CHAPTER 3

Your First JavaScript Script

In this chapter, you set up a productive script- writing and previewing environment on your computer — and then write a simple script whose results you will see in your JavaScript-compatible browser.

Because of differences in the way various personal computing operating systems behave, I present details of environments for two popular variants: Windows 95 and the MacOS. For the most part, your JavaScript authoring experience will be the same regardless of the operating system platform you use — including UNIX. Although there may be slight differences in font designs depending on your browser and operating system, the information will be the same. All illustrations of browser output in this book are made from the Windows 95 version of Netscape Navigator 4. If you're running another version of Navigator, don't fret if every pixel doesn't match with the illustrations in this book.

In This Chapter

Choosing basic JavaScript authoring tools

Setting up your authoring environment

Entering a simple script to a Web page

The Software Tools

The best way to learn JavaScript is to type the HTML and scripting code into documents in a text editor. Your choice of editor is up to you, though I provide you with some guidelines for choosing a text editor in the next section. HTML files are raw ASCII text files, and any scripting has to be done in those files. While learning JavaScript, you may be better off using a simple text editor rather than a high-powered word processor with HTML extensions.

Choosing a text editor

For the purposes of learning JavaScript in this book, avoid WYSIWYG (What You See Is What You Get) Web-page authoring tools for now. These tools will certainly come in handy afterward, when you can productively use those facilities for molding the bulk of your content and layout. But the examples in this book focus more on script content (which you must type in anyway), so there won't be much HTML that you have to type. Files for all complete Web page listings are also included on the companion CD-ROM.

An important factor to consider in your choice of editor is how easy it is to save standard text files. In the case of Windows, any program that not only saves the file as text by default but also lets you set the extension to .htm or .html will prevent a great deal of problems. If you were to use Microsoft Word, for example, the program tries to save files as binary Word files — something that no Web browser can load. To save the file initially as a text or .html extension file requires mucking around in the Save As dialog box. This requirement is truly a nuisance.

Nothing's wrong with using bare-essentials text editors. In Windows 95, that includes the WordPad program or a more fully featured product such as the shareware editor called TextPad. For the MacOS, SimpleText is also fine, though the lack of a search-and-replace function may get in the way when you start managing your Web site pages. A favorite among Mac HTML authors and scripters is BBEdit (Bare Bones Software), which includes a number of useful aids for scripters, such as optional line numbers (which help in debugging JavaScript). In Chapter 46, I describe a Windows-based JavaScript authoring tool named Infuse 2.0 by Acadia Software. This product includes a nice text editor window for entering scripts.

Choosing a browser

The other component required for learning JavaScript is the *browser*. You don't have to be connected to the Internet to test your scripts in the browser. You can do all of it offline. This means you can learn JavaScript and create cool-scripted Web pages with a laptop computer, even on a boat in the middle of an ocean.

The only requirement for the browser is that it be compatible with the current release of JavaScript. The instructions and samples in this book are designed for Netscape's Navigator 4. Microsoft's Internet Explorer 4 supports most of the core language of Navigator 4, but you'll still find occasional incompatibilities.

Setting up your authoring environment

To make the job of testing your scripts easier, make sure that you have enough free memory in your computer to let both your text editor and browser run simultaneously. You need to be able to switch quickly between editor and browser as you experiment and repair any errors that may creep into your code. The typical workflow entails the following steps:

1. Enter HTML and script code into the source document.
2. Save the latest version to disk.
3. Switch to the browser.
4. Do one of the following: If this is a new document, open the file via the browser's Open menu. Or if the document is already loaded, reload the file into the browser.

Steps 2 through 4 are the key ones you will follow frequently. I call this three-step sequence the *save-switch-reload sequence*. You will perform this sequence so often as you script that the physical act will quickly become second nature to you. How you arrange your application windows and effect the save-switch-reload sequence varies according to your operating system.

Windows

You don't have to have either the editor or browser window maximized (at full screen) to take advantage of them. In fact, you may find them easier to work with if you adjust the size and location of each window so both windows are as large as they can be while still enabling you to click on a sliver of the other's window. Or, in Windows 95, leave the taskbar visible so you can click the desired program's button to switch to its window (Figure 3-1). A monitor that displays more than 640 × 480 pixels certainly helps in offering more screen real estate for the windows and the taskbar.

