That's where that HTML comment comes in. To hide the contents of a <script> tag from old browsers that don't recognize it, simply surround the content with an HTML comment.

```
1   <html>
2   <head>
3         <title>Hiding Scripts from Old Browsers</title>
4         <script language="JavaScript" type="text/javascript"><!--
5               JavaScript statements go here
6         -->
7         </script>
8   </head>
9   <body>
10        <script language="JavaScript" type="text/javascript"><!--
11              JavaScript statements go here
12        -->
13        </script>
14
15  </body>
16  </html>
```

Script 2.7 Hiding <script> Content (not quite right)

There is only one problem with the above: once we get past the opening <script> tag and the opening HTML comment, we're in JavaScript land. Thus, the JavaScript interpreter in the browser will try to execute every statement it comes across until it encounters the closing <script> tag. When it comes to the -->, it runs into an error. The closing comment, -->, has no meaning to the JavaScript interpreter, so it generates an error. To avoid this error, we can comment out the closing HTML comment tag with a JavaScript single-line comment: //. Now our corrected code should look like this:

```
1   <html>
2   <head>
3         <title>Hiding Scripts from Old Browsers</title>
4         <script language="JavaScript" type="text/javascript"><!--
5               JavaScript statements go here
6         // —>
7         </script>
8   </head>
9   <body>
10        <script language="JavaScript" type="text/javascript"><!--
11              JavaScript statements go here
12        // —>
13        </script>
14
```

```
15  </body>
16  </html>
```

Script 2.8 Hiding <script> Content the Right Way

Today, most browsers support JavaScript. So is it really necessary to go to all this trouble anymore? For now, I would say yes. Even though most people use newer browsers that support JavaScript, it's really not that much extra work to make sure that your Web page is accessible to everyone, regardless of what browser they're using.

JavaScript Is Case Sensitive

One important thing you need to know about JavaScript is that it is case sensitive. That means that

```
Document.Write("Hello")
```

and

```
document.write("Hello")
```

are *not* equivalent.

The first means nothing to JavaScript, the second calls the `write` method of the `document` object. This is extremely important to keep in mind. As we continue through this book, we'll see many examples of how JavaScript's case sensitivity affects what we do.

Commenting Your Code: How and Why

I mentioned in an earlier section, while discussing pseudocode, the concept of commenting your programs. Commenting your scripts as you write them is a good habit to get into, especially any place you use tricky or complex coding. Not only is it helpful to you when you revisit code six months or a year later to clue you in on what you were doing and why you approached it that way, but also it is helpful to anyone else that might have to modify or enhance your code.

In today's Web development world, it is quite common for a team of people to work together on a project. With the high turnover rate in the information technology industry in general, it is also likely that someone else will take over editing and maintaining your code when you move up the ladder. Your well-commented code could win friends and influence people. Well, at least it will make the job of those coming after you easier and they'll think better of you as a programmer.

JavaScript supports two types of comments: single-line comments and multi-line comments.

Single-line Comments

Single-line comments are designated with two slash marks in a row (//). You'll likely use single-line comments the most often because you can place them on the same line as the code on which they comment. For example:

```
var visitor                                // Visitor's name
visitor = prompt("Enter your name: ", "") // Get visitor's name
```

I used two single-line comments above to describe what each line of code is doing. It is not necessary to comment every line of code. For instance, `document.write` statements are often self-explanatory and need no further comment. However, you should get in the habit of describing each new variable as you declare it and commenting on code that is not self-explanatory at first glance. While learning JavaScript, you might want to consider commenting every line of code until you become more familiar with the language.

Multi-line Comments

Multi-line comments begin with a slash and an asterisk (/*) and end with an asterisk and a slash (*/). They are most often used to

- Provide reference information for an entire script or library, such as title, author, last update, etc.
- Describe a block of code or user-defined function.
- List copyright notices and script license information.

For example:

```
/* Form Validation Library
   Author: Tina McDuffie     */
```

You should provide a multi-line comment at the beginning of every external JavaScript file. You should also use multi-line comments to describe each function definition and to list any important notices about code such as copyright.

To Semicolon or Not to Semicolon

Many programming languages such as C, C++, Java, Perl, and PHP require the use of semicolons as statement terminators. Not JavaScript. According to the author of the JavaScript language, Brendan Eich, "Requiring a semicolon after each statement when a new line would do, [was] out of the question—scripting for most people is about writing short snippets of code, quickly and without fuss."

