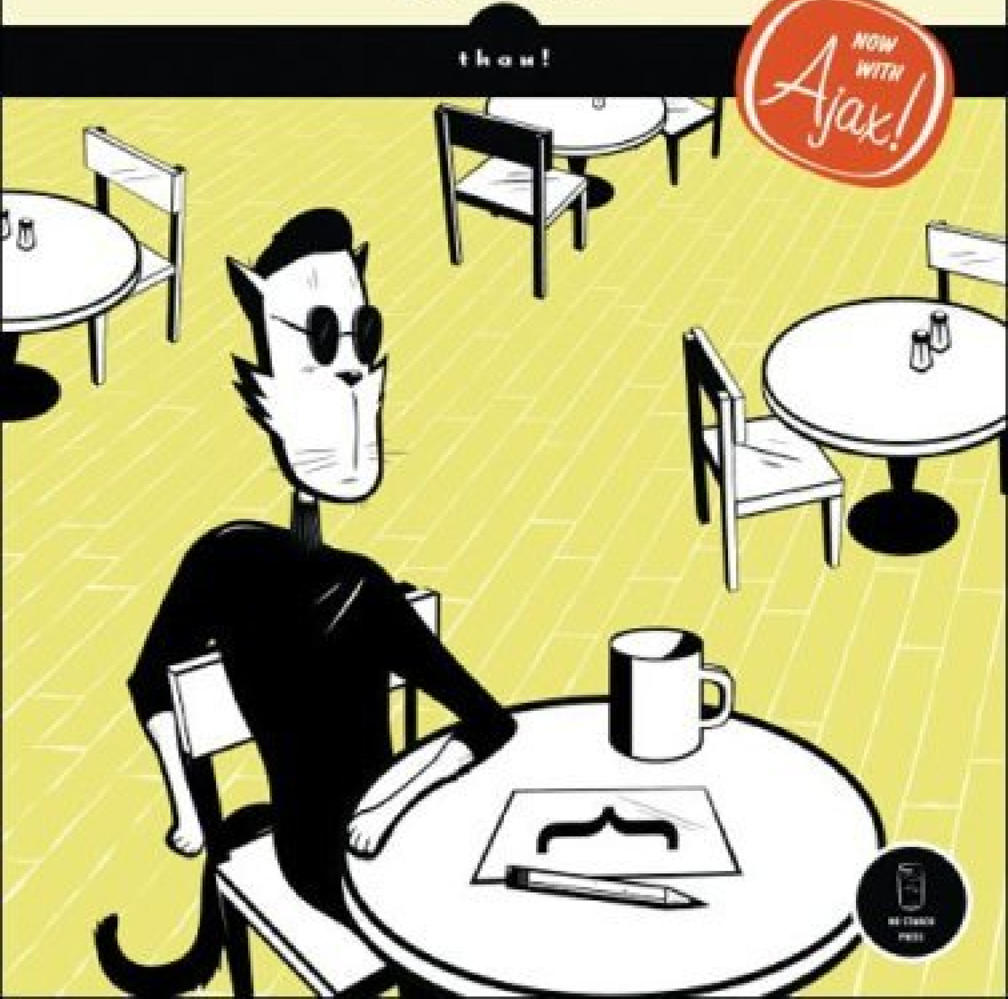
THE BOOK of

JAVASCRIPT

2ND EDITION

A PRACTICAL GUIDE TO INTERACTIVE WEB PAGES

thau!



THE BOOK *of*™ JAVASCRIPT
2ND EDITION

A PRACTICAL GUIDE TO *INTERACTIVE* WEB PAGES

by thau!

NO STARCH PRESS®

San Francisco

1

WELCOME TO JAVASCRIPT!

Welcome to *The Book of JavaScript.* JavaScript is one of the fastest and easiest ways to make your website truly dynamic—that is, interactive. If you want to spruce up tired-looking pages, you've got the right book.

This book will give you some ready-made JavaScripts you can implement immediately on your website, but, more importantly, it will take you step by step through sample scripts (both hypothetical and real-world examples) so that you understand how JavaScript works. With this understanding you can modify existing scripts to fit your specific needs as well as write scripts from scratch. Your knowledge of JavaScript will grow as you work through the book; each chapter introduces and explores in depth a new JavaScript topic by highlighting its application in real-life situations.

Is JavaScript for You?

If you want a quick, easy way to add interactivity to your website, if the thought of using complex programming languages intimidates you, or if you're interested in programming but simply don't know where to start, JavaScript is for you.

JavaScript, a programming language built into your web browser, is one of the best ways to add interactivity to your website because it's the only cross-browser language that works directly with web browsers. Other languages such as Java, Perl, PHP, and C don't have direct access to the images, forms, and windows that make up a web page.

JavaScript is also very easy to learn. You don't need any special hardware or software, you don't need access to a webserver, and you don't need a degree in computer science to get things working. All you need is a web browser and a text editor such as SimpleText or Notepad.

Finally, JavaScript is a complete programming language, so if you want to learn more about programming, it provides a great introduction. (If you don't give a hoot about programming, that's fine too. There are plenty of places—including this book and its companion website—where you can get prefab scripts to cut and paste right into your pages. But you'll get much more out of the book by using it as a tool for learning JavaScript programming.)

Is This Book for You?

This book assumes you don't have any programming background. Even if you have programmed before, you'll find enough that's new in JavaScript to keep you entertained. One of the best things about JavaScript is that you don't have to be a mega-expert to get it working on your web pages right away. You do need a working knowledge of HTML, however.

The Goals of This Book

The main goal of this book is to get you to the point of writing your own JavaScripts. An important tool in learning to write scripts is the ability to read other people's scripts. JavaScript is a sprawling language, and you can learn thousands of little tricks from other scripts. In fact, once you've finished this book, you'll find that viewing the source code of web pages that use JavaScript is the best way to increase your knowledge.

Each of the following chapters includes JavaScript techniques used in building professional sites. Along the way, I'll point out sites that use the technique described, and by viewing the source code of such sites you'll soon see there are many ways to script. Sometimes going through a site's code reveals interesting aspects of JavaScript that I don't cover in this book.

Beyond learning how to write your own JavaScript and read other people's scripts, I also want you to learn where to look for additional information on JavaScript. As I've noted, the best place to learn new techniques is to view the source code of web pages you find interesting. However, several websites also offer free JavaScripts. I'll be introducing some of these as we go along, but here are a few good examples to get you started:

- http://www.webmonkey.com/reference/javascript_code_library
- http://javascript.internet.com

- http://www.scriptsearch.com/JavaScript/Scripts
- http://www.javascriptsearch.com

Another good place to get information is a JavaScript reference book. *The Book of JavaScript* is primarily a tutorial for learning basic JavaScript and making your website interactive. It's not a complete guide to the language, which includes too many details for even a lengthy introduction to cover. If you're planning to become a true JavaScript master, I suggest picking up *JavaScript: The Definitive Guide* by David Flanagan (O'Reilly, 2006) after making your way through this book. The last 500 or so pages of Flanagan's book list every JavaScript command and the browsers in which it works.

What Can JavaScript Do?

JavaScript can add interactivity to your web pages in a number of ways. This book offers many examples of JavaScript's broad capabilities. The following are just two examples that illustrate what you can do with JavaScript.

The first example (Figure 1-1) is a flashing grid of colored squares (to get the full effect, browse to http://www.bookofjavascript.com/Chapter01/Fig01-01.html), created by a fellow named Taylor way back in 1996. Flashy, isn't it? In this example, a JavaScript function changes the color of a randomly chosen square in the grid every second or so.

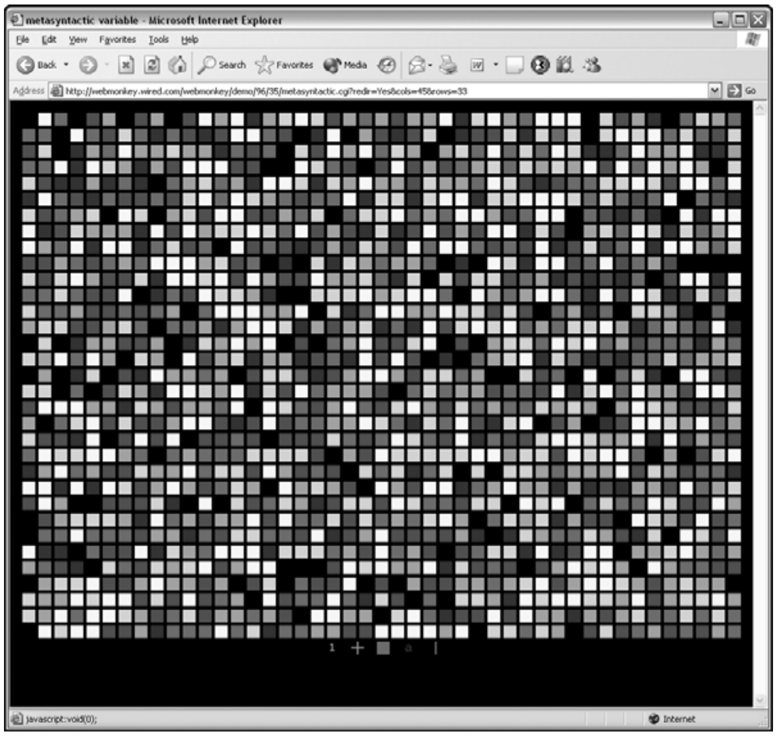
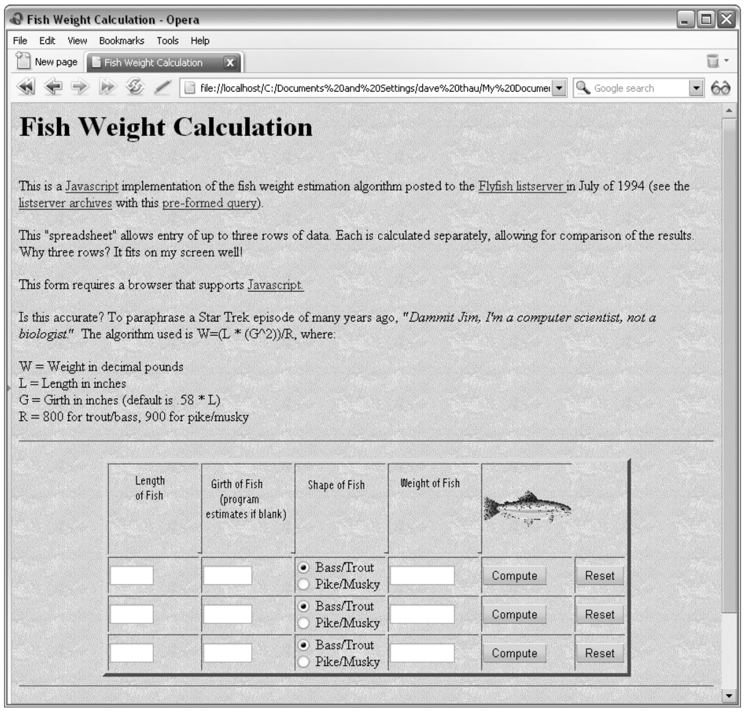