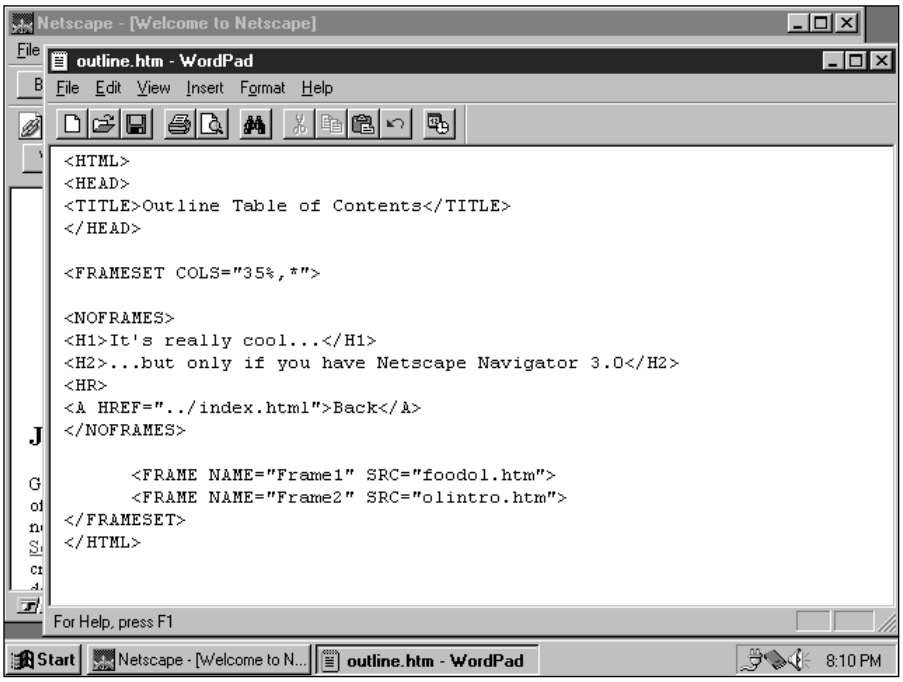

Figure 3-1: Editor and browser window arrangement in Windows 95

In practice, however, the Windows Alt+Tab task-switching keyboard shortcut makes the job of the save-switch-reload steps outlined earlier a snap. If you're running Windows 95 and also using a Windows 95-compatible text editor (which more than likely has a Ctrl+S file-saving keyboard shortcut) and the Netscape Navigator 4 browser (which has a Ctrl+R reload keyboard shortcut), you can effect the save-switch-reload from the keyboard, all with the left hand: Ctrl+S (save the source file); Alt+Tab (switch to the browser); Ctrl+R (reload the saved source file).

As long as you keep switching between the browser and text editor via Alt+Tab task switching, either program is always just an Alt+Tab away.

MacOS

If you expand the windows of your text editor and browser to full screen, you have to use the rather inconvenient Application menu (right-hand icon of the menu bar) to switch between the programs. A better method is to adjust the size and location of the windows of both programs so they overlap, while allowing a portion of the inactive window to remain visible (Figure 3-2). That way, all you have to do is click anywhere on the inactive window to bring its program to the front.

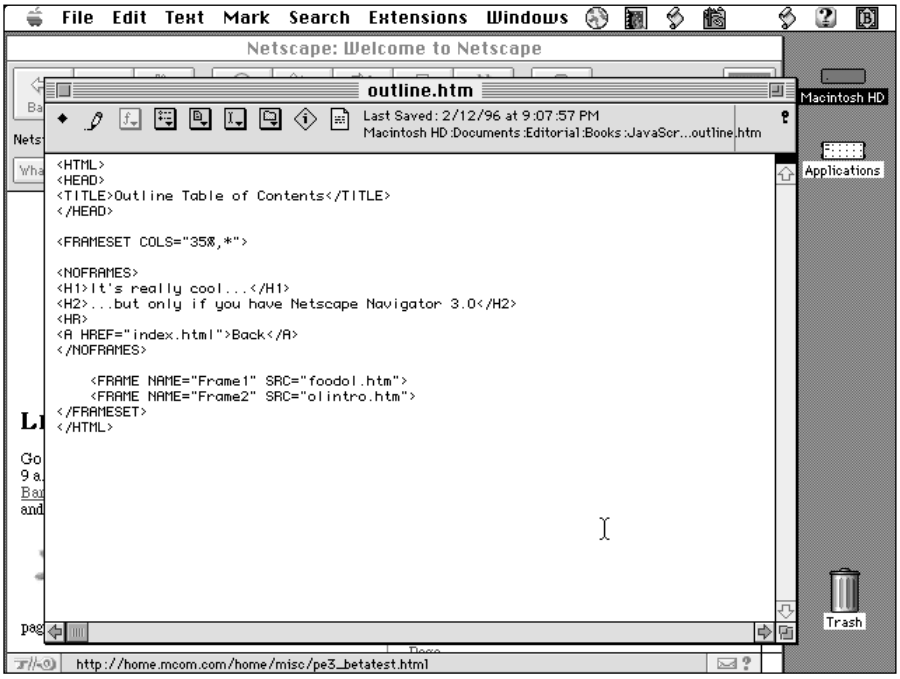

Figure 3-2: Editor and browser window arrangement on the Macintosh screen

With this arrangement, the save-switch-reload sequence is a two-handed affair. Assuming that you have Navigator 4 (which enables you to reload the current URL with a ⌘-R keyboard shortcut), the sequence is as follows:

1. Press ⌘-S (save the source file).
2. Click in the browser window.
3. Press ⌘-R (reload the saved source file).

To return to editing the source file, click on any exposed part of the text editor's window.

A useful utility called Program Switcher (`http://www.kamprath.net/claireware`) puts the Alt+Tab program switching functionality on the Mac keyboard. It is more convenient than using the Application menu.

What Your First Script Will Do

For the sake of simplicity, the kind of script you will be looking at is the kind that runs automatically when the browser opens the HTML page. Although all scripting and browsing work done here is offline, the behavior of the page would be identical if you placed the source file on a server and someone were to access it via the Web.

Figure 3-3 shows the page as it appears in the browser after you're finished (the exact wording differs slightly if you're running your browser on an operating system platform other than Windows 95 or NT, or if you're using a browser other than Netscape Navigator). The part of the page that is defined in regular HTML contains nothing more than an `<H1>`-level header with a horizontal rule under it. If someone were not using a JavaScript-equipped browser, all he or she would see is the header and horizontal rule (unless that person had a truly outmoded browser, in which case some of the script words would appear in the page).

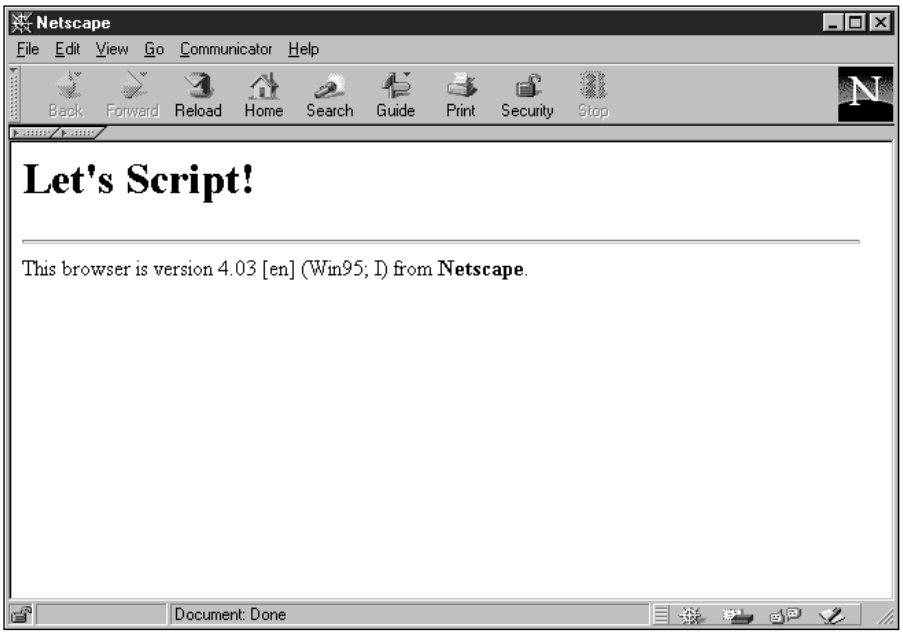

Figure 3-3: The finished page of your first JavaScript script

Below the rule, the script displays plain body text that combines static text with information about the browser you used to load the document. The script writes a stream of HTML information to the browser, including a tag to render a portion of the information in boldface. Even though two lines of code are writing information to the page, the result is rendered as one line, just as it would be if all the text had been hard-coded in HTML.

Entering Your First Script

It's time to start creating your first JavaScript script. Launch your text editor and browser. If your browser offers to dial your Internet service provider (ISP) or begins dialing automatically, cancel or quit the dialing operation. If the browser's Stop button is active, click it to halt any network searching it may be trying to do. You may receive a dialog box message indicating that the URL for your browser's home page (usually the home page of the browser's publisher — unless you've changed the settings) is unavailable. That's fine. You want the browser open, but you shouldn't be connected to your ISP. If you're automatically connected via a local area network in your office or school, that's also fine, but you won't be needing the network connection for now. Next, follow these steps to enter and preview your first JavaScript script:

1. Activate your text editor and create a new blank document.
2. Type the script into the window exactly as shown in Listing 3-1.