The only time you are *required* to use a semicolon in JavaScript is when you place more than one statement on a single line. Here's an example:

```
document.write("Hello, "); document.write("World")
```

You could just as easily have written the statements as

```
document.write("Hello, ")
document.write("World")
```

which, in my opinion, is more readable. Of course, you could've combined the two statements into one, too. The only time you're really likely to want to write two statements on a single line is in an event handler. We'll talk more about event handlers in Chapter 5.

To make the code simpler and give you one less thing to worry over, we've left the semicolons out of the scripts in this book, except where they're required, like in the aforementioned event handlers when you *need* more than one statement on a single line.

Introducing the `navigator` Object

Let's practice some of what we've learned by using JavaScript to write information about a visitor's browser. To do this, we'll need to make use of the `navigator` object. We worked with it a little in Chapter 1.

The `navigator` object gets its name from the browser that first supported JavaScript: Netscape Navigator. You can think of the `navigator` object as a browser object, as its properties describe the current browser.

Before we begin writing a script, look up the `navigator` object again in Appendix A. What properties are listed? Can you make a guess at what any of them mean? Table 2.3 describes each of them in turn.

Property	Description
appCodeName	The browser's internal code name.
appName	The browser's name.
appVersion	The browser's version, the platform on which it is running, and the country.
language	Specifies which human language the browser supports.
mimeTypes[ ]	A list of the MIME types supported by the browser.
platform	The platform on which the browser is running. Possible values for PCs and Macs are Win32, Win16, Mac68k, and MacPPC.
plugins[ ]	A list of all of the plug-ins currently installed on the browser.
userAgent	"User agent" is another name for a browser. This property specifies the user-agent header. Web servers often collect it.

Table 2.3 Properties of the `navigator` Object

Let's construct a script to report information about the browser currently being used. To do this we'll use the `document.write` method and the `navigator` properties listed above. Open Notepad or your favorite editor and type in the following script:

```
1   <html>
2   <head>
3         <title>Reporting Browser Information</title>
4   </head>
5   <body>
6   <h1>Your Browser</h1>
```

```
7   <script language="JavaScript" type="text/javascript"><!--
8     document.write("<b>appCodeName:</b> ", navigator.appCodeName,
9         "<br>")
10    document.write("<b>appName:</b> ", navigator.appName, "<br>")
11    document.write("<b>appVersion:</b> ", navigator.appVersion,
12        "<br>")
13    document.write("<b>language:</b> ", navigator.language,
14        "<br>")
15    document.write("<b>platform:</b> ", navigator.platform,
16        "<br>")
17    document.write("<b>userAgent:</b> ", navigator.userAgent,
18        "<br>")
19  // -->
20  </script>
21  </body>
22  </html>
```

Script 2.9 Reporting Browser Information

Save it and view it in as many different browsers as you have installed on your computer. Here's what the results look like when run in Netscape Navigator 7, Netscape Navigator 4.75, Internet Explorer 6.0, and Opera 6.05.

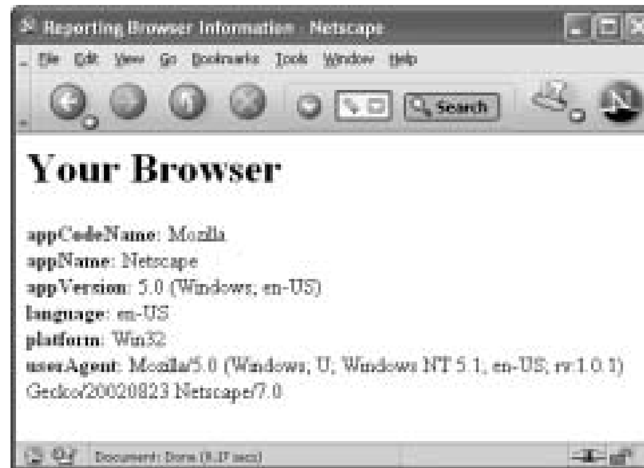

Figure 2.9 Results of Script 2.9 in Netscape Navigator 7

Figure 2.10 Results of Script 2.9 in Netscape Navigator 4.75

Figure 2.11 Results of Script 2.9 in Internet Explorer 6.0

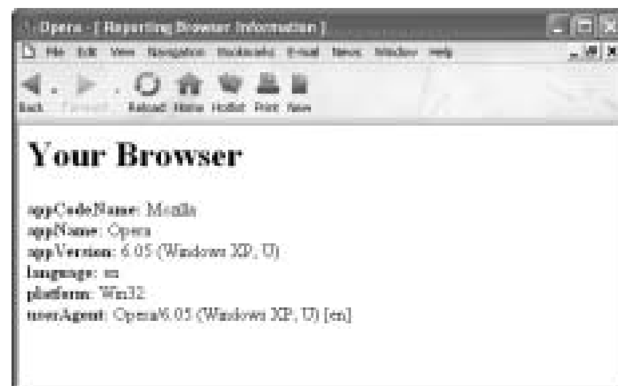

Figure 2.12 Results of Script 2.9 in Opera 6.05

As you can see, all four browsers reported the appCodeName as Mozilla. If we were trying to perform browser detection, appCodeName wouldn't be of much use. However, notice that the appName property does a better job of distinguishing among the browsers, as does userAgent. The "en" reported as the language means English. Netscape 7 further distinguished the language as United States English.