metasyntactic variable - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://webmonkey.wired.com/webmonkey/demo/96/35/metasyntactic.cgi?redir=Yes&cols=45&rows=33
Go
javascript:void(0);
Internet

Figure 1-1: A demonstration of JavaScript's artful abilities

Mousing over one of the five icons below the squares (number, plus sign, square, letter, and horizontal line) tells the page to use a new set of images on the grid. For example, mousing over the number icon tells the JavaScript to start replacing the squares with 1s and 0s. This page illustrates four important JavaScript features you'll learn about throughout the book:

- How to change images on a web page
- How to affect web pages over time
- How to add randomness to web pages
- How to dynamically change what's happening on a web page based on an action taken by someone viewing the page

Although Taylor's demo is beautiful, it's not the most practical application of JavaScript. Figure 1-2 (available at http://www.bookofjavascript.com/Chapter01/Fig01-02.html) shows you a much more practical use of JavaScript that calculates the weight of a fish based on its length. Enter the length and type of fish, and the JavaScript calculates the fish's weight. This fishy code demonstrates JavaScript's ability to read what a visitor has entered into a form, perform a mathematical calculation based on the input, and provide feedback by displaying the results in another part of the form. You may not find calculating a fish's weight a particularly useful application of JavaScript either, but you can use the same skills to calculate a monthly payment on a loan (Chapter 7), score a quiz (Chapter 10), or verify that a visitor has provided a valid email address (Chapter 11).


Fish Weight Calculation - Opera
File Edit View Bookmarks Tools Help
New page
Fish Weight Calculation
file://localhost/C:/Documents%20and%20Settings/dave%20thau/My%20Documei
Google search

Fish Weight Calculation

This is a Javascript implementation of the fish weight estimation algorithm posted to the Flyfish listserver in July of 1994 (see the listserver archives with this pre-formed query).

This "spreadsheet" allows entry of up to three rows of data. Each is calculated separately, allowing for comparison of the results. Why three rows? It fits on my screen well!

This form requires a browser that supports Javascript.

Is this accurate? To paraphrase a Star Trek episode of many years ago, *"Dammit Jim, I'm a computer scientist, not a biologist."* The algorithm used is W=(L * (G^2))/R, where:

W = Weight in decimal pounds
L = Length in inches
G = Girth in inches (default is .58 * L)
R = 800 for trout/bass, 900 for pike/musky

Length of Fish	Girth of Fish (program estimates if blank)	Shape of Fish	Weight of Fish		
		[✓ Bass/Trout] Pike/Musky		Compute	Reset
		[✓ Bass/Trout] Pike/Musky		Compute	Reset
		[✓ Bass/Trout] Pike/Musky		Compute	Reset

Figure 1-2: How much does my fish weigh?

These are just two examples of the many features JavaScript can add to your websites. Each chapter will cover at least one new application. If you want a preview of what you'll learn, read the first page or so of each chapter.

What Are the Alternatives to JavaScript?

Several other programming languages can add interactivity to web pages, but they all differ from JavaScript in important ways. The four main alternatives are CGI scripting, Java, VBScript, and Flash.

CGI Scripting

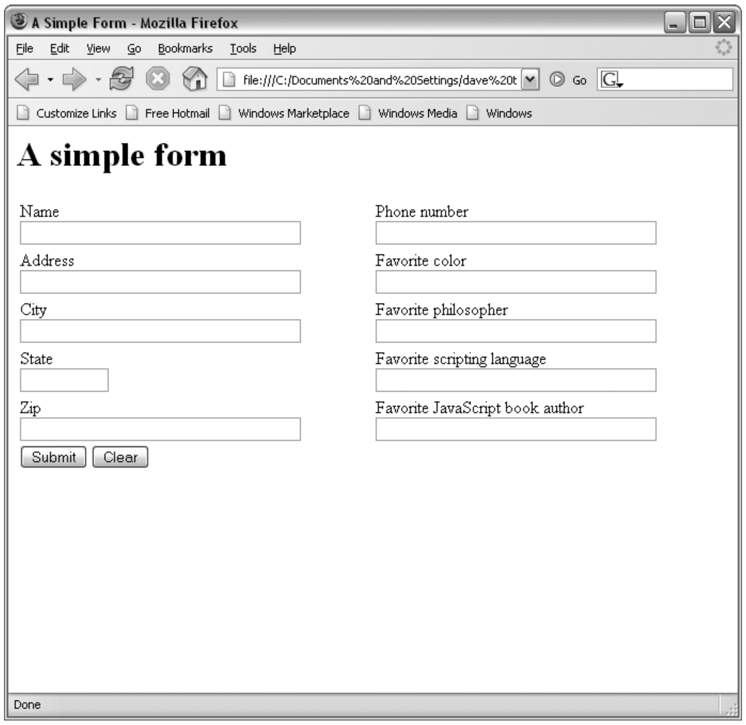
Before JavaScript, using CGI scripts was the only way to make web pages do more than hyperlink to other web pages containing fixed text. *CGI* stands for *Common Gateway Interface*. It's a protocol that allows a web browser running on your computer to communicate with programs running on webservers. It is most often used with HTML forms—pages where the user enters information and submits it for processing. For example, the user might see a web page containing places for entering the length and selecting the type of a fish, as well as a Compute button. When the user keys in the length, selects the type, and clicks the button, the information is sent to a CGI script on the server. The CGI script (which is probably written in a programming language like Perl, PHP, or C) receives the information, calculates the weight of the fish, and sends the answer, coded as an HTML page, back to the browser.

CGI scripts are very powerful, but because they reside on webservers, they have some drawbacks.

The Need for Back-and-Forth Communication

First, the connection between your web browser and the webserver limits the speed of your web page's interactivity. This may not sound like a big problem, but imagine the following scenario: You're filling out an order form with a dozen entry fields including name, address, and phone number (see Figure 1-3), but you forget to fill out the phone number and address fields. When you click the Submit button to send the information across the Internet to the webserver, the CGI script sees that you didn't fill out the form completely and sends a message back across the Internet requesting that you finish the job. This cycle could take quite a while over a slow connection. If you fill out the form incorrectly again, you have to wait through another cycle. People find this process tiresome, especially if they're customers who want their orders processed quickly.

With JavaScript, though, the programs you write run in the browser itself. This means that the browser can make sure you've filled out the form correctly before sending the form's contents to the webserver. JavaScript thus reduces the time your information spends traveling between the browser and the server.


A Simple Form - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
file:///C:/Documents%20and%20Settings/dave%20t
Go
Customize Links
Free Hotmail
Windows Marketplace
Windows Media
Windows
Submit
Clear
Done

A simple form

Name	Phone number
Address	Favorite color
City	Favorite philosopher
State	Favorite scripting language
Zip	Favorite JavaScript book author

Figure 1-3: A simple order form

Server Overload by Concurrent Access

Another drawback to CGI scripts is that a webserver running a CGI program can get bogged down if too many people call the script simultaneously (for example, if too many fishermen decided to run the weight calculator and click the Compute button at the same time). Serving up HTML pages is pretty easy for a webserver. However, some CGI scripts take a long time to run on a machine, and each time someone calls the script, the server has to start up another copy of it. As more and more people try to run the script, the server slows down progressively. If a thousand people are trying to run the script at once, the server might take so long to respond that either the user or the browser gives up, thinking the server is dead. This problem doesn't exist in JavaScript because its scripts run on each visitor's web browser—not on the webserver.

Security Restrictions

A third problem with CGI scripts is that not everyone has access to the parts of a webserver that can run CGI scripts. Since a CGI script can conceivably crash a webserver or exploit security flaws, system administrators generally guard these areas, only allowing fellow administrators access. If you have Internet access through an Internet service provider (ISP), you may not be allowed to write CGI scripts. If you are designing web pages for a company, you may not be given access to the CGI-enabled areas of the webserver.

JavaScript, on the other hand, goes right into the HTML of a web page. If you can write a web page, you can put JavaScript in the page without permission from recalcitrant administrators.

VBScript

The language most similar to JavaScript is Microsoft's proprietary language, VBScript (*VB* stands for *Visual Basic*). Like JavaScript, VBScript runs on your web browser and adds interactivity to web pages. However, VBScript works only on computers running Internet Explorer (IE) on Microsoft Windows, so unless you want to restrict your readership to people who use IE on Windows, you should go with JavaScript.

Java

Although JavaScript and Java have similar names, they aren't the same. Netscape, now a part of AOL, initially created JavaScript to provide interactivity for web pages, whereas Sun Microsystems wrote Java as a general programming language that works on all kinds of operating systems.