Listing 3-1: Source Code for script1.htm

```
<HTML>
<HEAD>
<TITLE>My First Script</TITLE>
</HEAD>

<BODY>
<H1>Let's Script...</H1>
<HR>
<SCRIPT LANGUAGE="JavaScript">
<!-- hide from old browsers
document.write("This browser is version " + navigator.appVersion)
document.write(" of <B>" + navigator.appName + "</B>.")
// end script hiding -->
</SCRIPT>
</BODY>
</HTML>
```

3. Save the document with the name *script1.htm*. (This is the lowest common denominator file-naming convention for Windows 3.1 — feel free to use an .html extension if your operating system allows it.)
4. Switch to your browser.
5. Choose Open File from the File menu and select script1.htm.

If you typed all lines as directed, the document in the browser window should look like the one in Figure 3-3 (with minor differences for your computer's operating system and browser version). If the browser indicates that a mistake exists somewhere as the document loads, don't do anything about it for now (click the OK button in the script error dialog box). Let's first examine the details

of the entire document so you understand some of the finer points of what the script is doing.

Examining the Script

You do not need to memorize any of the commands or syntax that I discuss in this section. Instead, relax and watch how the lines of the script become what you see in the browser. In Listing 3-1, all of the lines up to the <SCRIPT> tag are very standard HTML. Your JavaScript-enhanced HTML documents should contain the same style of opening tags you normally use.

The <SCRIPT> tag

Any time you include JavaScript verbiage in an HTML document, you must enclose those lines inside a <SCRIPT>. . .</SCRIPT> tag pair. These tags alert the browser program to begin interpreting all the text between these tags as a script. Because other scripting languages, such as Microsoft's VBScript, can take advantage of these script tags, you must specify the precise name of the language in which the enclosed code is written. Therefore, when the browser receives this signal that your script uses the JavaScript language, it uses its built-in JavaScript interpreter to handle the code. You can find parallels to this setup in real life: If you have a French interpreter at your side, you need to know that the person with whom you're conversing also knows French. If you encounter someone from Russia, the French interpreter won't be able to help you. Similarly, if your browser has only a JavaScript interpreter inside, it won't be able to understand code written in VBScript.

Now is a good time to instill an aspect of JavaScript that will be important to you throughout all your scripting ventures: JavaScript is case-sensitive. Therefore, any item in your scripts that uses a JavaScript word must be entered with the correct uppercase and lowercase letters. Your HTML tags (including the <SCRIPT> tag) can be in the case of your choice, but everything in JavaScript is case-sensitive. When a line of JavaScript doesn't work, wrong case is the first thing to look for. Always compare your typed code against the listings printed in this book and against the various vocabulary entries discussed throughout it.

A script for all browsers

The next line after the <SCRIPT> tag in Listing 3-1 appears to be the beginning of an HTML comment tag. It is, but the JavaScript interpreter treats comment tags in a special way. Although JavaScript dutifully ignores a line that begins with an HTML comment start tag, it treats the next line as a full-fledged script line. In other words, the browser begins interpreting the next line after a comment start tag. If you want to put a comment inside JavaScript code, the comment must start with a double slash (//). Such a comment may go near the end of a line (such as after a JavaScript statement that is to be interpreted by the browser) or on its own line. In fact, the latter case appears near the end of the script. The comment line starts with two slashes.

Step back for a moment and notice that the entire script (including comments) is contained inside a standard HTML comment tag (<!--comment-->). The value

of this containment is not clear until you see what happens to your scripted HTML document in a non–JavaScript-compatible browser. Such a browser would blow past the <SCRIPT> tag as being an advanced tag it doesn't understand. But it would treat a line of script as regular text to be displayed in the page. By enclosing script lines between HTML comment tags, most older browsers won't display the script lines. Still, some old browsers can get tripped up and present some ugliness because they interpret any > symbol (not the whole --> symbol) to be an end-of-comment character. Figure 3-4 shows the results of your first script when viewed in a now-obsolete version of the America Online Web browser (version 2.5 for Windows).