One browser, Opera, has the capability of being able to report one of several different browser identifications, depending on which choice you make in Opera's preferences. Opera has the ability to emulate other browsers. This can make it difficult to determine for certain whether or not a visitor is using Opera. Here's an example of what Opera reports when in Internet Explorer mode. To change the mode in Opera, choose File on the menu bar and choose Preferences. In Network, choose "Identify as MSIE 5.0" under Browser identification.

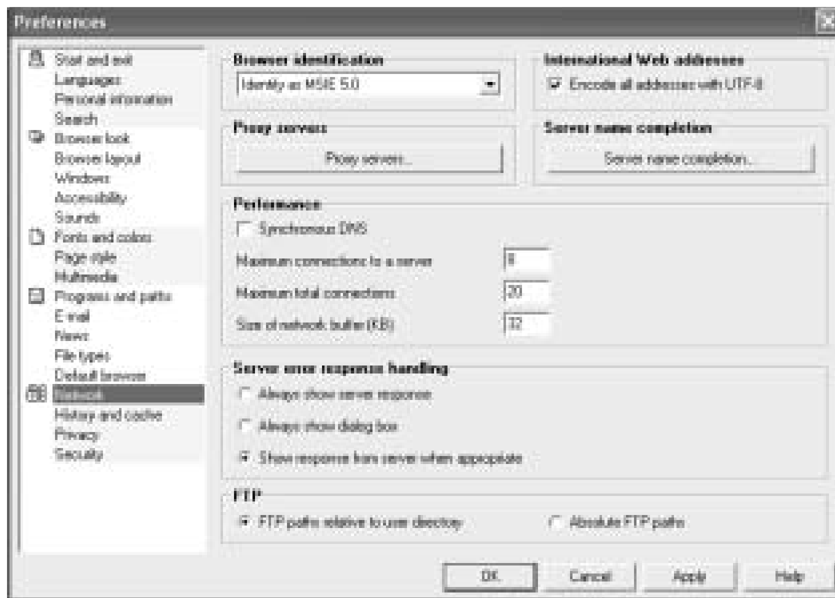

Figure 2.13 Setting the Browser Identification in Opera 6.05

Here's what Opera 6.05 reports in IE mode:

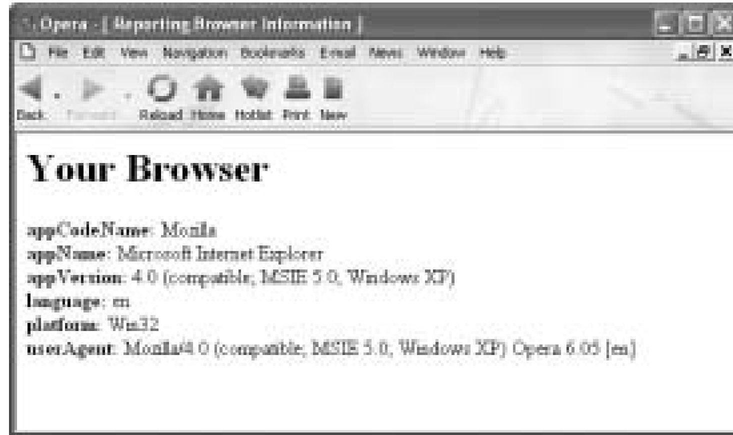

Figure 2.14 Results of Script 2.9 in Opera 6.05 in IE Mode

Summary

In this chapter we looked at the tools of the trade for programming in JavaScript. There aren't many: a simple text editor for writing code and a browser for viewing the results are all that are required. However, there are better alternatives with a little more punch and utility. My personal favorite is HomeSite.

After examining the features of a few editors and browsers, we looked at the script development process, which includes the following phases:

1. Requirements Analysis Phase: determine needs and requirements.
2. Design Phase: formulate a plan that will meet the needs and requirements.
3. Implementation Phase: write the code, testing regularly as you work; deploy to the server; and test again.
4. Support and Maintenance Phase: maintain and support the finished project.

The workflow for scripting JavaScript is very similar to that of writing plain old HTML: primarily a continuous cycle of coding, testing in the browser, coding, and testing in the browser.