Flash

Flash is a tool from Macromedia developed to add animation and interactivity to websites. Almost all modern browsers can view Flash animations or can easily download the Flash plug-in. Flash animations look great, and a basic Flash animation requires no programming skills at all. To create Flash animations, however, you must purchase a Flash product from Macromedia.

While some people consider Flash and JavaScript to be competitors, that's not the case. In fact, you can call JavaScript programs from Flash, and you can manipulate Flash animations using JavaScript. Web page designers will often blend the two, using Flash for animations and JavaScript for interactivity that does not involve animations. Flash animations can also be made more interactive using a language called ActionScript, which is almost exactly like JavaScript.

JavaScript's Limitations

Yes, JavaScript does have limitations, but these limitations are natural and unavoidable by-products of its main purpose: to add interactivity to your web pages.

JavaScript Can't Talk to Servers

One of JavaScript's drawbacks is also its main strength: It works entirely within the web browser. As we've seen, this cuts down on the amount of time your browser spends communicating with a webserver. On the other hand, this also means that JavaScript can't communicate with other machines and therefore can't handle some server tasks you may need to do.

For example, JavaScript can't aggregate information collected from your users. If you want to write a survey that asks your visitors a couple of questions, stores their answers in a database, and sends a thank-you email when they finish, you'll have to use a program that runs on your webserver. As we'll see in Chapter 7, JavaScript can make the survey run more smoothly, but once a visitor has finished filling out the questions, JavaScript can't store the information on the server, because it can't contact the server. In order to store the survey information, you need to use a program that runs on a webserver. Sending email with JavaScript is also impossible, because to send email JavaScript would have to contact a mail server. Again, you need a server-side program for this job.

Although JavaScript can't directly control programs that run on webservers, it can ask webservers to run programs, and it can send information to those programs. We'll see examples of that in Chapters 7 and 14, and we'll get a taste for writing server-side programs in Chapters 15 and 16.

JavaScript Can't Create Graphics

Another of JavaScript's limitations is that it can't create its own graphics. Whereas more complicated languages can draw pictures, JavaScript can only manipulate existing pictures (that is, GIF or JPEG files). Luckily, because JavaScript can manipulate created images in so many ways, you shouldn't find this too limiting.

JavaScript Works Differently in Different Browsers

Perhaps the most annoying problem with JavaScript is that it works somewhat differently in different browsers. JavaScript was introduced in 1996 by Netscape in version 2 of Netscape Navigator. Since then, JavaScript has changed, and every browser implements a slightly different version of it—often adding browser-specific features. Luckily, starting in the late 1990s, the European Computer Manufacturers Association (ECMA) began publishing standards for JavaScript, which they call ECMAScript. About 99 percent of all browsers being used today comply with at least version 3 of the ECMA standard. These include Internet Explorer version 5.5 and later, Netscape version 6 and later, Mozilla, Firefox, all versions of Safari, and Opera version 5 and later. Because almost all browsers currently in use adhere to version 3 of the ECMA standard, I'll be using that as the standard version of JavaScript in the book. Where incompatibilities between browsers arise, I'll point them out.

Getting Started

We're about ready to begin. To write JavaScripts, you need a web browser and a text editor. Any text editor will do: Notepad or WordPad in Windows and SimpleText on a Macintosh are the simplest choices. Microsoft Word or Corel's WordPerfect will work as well. You can also use a text editor such as BBEdit or HomeSite, which are designed to work with HTML and JavaScript.

Some tools for building websites will actually write JavaScript for you—for example, Adobe's Dreamweaver and GoLive. These tools work fine when you want to write JavaScripts for common features such as image rollovers and you know you'll never want to change them. Unfortunately, the JavaScript often ends up much longer than necessary, and you may find it difficult to understand and change to suit your needs. Unless you want a JavaScript that works exactly like one provided by the package you've purchased, you're often best off writing scripts by hand. Of course, you can also use one of these tools to figure out how you want your page to behave and then go back and rewrite the script to suit your specific needs.

NOTE *Always save documents as text only, and end their names with* .html *or* .htm. *If you're using Microsoft Word or WordPerfect and you don't save your documents as text-only HTML or HTM files, both programs will save your documents in formats web browsers can't read. If you try to open a web page you've written and the browser shows a lot of weird characters you didn't put in your document, go back and make sure you've saved it as text only.*

Where JavaScript Goes on Your Web Pages

Now let's get down to some JavaScript basics. Figure 1-4 shows you the thinnest possible skeleton of an HTML page with JavaScript.

```
  <html>
  <head>
  <title>JavaScript Skeleton</title>
❶ <script type = "text/javascript">
  // JavaScript can go here!
  // But no HTML!
❷ </script>
  </head>
  <body>
  <script type = "text/javascript">
  // JavaScript can go here too!
  // But no HTML!
  </script>
  </body>
  </html>
```

Figure 1-4: An HTML page with JavaScript

In Figure 1-4, you can see the JavaScript between the `<script type = "text/javascript">` and `</script>` tags in ❶ and ❷.

Note that you can also start JavaScript with this `<script>` tag:

```
<script language = "JavaScript">
```

Although this will work in all browsers, it's better to stick to the official format:

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

If you feel like being extra clear, you can explicitly state which version of JavaScript your script will support. ECMAScript version 3 is also called JavaScript version 1.5. To tell a browser to run the JavaScript only if it understands JavaScript version 1.5, you can use this `<script>` tag:

```
<script type = "text/javascript" language = "JavaScript1.5">
```

Unfortunately, not all browsers check the language attribute for a version number, and the ones that don't check are, of course, the ones that don't understand JavaScript 1.5. So those browsers will happily try to run your JavaScript and will probably generate a JavaScript error. I'll talk more about ways to deal with older browsers in the next section and throughout the book. All in all, I recommend just sticking with `<script type = "text/javascript">`.

With one exception, which Chapter 4 will cover, all JavaScript goes between the open `<script>` and close `</script>` tags. Furthermore, you can't include any HTML between `<script>` and `</script>`. Between those tags, your browser assumes that everything it sees is JavaScript. If it sees HTML in there, or anything else it can't interpret as JavaScript, it gets confused and gives you an error message.

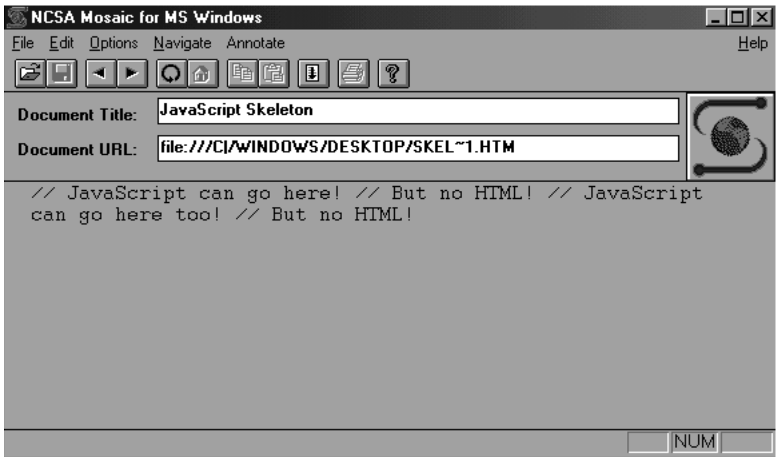
These JavaScript tags can go in either the head (between `<head>` and `</head>`) or the body (between `<body>` and `</body>`) of your HTML page. It doesn't matter too much where you put them, although you're generally best off putting as much JavaScript in the head as possible. That way you don't have to look for it all over your web pages.

One final thing worth mentioning here is that the lines that start with two slashes are JavaScript comments. The browser ignores any text that appears after two slashes. Documenting your work with comments is extremely important, because programming languages aren't easily understood. The script you're writing may make perfect sense while you're writing it, but a few days later, when you want to make a little modification, you might spend hours just figuring out what you wrote the first time. If you comment your code, you'll have a better chance to save yourself the hassle of remembering what you were thinking when you wrote that bizarre code at 2 AM in the midst of what seemed like an amazingly lucid caffeine haze.

Dealing with Older Browsers

There's a slight problem with the JavaScript skeleton in Figure 1-4 (besides the fact that it doesn't really have any JavaScript in it): Netscape didn't introduce the `<script>` tag until version 2.0 of Netscape Navigator, so any browser released before 1997 won't recognize the tag.

When a browser sees an HTML tag it doesn't understand, it just ignores that tag. That's generally a good thing. However, a browser that doesn't understand JavaScript will write your lines of JavaScript to the browser as text. Figure 1-5 shows how the JavaScript skeleton in Figure 1-4 would be displayed in an older browser.


NCSA Mosaic for MS Windows
File Edit Options Navigate Annotate Help
Document Title: JavaScript Skeleton
Document URL: file:///C|/WINDOWS/DESKTOP/SKEL~1.HTM
NUM

```
// JavaScript can go here! // But no HTML! // JavaScript
can go here too! // But no HTML!
```

Figure 1-5: What Figure 1-4 would display in an older browser

Although more than 99 percent of the browsers in use today understand JavaScript, most popular sites (Google, for example) still add lines like ❶ and ❷ of Figure 1-6, to hide their JavaScript from browsers that don't understand JavaScript.

```
<script type = "text/javascript">
❶ <!-- hide me from older browsers
// JavaScript goes here
❷ // show me -->
</script>
```

Figure 1-6: Hiding JavaScript from browsers that don't understand it

The important symbols are the `<!--` code in ❶ and the `// -->` comments in ❷. These weird lines work because in HTML, the `<!--` and `-->` are tags that mark the beginning and end of an entire block of comments. Older browsers that don't recognize the `<script>` tag see the comment markers and therefore don't try to display any of the JavaScript code between them. In JavaScript, on the other hand, `<!--` is the beginning of a comment that reaches only to the end of that one line, so browsers that understand JavaScript don't ignore the rest of the JavaScript between ❶ and ❷. The words in the tags (`hide me from older browsers` and `show me`) aren't important; they're just there to help you understand the code better. You can make those whatever you want or just leave them out entirely. It's the `<!--` and `// -->` tags that are important.