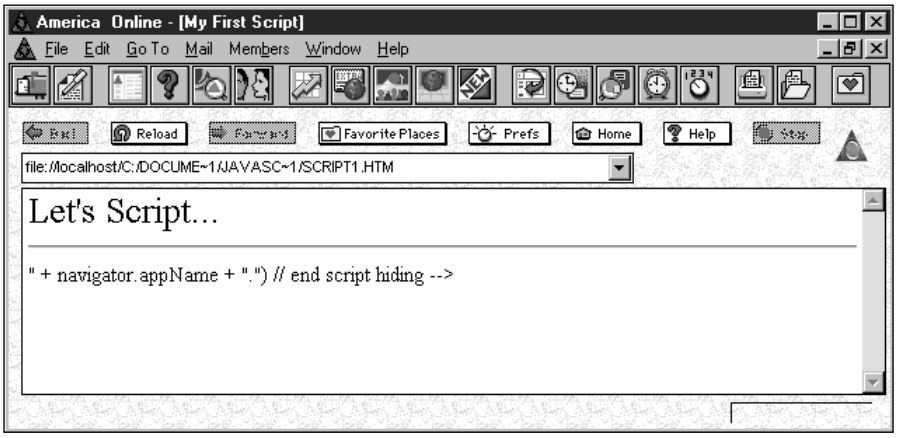

Figure 3-4: By enclosing script lines between HTML comments, the entire script is ignored by most, but not all, non-JavaScript browsers. Here, an old America Online browser shows part of the script anyway.

Remember, too, that some users don't have access to modern browsers or graphical browsers (they use the Lynx text-oriented UNIX Web reader software or Lynx-like browsers in hand-held computers). By embracing your script lines within these comments, your Web pages won't look completely broken in relatively modern non-JavaScript browsers.

Notice that the comment lines that shield older browsers from your scripts go inside the <SCRIPT>...</SCRIPT> tags. Do not put these comment lines above the <SCRIPT> tag or below the </SCRIPT> tag and expect them to work.

One more issue about the script-hiding comment lines in this book. To save space on the page, most examples do not have comment lines inserted in them. But as you can see in the full-fledged application examples on the CD-ROM (in the folder named Bonus Applications Chapters), the comment lines are where they should be. For any pages you produce for public consumption, always encase your script lines inside these comments.

Displaying some text

Both script lines use one of the possible actions a script can ask a document to perform (`document.write()`, meaning display text in the current document). You learn more about the document object in Chapter 16.

Whenever you ask an object (a document in this case) to perform a task for you, the name of the task is always followed by a set of parentheses. In some cases — the `write()` task, for example — JavaScript needs to know what information it should act on. That information (called a *parameter*) goes inside parentheses after the name of the task. Thus, if you want to write the name of the first U.S. president to a document, the command to do so is

```
document.write("George Washington")
```

The line of text that the script writes starts with some static text (`"This browser is version`") and adds some evaluated text (the version of the browser) to it. The writing continues with more static text that includes an HTML tag (`"of <B>"`), more evaluated text (the name of the browser application), and ends with an HTML closing tag and the sentence's period (`"</B>."`). JavaScript uses the plus symbol (+) to join (*concatenate*) text components into a larger, single string of text characters to be written by the document. Neither JavaScript nor the + symbol knows anything about words and spaces, so the script is responsible for making sure that the proper spaces are passed along as part of the parameters. Notice, therefore, that an extra space exists after the word "version" in the first `document.write()` parameter, as well as spaces on both sides of "of" in the second `document.write()` parameter.

To fetch the information about the browser version and name for your parameters, you call upon JavaScript to extract the corresponding properties from the navigator object. You extract a property by appending the property name to the object name (navigator in this case) and separating the two names with a period. If you're searching for some English to mentally assign to this scheme as you read it, start from the right side and call the right item a property "of" the left side: the `appVersion` property of the navigator object. This dot syntax looks a great deal like the `document.write()` task, but a property name does not have parentheses after it. In any case, the reference to the property in the script tells JavaScript to insert the value of that property in the spot where the call is made. For your first attempt at the script, JavaScript substitutes the internal information about the browser as part of the text string that gets written to the document.

Have Some Fun

If you encountered an error in your first attempt at loading this document into your browser, go back to the text editor and check the lines of the script section against Listing 3-1, looking carefully at each line in light of the explanations. There may be a single character out of place, a lowercase letter where an uppercase one belongs, or a quote or parenthesis missing. Make necessary repairs, switch to your browser, and click Reload.

To see how dynamic the script in script1.htm is, go back into the text editor and replace the word "browser" with "client software." Save, switch, and reload to see

how the script has changed the text in the document. Feel free to substitute other text for the quoted text in the `document.write()` statement. Or, add more text with additional `document.write()` statements. The parameters to `document.write()` are HTML text, so you can even write a "`<BR>`" to make a line break. Always be sure to save, switch, and reload to see the results of your handiwork.

✦ ✦ ✦