Next we looked at the attributes of the <script> tag and the various places we could put JavaScript statements:

- In a <script> tag in the <head> of an HTML document. This is the best place to put statements that must execute before the page content displays. This is also a good place to declare functions and global variables.
- In a <script> tag in the <body> of an HTML document. This is the best and only location to place statements that will actually write Web page content.
- Inline with HTML as an event handler. It is used to call an event handler.

- In an external JavaScript file. This is a good place for frequently used functions. It is also a way to help hide your code somewhat.
- Inline using the javascript pseudo-protocol. This is used in href attributes only.

We discovered how and why to hide the contents of our <script> tags from old browsers. JavaScript, it turns out, is case sensitive, so we need to be conscious of the type case we use when writing scripts. It is also a good idea to comment our scripts as we write them. JavaScript supports two kinds of comments: single-line (//) and multi-line (/* . . . */).

Finally, we put what we've learned to work and wrote a script that displays information about the browser the visitor is using. To test it, we ran our script in several different browsers and noted the differences in each.

Review Questions

1. At a minimum, what tools does a JavaScript programmer need to code and test a Web page that uses JavaScript?
2. What features does an editor like HomeSite offer that make it a more desirable alternative to Notepad?
3. Why do you need a Web browser for developing scripts?
4. How can JavaScript Console, which comes built into Netscape Navigator, assist a JavaScript programmer?
5. What is Opera?
6. Describe the script development process.
7. Describe the major goal and processes involved in each phase of the script development process.
8. Describe each of the following attributes of the <script> tag:
   a. language
   b. type
   c. src
9. What is MIME?
10. What are the five ways in which you can integrate JavaScript into your Web pages; that is, where can you put JavaScript statements?
11. For each of the places you listed in question 10, list an example of the type of task that makes that particular place the best choice.
12. Why should you hide scripts from old browsers? What happens if you don't?
13. How can you hide scripts from old browsers that don't support the <script> tag?
14. What does it mean to say that JavaScript is case sensitive?
15. What two types of comments does JavaScript support? For each type, explain how to write that type of comment in JavaScript and when you would most likely use that type of comment.
16. Name and describe three properties of the `navigator` object.
17. Why is the `navigator` object called "navigator" and not "browser"?
18. Why does Script 2.9 display different results in different browsers?

19. Which browser reports different results for Script 2.9 depending on the user's preferences?
20. Why is it a good idea to have multiple browsers and multiple versions of each browser?

Exercises

1. Explain when you should attach an external JavaScript file and when it is better to embed JavaScript in the HTML document.
2. Write a set of empty script tags complete with the appropriate attributes and necessary notation to hide the script's contents from old browsers.
3. Write the appropriate <script> tag to attach an external JavaScript file named myLib.js located in the scripts subdirectory off of the root of the Web site. Assume the document you are attaching the script to is in the root as well.
4. Write the appropriate <script> tag to attach an external JavaScript file named myLib.js located in the scripts subdirectory off of the root of the Web site. Assume the document you are attaching the script to is in a subdirectory named products off of the root.
5. Research text editors on the Web. Find three that look like good ones to you and compare their features. Summarize your findings. How much does each cost? Were you able to find any really good freeware text editors?
6. Research debugging on the Web. What is it? See if you can find any good debugging pointers specifically for JavaScript. Summarize what you've learned.

Scripting Exercises

Create a folder named assignment02 to hold the documents you create during this assignment. Save a copy of the personal home page you created at the end of Chapter 1 as home.html in the assignment02 folder.

1. Create a simple JavaScript program that writes "Greetings, Earthlings!" in an <h1> tag and an image that displays a picture of an alien, Marvin the Martian, or a spaceship. (You should have no trouble finding one online.) All HTML code within the <body> should be created with JavaScript. At the bottom of the document, write "This document was created with JavaScript." Save the document as aliens.html.
2. Create a Web page named docColors.html with a white background, black text, blue links, purple visited links, and red active links. Insert a script that writes the values of the document's color settings. Acquire the color settings from the corresponding properties of the `document` object. See Appendix A for a list of document properties.
3. Modify docColors.html so that at the end of the document, the background color is changed to a color of your choice. Save it as docColors2.html.
4. Insert a script into home.html that writes a new section heading and a blockquote, like this:

```
<h2>My Favorite Quote</h2>
<blockquote>your favorite quote goes here</blockquote>
```

Place your favorite quote between the <blockquote> tags.

5. Add a script to home.html that writes a numbered list of your hobbies and interests under the My Hobbies heading. Use JavaScript to write the appropriate <ol> and <li> tags and their content.