This trick may be a bit tough to understand at first. If so, don't worry—just remember to put the `<!--` tag on its own line right after `<script>` and the `// -->` tag on its own line right before `</script>`, and people with older browsers will thank you.

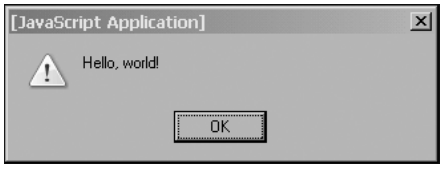
Your First JavaScript

It's time to run your first JavaScript program. I'll explain the code in Figure 1-7 in the next chapter, so for now, just type the code into your text editor, save it as my_first_program.html, and then run it in your browser. If you don't want to type it all in, run the example at http://www.bookofjavascript.com/Chapter01/Fig01-07.html.

```
  <html>
  <head>
  <title>JavaScript Skeleton</title>
  </head>
  <body>
  <script type = "text/javascript">
  <!-- hide me from older browsers
  // say Hello, world!
❶ alert("Hello, world!");
  // show me -->
  </script>
  </body>
  </html>
```

Figure 1-7: Your first JavaScript program

When a browser reads this file, the JavaScript in ❶ instructs the browser to put up a little window with the words *Hello, world!* in it. Figure 1-8 shows you what this looks like in a web browser. Traditionally, this is the first script you write in any programming language. It gets you warmed up for the fun to come.



Hello, world!

Figure 1-8: Window launched by the "Hello, world!" script

Summary

Congratulations—you're now on your way to becoming a bona fide JavaScripter! This chapter has given you all the basic tools you need and has shown you how to get a very basic JavaScript program running. If you followed everything here, you now know:

- Some of the great things JavaScript can do
- How JavaScript compares to CGI scripting, VBScript, Java, and Flash

- JavaScript's main limitations
- Where JavaScript goes on the page
- How to write JavaScript older browsers won't misunderstand

Assignment

Try typing Figure 1-7 into a text editor and running it in a web browser. You'll find the next chapter's assignments hard to do if you can't get Figure 1-7 to work.

If you're sure you've recreated Figure 1-7 exactly and it's not working, make sure you're saving the file as text only. You may also find it helpful to peruse Chapter 14, which discusses ways to fix broken code. Although you may not understand everything in that chapter, you may find some helpful tips.

If it's still not working, try running the version of Figure 1-7 at http://www.bookofjavascript.com/Chapter01/Fig01-07.html. If that doesn't work, you may be using a browser that doesn't support JavaScript, or your browser may be set to reject JavaScript. If you're sure you're using a browser that supports JavaScript (Netscape 2.0 and later versions, and Internet Explorer 3.0 and later), check your browser's options and make sure it's set to run JavaScript.

Once you're comfortable with the concepts covered in this chapter, you'll be ready to write some code!

2

USING VARIABLES AND BUILT-IN FUNCTIONS TO UPDATE YOUR WEB PAGES AUTOMATICALLY

With JavaScript you can update the content of your pages automatically—every day, every hour, or every second. In this chapter, I'll focus on a simple script that automatically changes the date on your web page.

Along the way you'll learn:

- How JavaScript uses variables to remember simple items such as names and numbers
- How JavaScript keeps track of more complicated items such as dates
- How to use JavaScript functions to write information to your web page

Before getting into the nuts and bolts of functions and variables, let's take a look at a couple of examples of web pages that automatically update themselves, starting with the European Space Agency (http://www.esa.int). As you can see in Figure 2-1, the ESA's home page shows you the current date. Rather than change the home page every day, the ESA uses JavaScript to change the date automatically.

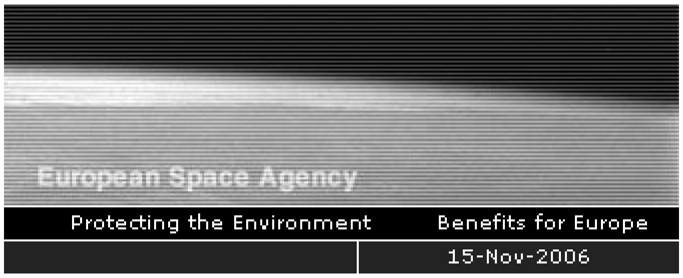


Figure 2-1: Using JavaScript to display the current date

An even more frequently updated page is the home page of the *Book of JavaScript* website (http://www.bookofjavascript.com), which updates the time as well as the date (see Figure 2-2). You don't have to sit in front of your computer, updating the dates and times on your websites. JavaScript can set you free! The ability to write HTML to web pages dynamically is one of JavaScript's most powerful features.

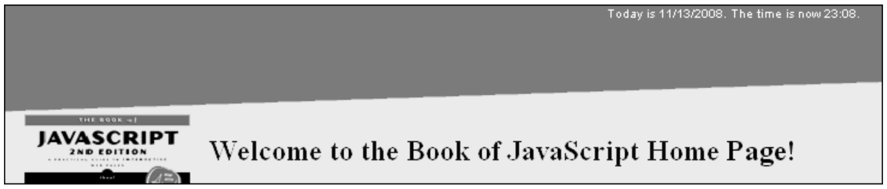

Today is 11/13/2008. The time is now 23:08.
THE BOOK of JAVASCRIPT 2ND EDITION
Welcome to the Book of JavaScript Home Page!

Figure 2-2: Dynamically updating the date and time

To understand how to update the date and time on the page, you'll first have to learn about variables, strings, and functions. Your homework assignment at the end of this chapter will be to figure out how to add seconds to the time.

Variables Store Information

Think back to those glorious days of algebra class when you learned about variables and equations. For example, if $x = 2$, $y = 3$, and $z = x + y$, then $z = 5$. In algebra, variables like x, y, and z store or hold the place of numbers. In JavaScript and other programming languages, variables also store other kinds of information.

Syntax of Variables

The *syntax* of variables (the set of rules for defining and using variables) is slightly different in JavaScript from what it was in your algebra class. Figure 2-3 illustrates the syntax of variables in JavaScript with a silly script that figures out how many seconds there are in a day.

NOTE *Figure 2-3 does not write the results of the JavaScript to the web page—I'll explain how to do that in Figure 2-4.*

```
<html>
<head>
<title>Seconds in a Day</title>

<script type = "text/javascript">
<!-- hide me from older browsers

❶ var seconds_per_minute = 60;
  var minutes_per_hour = 60;
  var hours_per_day = 24;

❷ var seconds_per_day = seconds_per_minute * minutes_per_hour * hours_per_day;

// show me -->
❸ </script>
</head>
<body>

<h1>Know how many seconds are in a day?</h1>
<h2>I do!</h2>

</body>
</html>
```

Figure 2-3: Defining and using variables

There's a lot going on here, so let's take it line by line. Line ❶ is a *statement* (a statement in JavaScript is like a sentence in English), and it says to JavaScript, "Create a variable called `seconds_per_minute` and set its value to 60." Notice that ❶ ends with a semicolon. Semicolons in JavaScript are like periods in English: They mark the end of a statement (for example, one that defines a variable, as above). As you see more and more statements, you'll get the hang of where to place semicolons.

The first word, `var`, introduces a variable for the first time—you don't need to use it after the first instance, no matter how many times you employ the variable in the script.

NOTE *Many people don't use* `var` *in their code. Although most browsers let you get away without it, it's always a good idea to put* `var` *in front of a variable the first time you use it. (You'll see why when I talk about writing your own functions in Chapter 6.)*

Naming Variables

Notice that the variable name in ❶ is pretty long—unlike algebraic variables, it's not just a single letter like *x*, *y*, or *z*. When using variables in JavaScript (or any programming language), you should give them names that indicate what piece of information they hold. The variable in ❶ stores the number of seconds in a minute, so I've called it `seconds_per_minute`.

If you name your variables descriptively, your code will be easier to understand while you're writing it, and much easier to understand when you return to it later for revision or enhancement. Also, no matter which programming language you use, you'll spend about 50 percent of your coding time finding and getting rid of your mistakes. This is called *debugging*—and it's a lot easier to debug code when the variables have descriptive names. You'll learn more about debugging in Chapter 14.

There are four rules for naming variables in JavaScript:

1. The initial character must be a letter, an underscore, or a dollar sign, but subsequent characters may be numbers as well.
2. No spaces are allowed.
3. Variables are case sensitive, so `my_cat` is different from `My_Cat`, which in turn is different from `mY_cAt`. As far as the computer is concerned, each of these would represent a different variable—even if that's not what the programmer intended. (You'll see an example of this in the section "alert()" on page 22.) To avoid any potential problems with capitalization, I use lowercase for all my variables, with underscores (`_`) where there would be spaces in ordinary English.
4. You can't use reserved words. *Reserved words* are terms used by the JavaScript language itself. For instance, you've seen that the first time you use a variable, you should precede it with the word `var`. Because JavaScript uses the word `var` to introduce variables, you can't use `var` as a variable name. Different browsers have different reserved words, so the best thing to do is avoid naming variables with words that seem like terms JavaScript might use. Most reserved words are fairly short, so using longer, descriptive variable names keeps you fairly safe. I often call my variables things like `the_cat`, or `the_date` because there are no reserved words that start with the word *the*. If you have a JavaScript that you're *certain* is correct, but it isn't working for some reason, it might be because you've used a reserved word.

Arithmetic with Variables

Line ❷ in Figure 2-3 introduces a new variable called `seconds_per_day` and sets it equal to the product of the other three variables using an asterisk (`*`), which means multiplication. A plus sign (`+`) for addition, a minus sign (`-`) for subtraction, and a slash (`/`) for division represent the other major arithmetic functions.

When the browser finishes its calculations in our example, it reaches the end of the JavaScript in the head (❸) and goes down to the body of the HTML. There it sees two lines of HTML announcing that the page knows how many seconds there are in a day.

```
<h1>Know how many seconds are in a day?</h1>
<h2>I do!</h2>
```

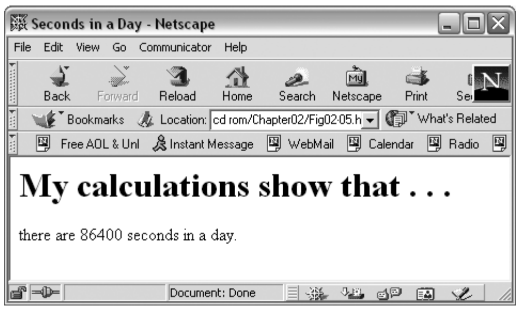
So now you have a page that knows how many seconds there are in a day. Big deal, right? Wouldn't it be better if you could tell your visitors what the answer is? Well, you can, and it's not very hard.

Write Here Right Now: Displaying Results

JavaScript uses the `write()` function to write text to a web page. Figure 2-4 shows how to use `write()` to let your visitors know how many seconds there are in a day. (The new code is in bold.) Figure 2-5 shows the page this code displays.

```
<html>
<head>
<title>Seconds in a Day</title>

<script type = "text/javascript">
<!-- hide me from older browsers

var seconds_per_minute = 60;
var minutes_per_hour = 60;
var hours_per_day = 24;

var seconds_per_day = seconds_per_minute * minutes_per_hour * hours_per_day;

// show me -->
</script>
</head>
<body>

<h1>My calculations show that . . .</h1>

<script type = "text/javascript">
<!-- hide me from older browsers

❶ window.document.write("there are ");
window.document.write(seconds_per_day);
window.document.write(" seconds in a day.");

// show me -->
</script>

</body>
</html>
```

Figure 2-4: Using `write()` to write to a web page


Seconds in a Day - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print
Bookmarks Location: cd rom/Chapter02/Fig02-05.h What's Related
Free AOL & Unl Instant Message WebMail Calendar Radio
Document: Done

My calculations show that . . .

there are 86400 seconds in a day.

Figure 2-5: JavaScript's calculations

Line-by-Line Analysis of Figure 2-4

Line ❶ in Figure 2-4 writes the words *there are* to the web page (only the words between the quotes appear on the page). Don't worry about all the periods and what `window` and `document` really mean right now (I'll cover these topics in depth in Chapter 4, when we talk about image swaps). For now, just remember that if you want to write something to a web page, use `window.document.write("whatever");`, placing the text you want written to the page between the quotes. If you don't use quotes around your text, as in

```
window.document.write(seconds_per_day);
```

then JavaScript interprets the text between the parentheses as a variable and writes whatever is stored in the variable (in this case, `seconds_per_day`) to the web page (see Figure 2-6). If you accidentally ask JavaScript to write out a variable you haven't defined, you'll get a JavaScript error.

Be careful not to put quotes around variable names if you want JavaScript to know you're talking about a variable. If you add quotes around the `seconds_per_day` variable, like this:

```
window.document.write("seconds_per_day");
```

then JavaScript will write *seconds_per_day* to the web page. The way JavaScript knows the difference between variables and regular text is that regular text has quotes around it and a variable doesn't.

Strings

Any series of characters between quotes is called a *string*. (You'll be seeing lots of strings throughout this book.) Strings are a basic type of information, like numbers—and like numbers, you can assign them to variables.

To assign a string to a variable, you'd write something like this:

```
var my_name = "thau!";
```

The word `thau!` is the string assigned to the variable `my_name`.

You can stick strings together with a plus sign (+), as shown in the bolded section of Figure 2-6. This code demonstrates how to write output to your page using strings.

```
<html>
<head>
<title>Seconds in a Day</title>
<script type = "text/javascript">
<!-- hide me from older browsers

var seconds_per_minute = 60;
var minutes_per_hour = 60;
var hours_per_day = 24;

var seconds_per_day = seconds_per_minute * minutes_per_hour * hours_per_day;
```

```
// show me -->
</script>
</head>
<body>

<h1>My calculations show that . . .</h1>

<script type = "text/javascript">
<!-- hide me from older browsers

❶ var first_part = "there are ";
❷ var last_part = " seconds in a day.";
❸ var whole_thing = first_part + seconds_per_day + last_part;

window.document.write(whole_thing);

// show me -->
</script>

</body>
</html>
```

Figure 2-6: Putting strings together

Line-by-Line Analysis of Figure 2-6

Line ❶ in Figure 2-6,

```
var first_part = "there are ";
```

assigns the string `"there are"` to the variable `first_part`. Line ❷,

```
var last_part = " seconds in a day.";
```

sets the variable `last_part` to the string `"seconds in a day."` Line ❸ glues together the values stored in `first_part`, `seconds_per_day`, and `last_part`. The end result is that the variable `whole_thing` includes the whole string you want to print to the page, *there are 86400 seconds in a day*. The `window.document.write()` line then writes `whole_thing` to the web page.

NOTE *The methods shown in Figures 2-4 and 2-6 are equally acceptable ways of writing* there are 86400 seconds in a day. *However, there are times when storing strings in variables and then assembling them with the plus sign (+) is clearly the best way to go. We'll see a case of this when we finally get to putting the date on a page.*

More About Functions

Whereas variables store information, *functions* process that information.

All functions take the form *`functionName`*`()`. Sometimes there's something in the parentheses and sometimes there isn't. You've already seen one of JavaScript's many built-in functions, `window.document.write()`, which writes whatever lies between the parentheses to the web page. Before diving into the date functions that you'll need to write the date to your web page, I'll talk about two interesting functions, just so you get the hang of how functions work.

alert()

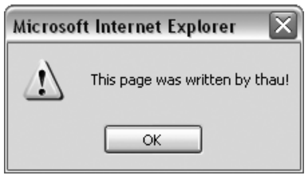
One handy function is `alert()`, which puts a string into a little announcement box (also called an *alert box*). Figure 2-7 demonstrates how to call an `alert()`, and Figure 2-8 shows what the alert box looks like.

```
  <html>
  <head>
  <title>An Alert Box</title>

  <script type = "text/javascript">
  <!-- hide me from older browsers
❶ alert("This page was written by thau!");
  // show me -->
  </script>

  <body>
❷ <h1>To code, perchance to function</h1>
  </body>
  </html>
```

Figure 2-7: Creating an alert box

The first thing visitors see when they come to the page Figure 2-7 creates is an alert box announcing that I wrote the page (Figure 2-8). The alert box appears because of ❶, which tells JavaScript to execute its `alert()` function.

While the alert box is on the screen, the browser stops doing any work. Clicking OK in the alert box makes it go away and allows the browser to finish drawing the web page. In this case, that means writing the words *To code, perchance to function* to the page (❷).



This page was written by thau!

Figure 2-8: The alert box

The `alert()` function is useful for troubleshooting when your JavaScript isn't working correctly. Let's say you've typed in Figure 2-6, but when you run the code, you see that you must have made a typo—it says there are 0 seconds in a day instead of 86400. You can use `alert()` to find out how the different variables are set before multiplication occurs. The script in Figure 2-9 contains an error that causes the script to say there are "undefined" seconds in a year; and to track down the error, I've added `alert()` function statements that tell you why this problem is occurring.

```
  <html>
  <head>
  <title>Seconds in a Day</title>

  <script type = "text/javascript">
  <!-- hide me from older browsers

  var seconds_per_minute = 60;
  var minutes_per_hour = 60;
❶ var Hours_per_day = 24;
❷ alert("seconds per minute is: " + seconds_per_minute);
❸ alert("minutes per hour is: " + minutes_per_hour);
❹ alert("hours per day is: " + hours_per_day);
❺ var seconds_per_day = seconds_per_minute * minutes_per_hour * hours_per_day;



  // show me -->
  </script>
  </head>
  <body>

  <h1>My calculations show that . . .</h1>

  <script type = "text/javascript">
  <!-- hide me from older browsers

  var first_part = "there are ";
  var last_part = " seconds in a day.";
  var whole_thing = first_part + seconds_per_day + last_part;

  window.document.write(whole_thing);

  // show me -->
  </script>

  </body>
  </html>
```

Figure 2-9: Using `alert()` to find out what's wrong

Line-by-Line Analysis of Figure 2-9

The problem with this script is in ❶. Notice the accidental capitalization of the first letter in `Hours_per_day`. This is what causes the script to misbehave. Line ❺ multiplies the other numbers by the variable `hours_per_day`, but `hours_per_day` was not set—remember, JavaScript considers it a different variable from `Hours_per_day`—so JavaScript thinks its value is either 0 or undefined, depending on your browser. Multiplying anything by 0 results in 0, so the script calculates that there are 0 seconds in a day. The same holds true for browsers that think `hours_per_day` is undefined. Multiplying anything by something undefined results in the answer being undefined, so the browser will report that there are undefined seconds in a day.

This script is short, making it easy to see the mistake. However, in longer scripts it's sometimes hard to figure out what's wrong. I've added ❷, ❸, and ❹ in this example to help diagnose the problem. Each of these statements puts a variable into an alert box. The alert in ❷ will say `seconds_per_minute is: 60`. The alert in ❹ will say `hours_per_day is: 0`, or, depending on your browser, the alert won't appear at all. Either way, you'll know there's a problem with the `hours_per_day` variable. If you can't figure out the mistake by reading the script, you'll find this type of information very valuable. Alerts are very useful debugging tools.

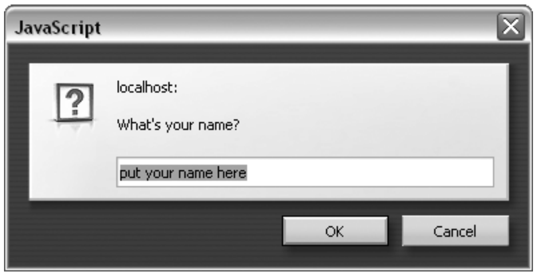
prompt()

Another helpful built-in function is `prompt()`, which asks your visitor for some information and then sets a variable equal to whatever your visitor types. Figure 2-10 shows how you might use `prompt()` to write a form letter.

```
  <html>
  <head>
  <title>A Form Letter</title>
  <script type = "text/javascript">
  <!-- hide me from older browsers

❶ var the_name = prompt("What's your name?", "put your name here");

  // show me -->
  </script>
  </head>
  <body>
❷ <h1>Dear

  <script type = "text/javascript">
  <!-- hide me from older browsers

  document.write(the_name);

  // show me -->
  </script>

  ,</h1>

  Thank you for coming to my web page.

  </body>
  </html>
```

Figure 2-10: Using `prompt()` to write a form letter

Notice that `prompt()` in ❶ has two strings inside the parentheses: `"What's your name?"` and `"put your name here"`. If you run the code in Figure 2-10, you'll see a prompt box that resembles Figure 2-11. (I've used the Opera browser in this illustration; prompt boxes will look somewhat different in IE and other browsers.) If you type `Rumpelstiltskin` and click OK, the page responds with *Dear Rumpelstiltskin, Thank you for coming to my web page.*



localhost:

What's your name?

put your name here

Figure 2-11: Starting a form letter with a prompt box

The text above the box where your visitors will type their name (`"What's your name?"`) is the first string in the prompt function; the text inside the box (`"put your name here"`) is the second string. If you don't want anything inside the box, put two quotes (`""`) right next to each other in place of the second string to keep that space blank:

```
var the_name = prompt("What's your name?", "");
```

If you look at the JavaScript in the body (starting in ❷), you'll see how to use the variable `the_name`. First write the beginning of the heading to the page using normal HTML. Then launch into JavaScript and use `document.write(the_name)` to write whatever name the visitor typed into the prompt box for your page. If your visitor typed `yertle the turtle` into that box, *yertle the turtle* gets written to the page. Once the item in `the_name` is written, you close the JavaScript tag, write a comma and the rest of the heading using regular old HTML, and then continue with the form letter. Nifty, eh?

The `prompt()` function is handy because it enables your visitor to supply the variable information. In this case, after the user types a name into the prompt box in Figure 2-10 (thereby setting the variable `the_name`), your script can use the supplied information by calling that variable.

Parameters

The words inside the parentheses of functions are called *parameters*. The `document.write()` function requires one parameter: a string to write to your web page. The `prompt()` function takes two parameters: a string to write above the box and a string to write inside the box.

Parameters are the only aspect of a function you can control; they are your means of providing the function with the information it needs to do its job. With a `prompt()` function, for example, you can't change the color of the box, how many buttons it has, or anything else; in using a predefined prompt box, you've decided that you don't need to customize the box's appearance. You can only change the parameters it specifically provides—

namely, the text and heading of the prompt you want to display. You'll learn more about controlling what functions do when you write your own functions in Chapter 6.

Writing the Date to Your Web Page

Now that you know about variables and functions, you can print the date to your web page. To do so, you must first ask JavaScript to check the local time on your visitor's computer clock:

```
var now = new Date();
```

The first part of this line, `var now =`, should look familiar. It sets the variable `now` to some value. The second part, `new Date()`, is new; it creates an object.

Objects store data that require multiple pieces of information, such as a particular moment in time. For example, in JavaScript you need an object to describe *2:30 PM on Saturday, January 7, 2006, in San Francisco*. That's because it requires many different bits of information: the time, day, month, date, and year, as well as some representation (in relation to Greenwich Mean Time) of the user's local time. As you can imagine, working with an object is a bit more complicated than working with just a number or a string.

Because dates are so rich in information, JavaScript has a built-in `Date` object to contain those details. When you want the user's current date and time, you use `new Date()` to tell JavaScript to create a `Date` object with all the correct information.

NOTE *You must capitalize the letter* `D` *in* `Date` *to tell JavaScript you want to use the built-in* `Date` *object. If you don't capitalize it, JavaScript won't know what kind of object you're trying to create, and you'll get an error message.*

Built-in Date Functions

Now that JavaScript has created your `Date` object, let's extract information from it using JavaScript's built-in date functions. To extract the current year, use the `Date` object's `getYear()` function:

```
var now = new Date();
var the_year = now.getYear();
```

Date and Time Methods

In the code above, the variable `now` is a `Date` object, and the function `getYear()` is a method of the `Date` object. *Methods* are simply functions that are built in to objects. For example, the `getYear()` function is built in to the `Date` object and gets the object's year. Because the function is part of the `Date` object, it is called a method. To use the `getYear()` method to get the year of the date stored in the variable `now`, you would write:

```
now.getYear()
```

Table 2-1 lists commonly used date methods. (You can find a complete list of date methods in Appendix C.)

Table 2-1: Commonly Used Date and Time Methods

Name	Description
`getDate()`	The day of the month as an integer from 1 to 31
`getDay()`	The day of the week as an integer where 0 is Sunday and 1 is Monday
`getHours()`	The hour as an integer between 0 and 23
`getMinutes()`	The minutes as an integer between 0 and 59
`getMonth()`	The month as an integer between 0 and 11 where 0 is January and 11 is December
`getSeconds()`	The seconds as an integer between 0 and 59
`getTime()`	The current time in milliseconds where 0 is January 1, 1970, 00:00:00
`getYear()`	The year, but this format differs from browser to browser

NOTE *Notice that `getMonth()` returns a number between 0 and 11; if you want to show the month to your site's visitors, to be user-friendly you should add 1 to the month after using `getMonth()`, as shown in ❷ in Figure 2-12.*

Internet Explorer and various versions of Netscape deal with years in different and strange ways:

- Some versions of Netscape, such as Netscape 4.0 for the Mac, always return the current year minus 1900. So if it's the year 2010, `getYear()` returns 110.
- Other versions of Netscape return the full four-digit year except when the year is in the twentieth century, in which case they return just the last two digits.
- Netscape 2.0 can't deal with dates before 1970 at all. Any date before January 1, 1970 is stored as December 31, 1969.
- In Internet Explorer, `getYear()` returns the full four-digit year if the year is after 1999 or before 1900. If the year is between 1900 and 1999, it returns the last two digits.

You'd figure a language created in 1995 wouldn't have the Y2K problem, but the ways of software developers are strange. Later in this chapter I'll show you how to fix this bug.

Code for Writing the Date and Time

Now let's put this all together. To get the day, month, and year, we use the `getDate()`, `getMonth()`, and `getYear()` methods. To get the hour and the minutes, we use `getHours()` and `getMinutes()`.

Figure 2-12 shows you the complete code for writing the date and time (without seconds) to a web page, as seen on the *Book of JavaScript* home page.

```
<html>
<head><title>The Book of JavaScript</title>
<script type = "text/javascript">
<!-- hide me from older browsers
// get the Date object
//
❶ var date = new Date();

// get the information out of the Date object
//
var month = date.getMonth();
var day = date.getDate();
var year = date.getYear();
var hour = date.getHours();
var minutes = date.getMinutes();
❷ month = month + 1;  // because January is month 0
// fix the Y2K bug
//
❸ year = fixY2K(year);

// fix the minutes by adding a 0 in front if it's less than 10
//
❹ minutes = fixTime(minutes);

// create the date string
//
❺ var date_string = month + "/" + day + "/" + year;
❻ var time_string = hour + ":" + minutes;
❼ var date_time_string = "Today is " + date_string + ".  The time is now " +
    time_string + ".";

// This is the Y2K fixer function--don't worry about how this works,
// but if you want it in your scripts, you can cut and paste it.
//
function fixY2K(number) {
  if (number < 1000) {
    number = number + 1900;
  }
  return number;
}

// This is the time fixer function--don't worry about how this works either.
function fixTime(number) {
  if (number < 10) {
    number = "0" + number;
  }
  return number;
}

// show me -->
</script>
</head>
<body>
```

```
❽ <h1>Welcome to the Book of JavaScript Home Page!</h1>

  <script type = "text/javascript">
  <!-- hide me from older browsers
❾ document.write(date_time_string);
  // show me -->
  </script>
  </body>
  </html>
```

Figure 2-12: Writing the current date and time to a web page

Line-by-Line Analysis of Figure 2-12

Here are a few interesting things in this example.

Getting the Date and Time

The lines from ❶ up until ❷ get the current date and time from the visitor's computer clock and then use the appropriate date methods to extract the day, month, year, hours and minutes. Although I'm using a variable name `date` in ❶ to store the date, I could have used any variable name there: `the_date`, `this_moment`, `the_present`, or any valid variable name. Don't be fooled into thinking that a variable needs to have the same name as the corresponding JavaScript object; in this case, `date` just seems like a good name.

Making Minor Adjustments

Before building the strings we will write to the website, we need to make some little adjustments to the date information just collected. Here's how it works:

- Line ❷ adds 1 to the month because `getMonth()` thinks January is month 0.
- Line ❸ fixes the Y2K problem discussed earlier in the chapter, in which the `getYear()` method returns the wrong thing on some older browsers. If you feed `fixY2K()` the year returned by `date.getYear()`, it will return the correct year. The `fixY2K()` function is not a built-in JavaScript function. I had to write it myself. Don't worry about how the function works right now.
- Line ❹ fixes a minor formatting issue, using another function that's not built-in. If the script is called at 6 past the hour, `date.getMinutes()` returns 6. If you don't do something special with that 6, your time will look like 11:6 instead of 11:06. So `fixTime()` sticks a zero in front of a number if that number is less than 10. You can use `fixTime()` to fix the seconds too, for your homework assignment.

Getting the String Right

Now that we've made a few minor adjustments, it's time to build the strings. Line ❺ builds the string for the date. Here's the *wrong* way to do it:

```
var date_string = "month / day / year";
```

If you wrote your code this way, you'd get a line that says *Today is month / day / year*. Why? Remember that JavaScript doesn't look up variables if they're inside quotes. So place the variables outside the quote marks and glue everything together using plus signs (+):

```
var date_string = month + "/" + day + "/" + year;
```

This may look a little funny at first, but it's done so frequently that you'll soon grow used to it. Line ❻ creates the string to represent the time. It is very similar to ❺. Line ❼ puts ❺ and ❻ together to create the string that will be written to the website. Lines ❺ through ❼ could all have been written as one long line:

```
var date_time_string = "Today is " + month + "/" + day + "/" + year +
    ". The time is now " + hour + ":" + minutes + ".";
```

However, using three lines makes the code easier for people to read and understand. It's always best to write your code as if other people are going to read it.

What Are Those Other Functions?

The JavaScript between ❼ and ❽ defines the `fixY2K()` and `fixTime()` functions. Again, don't worry about these lines for now. We'll cover how to write your own functions in glorious detail in Chapter 6.

JavaScript and HTML

Make sure to place your JavaScript and HTML in the proper order. In Figure 2-12, the welcoming HTML in ❽ precedes the JavaScript that actually writes the date and time in ❾, since the browser first writes that text and then executes the JavaScript. With JavaScript, as with HTML, browsers read from the top of the page down. I've put `document.write()` in the body so that the actual date information will come after the welcome text. I've put the rest of the JavaScript at the head of the page to keep the body HTML cleaner.

Why document.write()?

Notice that the code in Figure 2-11 uses `document.write()` instead of `window.document.write()`. In general, it's fine to drop the word `window` and the first dot before the word `document`. In future chapters I'll tell you when the word `window` must be added.

How the European Space Agency Writes the Date to Its Page

The JavaScript used by the European Space Agency is very much like the code I used for the *Book of JavaScript* web page. One big difference between the two is that the ESA prints out the month using abbreviations like *Jan* and *Feb* for *January* and *February*. They do this using arrays, a topic discussed in Chapter 8, so in Figure 2-13 I've modified their code a bit to focus on topics covered so far.

```
<script type = "text/javascript">
var now = new Date();
var yyyy = now.getFullYear();
var mm = now.getMonth() + 1;
❶ if (10 > mm) mm = '0' + mm;
var dd = now.getDate();
❷ if (10 > dd) dd = '0' + dd;
document.write(dd + '-' + mm + '-' + yyyy);
</script>
```

Figure 2-13: How the European Space Agency writes the date to its page

Everything here should look very familiar to you, except for ❶ and ❷, which will make more sense after you've read Chapter 3. If anything else in the ESA script seems unclear to you, try doing the homework assignment. In fact, do the homework assignment even if it all seems extremely clear. The only way to really learn JavaScript is to do it. Go ahead, give that homework a shot! And enjoy!

Summary

This chapter was chock-full of JavaScript goodness. Here's a review of the most important points for you to understand:

- How to declare and use variables (use `var` the first time and use valid and descriptive variable names)
- How to write to web pages with `document.write()`
- How to get the current date from JavaScript with the `Date` object and its various methods

If you got all that, you're well on your way to becoming a JavaScript superstar. Try the following assignment to test your JavaScript skills.

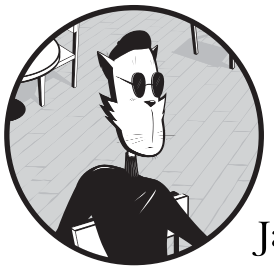
Assignment

Change the script in Figure 2-12 so that it writes out the seconds as well as the hour and minutes.

If you're feeling like getting ahead of the game, you can try, for a big chunk of extra credit, to change the time from a 24-hour clock to a 12-hour clock. The `getHours()` method returns the hour as a number between 0 and 23. See if you can figure out how to adjust that time to be between 1 and 12. You'll have to use some tricks I haven't covered in this chapter. If you can't figure this out now, you'll be able to do it by the end of the next chapter.

3

GIVING THE BROWSERS WHAT THEY WANT



Much to the dismay of web developers everywhere, different browsers implement JavaScript and HTML in slightly different ways. Wouldn't it be great if you could serve each browser exactly the content it could understand?

Fortunately, you can use JavaScript to determine which browser a visitor is using. You can then use that information to deliver content suitable for that specific browser, either by redirecting the visitor to a page containing content especially tailored for that browser or by writing your JavaScripts so that the same page does different things depending on the browser looking at it.

This chapter covers the three topics you need to understand to deliver browser-specific pages using redirects:

- How to determine which browser your visitor is using
- How to redirect the visitor to other pages automatically
- How to send the visitor to the page you want, depending on which browser he or she is using

As in Chapter 2, while learning how to handle an important web authoring task, you'll also be introduced to fundamental elements of the JavaScript language—in this case, `if-then` statements and related methods for implementing logical decision making in your scripts.

Let's first talk about determining which browser a visitor is using.

A Real-World Example of Browser Detection

Before we get into the details of how browser detection works, let's look at a real-world example.

Netscape, the company that brought you the Netscape Navigator browser, has a complicated home page with lots of interesting features. They've taken great pains to make their home page look good to most browsers, including early versions of their own browser. If you compare the Netscape home page seen with Netscape Navigator 4 (Figure 3-1) to the page seen using Navigator 8 (Figure 3-2), you'll notice some subtle differences. Among other things, the news blurb at the bottom of Figure 3-2 has a little navigational element in the lower-right corner. Clicking the numbers in that corner cycles you through different news blurbs. Figure 3-1 does not have these numbers, probably because there isn't a good way to provide this fancy functionality in the old Netscape Navigator.

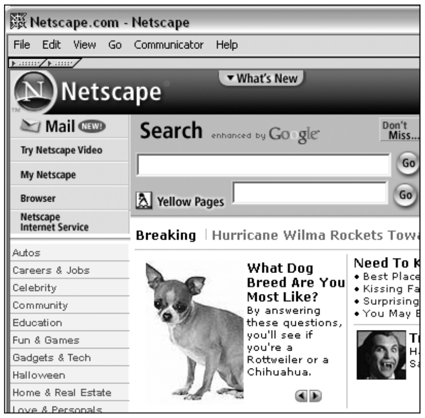


Figure 3-1: Netscape Navigator 4 view of Netscape home page

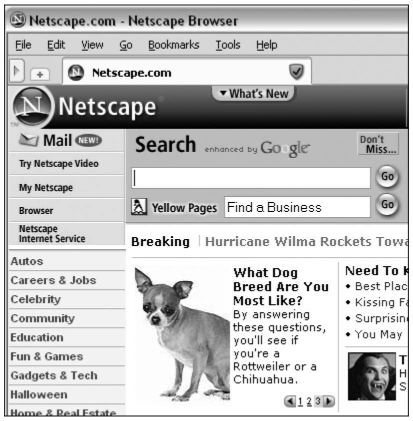


Figure 3-2: Netscape Navigator 8 view of Netscape home page

How does Netscape show the numbers to only those browsers that can provide this feature? There are two steps. First you have to determine which browser your visitor is using. Once you know the browser, you know what JavaScript and HTML features it supports. Then you have to figure out how to control what the person will see based on the known capabilities of the browser.

Browser Detection Methods

A browser is identified by its name (Netscape, Firefox, Internet Explorer, and so on) combined with its version number. Your JavaScript needs to determine both of these items. There are two ways to approach this task: a quick but rough method and a slightly less quick but more accurate method.

Quick-but-Rough Browser Detection

In general, the line

```
var browser_name = navigator.appName;
```

determines who made the browser. If the user is using a Netscape browser, the variable `browser_name` will be set to the string `"Netscape"`. If it's a Microsoft Internet Explorer browser, `browser_name` will be set to `"Microsoft Internet Explorer"`. Every JavaScript-enabled browser must have the variable `navigator.appName`. If you use Opera, `navigator.appName` equals `"Opera"`. Unfortunately, some browsers travel incognito. For example, the `navigator.appName` for Firefox is `"Netscape"`. The JavaScript in Firefox is the same as that for Netscape browsers, so in general, it's fine to treat Firefox browsers as Netscape browsers. But, as you can see, if you want to be sure about the browser being used, you can't rely on `naviagor.appName`.

There's a similar rough method for determining the browser version being used: `navigator.appVersion`. Unfortunately, `navigator.appVersion` isn't just a number but a sometimes cryptic string that varies from browser to browser. For example, the Macintosh browser Safari has this nice, simple `navigator.appVersion` string: `"5.0"`. By contrast, Internet Explorer 6.0 running under Windows XP has a `navigator.appVersion` that looks like this: `"4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR 1.1.4322)"`. To see the `navigator.appVersion` string for your browser, type this into the browser's address box (where you normally enter web addresses):

```
javascript:alert(navigator.appVersion)
```

If you care only about whether a person is using a 4.0 browser or later, you can pick out the version numbers from those `navigator.appVersion` strings with the `parseFloat()` command, which looks at the string and grabs the first item that resembles a *floating-point number* (a number that contains a decimal point). Thus the line

```
var browser_version = parseFloat(navigator.appVersion);
```

sets the variable `browser_version` to the first number in the `navigator.appVersion` string. For most browsers, this will be the actual version number. For Internet Explorer, it will be 4.0 for any version of the browser 4.0 or later. You can see why I call this method rough.

More Accurate Browser Detection

JavaScript has another variable that contains information about the browser being used: `navigator.userAgent`. This variable identifies both the manufacturer of the browser and its version. As it did with `navigator.appVersion`, however, the formatting of the string varies from browser to browser.

Because the `navigator.userAgent` strings are different from each other, there is no simple way to extract the information you want. Fortunately, people have already written *browser sniffers*: bits of JavaScript that will do all the hard work of browser identification for you. You can find brwsniff.js, which I downloaded from http://jsbrwsniff.sourceforge.net, at http://www.bookofjavascript.com/Chapter03.

To use this file, put it in the same folder as the web page containing your JavaScript. Then, put this line in the header of your web page:

```
<script type = "text/javascript" src = "brwsniff.js"></script>
```

This tells JavaScript to add the contents of the file named brwsniff.js to your web page. Now you can use the JavaScript stored in that file.

To use the JavaScript in brwsniff.js to determine the name and version of the browser being used to view your web page, add these lines of JavaScript:

```
❶ var browser_info = getBrowser();
❷ var browser_name = browserInfo[0];
❸ var browser_version = browserInfo[1];
```

Line ❶ calls a function in brwsniff.js that reads the `navigator.userAgent` string and compares it to all the different browser version strings it knows. Once it determines the name and version of the browser, the function loads this information into a variable called `browser_info`. All the variables we've seen so far store one piece of information—a string or a number, for example. This `browser_info` variable is an *array*, a type of variable designed to hold multiple items of related information. You'll learn how to work with arrays in Chapter 8. For now it's enough to know that an array is a variable that can store more than one piece of information. Line ❷ puts the first bit of information stored in the array into a variable called `browser_name`. Line ❸ puts the second piece of information stored in `browser_info` into a variable named `browser_version`. Used together, these two variables tell you what kind of browser is viewing the web page. Try the web page in Figure 3-3 on your own browser.

NOTE *This `<script>` tag does not require the `<!--` and `//-->` to hide it from older browsers because there is no code between the opening and closing tags.*

The quick but rough method of browser detection should work for most situations, especially when you don't need to know exactly which browser is being used. For the cases in which you do need the exact name and version, you should use a browser sniffer like the one just described.

```
<html>
<head>
<title>I Know Which Browser You're Using!</title>
<script type = "text/javascript" src = "brwsniff.js"></script>
</head>
<body>
<script type = "text/javascript">
<!-- hide me from older browsers

var browser_info = getBrowser();
var browser_name = browser_info[0];
var browser_version = browser_info[1];
document.write ("You're using " + browser_name + " version " +
    browser_version);

// show me -->
</script>
</body>
</html>
```

Figure 3-3: Finding the browser version number with a browser sniffer

Redirecting Visitors to Other Pages

Now that you understand browser detection, you can tailor your site to provide information specific to each browser. There are two main ways to do this. First, you can use `document.write()`, which we saw in the last chapter, to display one message on your page if the site visitor is using Netscape Navigator 4, and a different message on the same page for Internet Explorer 6.0. Alternatively, you can redirect your visitors to separate pages specifically tailored to different browsers. To redirect visitors to another page, you'd write something like this:

```
window.location.href = "http://www.mywebsite.com/page_for_netscape4.html";
```

When JavaScript sees a line like this, it loads the page with the specified URL into the browser.

NOTE *Are you wondering "What's with all these periods in commands like `window.location.href` and `navigator.appName`?" Never fear. I'll address these when I discuss image swaps and dot notation in Chapter 4.*

In general, it's probably best to use `document.write()` instead of redirecting the user. Because there are so many browsers, trying to maintain a different page for each one can quickly become burdensome. However, if you just want to redirect someone with an older browser to a page that tells them to upgrade, redirection is probably the best way to go.

if-then Statements

Now that you know which browser your visitor is using, you need to learn how to tell JavaScript to write different things depending on the browser being used—in other words, how to implement a *logical test*, choosing between different actions based on specific information. *Branching* is a fundamental technique in any programming or scripting language. Be sure to read this section if you're not already familiar with the concept.

To alter your web pages based on the browser a visitor is using, you tell JavaScript something like, "*If* the visitor is using Internet Explorer, then write this IE-tailored content."

An `if-then` statement in JavaScript looks like this:

```
if (navigator.appName == "Microsoft Internet Explorer")
{
  // write IE-specific content
  document.write("Welcome, Internet Explorer user!");
}
```

Here's the basic structure of an `if-then` statement:

```
if (some test)
{
  statement_1;
  statement_2;
  statement_3;
  ...
}
```

NOTE *JavaScript is unforgiving:* `if` *must be lowercase, and you must put parentheses around the test that follows it.*

The test that appears between the parentheses must be either true or false. If the variable `navigator.appName` equals `"Microsoft Internet Explorer"`, the test between the parentheses is true, and the statements located between the curly brackets are executed. If the variable doesn't equal `"Microsoft Internet Explorer"`, the test between the parentheses is false, and the statements between the curly brackets aren't executed.

Boolean Expressions

The test in the parentheses after `if` is a *Boolean expression*—an expression that's either true or false. In JavaScript, a Boolean expression is usually a statement about the values of one or more variables. Table 3-1 lists some of the symbols you'll be using to form Boolean expressions in JavaScript.

NOTE *Boolean expressions are named after George Boole (1815–1864), who invented a way to express logical statements in mathematical form.*

Table 3-1: Symbols in Boolean Expressions

Test	Meaning	Example (All of These Are True)
`<`	Less than	`1 < 3`
`>`	Greater than	`3 > 1`
`==`	The same as (equal)	`"happy" == "happy", 3 == 3`
`!=`	Different from (not equal)	`"happy" != "crabby", 3 != 2`
`<=`	Less than or equal to	`2 <= 3, 2 <= 2`
`>=`	Greater than or equal to	`3 >= 1, 3 >= 3`

Notice in Table 3-1 that you must use two equal signs when you want JavaScript to test for equality in an `if-then` statement Boolean expression. In fact, accidentally using one equal sign instead of two in an `if-then` statement is probably *the* major cause of mind-blowing programming errors. As you learned in Chapter 2, a single equal sign is used to assign a value to a variable. So if you accidentally use only one equal sign, JavaScript thinks you mean to set the variable on the left of the equal sign to the value of whatever is on the right of the equal sign, and it will act as if the test result is always true.

Here's an example of the trauma that this mistake can cause. Say you want to write a JavaScript that puts *Happy Birthday, Mom!* on your web page when it's your mother's birthday. If her birthday were August 6, you might write something like Figure 3-4 (which contains the dreaded error).

If you try this script, you'll see that it *always* prints *Happy Birthday, Mom!* to the web page, which is great for Mom, but probably not what you want.

```
  <script type = "text/javascript">
  <!-- hide me from older browsers

  var today = new Date();
  var day = today.getDate();
❶ var month = today.getMonth();

❷ if (month = 7) // remember, January is month 0, so August is month 7
  {
❸  if (day = 6)
    {
❹    document.write("<h1>Happy Birthday, Mom!</h1>");
    }
  }

  // show me -->
  </script>
```

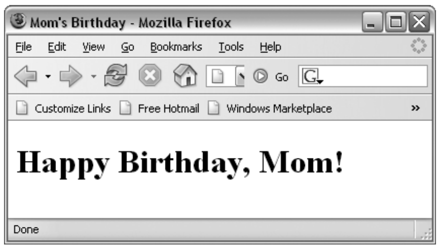
Figure 3-4: Mom's birthday greeting—broken version

The script starts off correctly. When JavaScript sees ❶, it sets the variable `month` to whatever month it is. If you're running the script in March, it sets `month` to 2. The problem arises in the next line, though:

```
if (month = 7)
```

Here JavaScript sees one equal sign and thinks you want to *set* the variable `month` to the value `7`. The script does what you're telling it to do, and then acts as if your test is true.

Since the result is true, JavaScript moves to the curly brackets, where it finds ❸, another `if-then` statement that incorrectly uses one equal sign instead of two. This line sets the variable `day` to the value `6` and again results in a true statement. JavaScript then moves to the second set of curly brackets, where it sees that it's supposed to ❹ write `<h1>Happy Birthday, Mom!</h1>`, which it does—every time someone visits the page (see Figure 3-5).


Mom's Birthday - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Go
Customize Links Free Hotmail Windows Marketplace
Done

Happy Birthday, Mom!

Figure 3-5: Mom's birthday greeting

NOTE *I remember the difference between one and two equal signs by thinking* is the same as *instead of* equals *when I'm doing an* `if-then` *test, and remembering that* is the same as *translates into two equal signs.*

Nesting

Figure 3-4 is the first example I've used of *nesting*—one `if-then` statement inside another. Although it sometimes makes sense to nest your `if-then` statements, things get confusing if you start to get three or more levels deep (one `if-then` statement inside the curly brackets of another `if-then` statement, which itself is inside the curly brackets of a third `if-then` statement).

Try to write your code so that it doesn't need more than two levels of nesting. If you find yourself with `if-then` statements more than two levels deep, it often means that you're doing something complicated enough to justify writing a new function to handle some of the complexity. (More on that in Chapter 6.)

if-then-else Statements

There are a couple of fancier versions of the `if-then` statement. The first is the `if-then-else` statement:

```
if (navigator.appName == "Microsoft Internet Explorer")
{
  // write IE-specific content
  document.write("Welcome, Internet Explorer user!");
}
else
{
  // write netscape specific content
  document.write("Welcome, Netscape user!");
}
```

This reads nicely in English if you read *else* as *otherwise*: "If they're using Internet Explorer, show them IE-specific content, otherwise send them Netscape-specific content."

if-then-else-if Statements

The above code assumes that there are only two browser manufacturers in the world, when in fact there are a multitude. We can solve this problem with an `if-then-else-if` statement that, if a visitor has a browser other than Netscape or Internet Explorer, displays content regarding unknown browsers.

```
if (navigator.appName == "Netscape")
{
  // write netscape-specific content
  document.write("Welcome, Netscape user!");
}
else if (navigator.appName == "Microsoft Internet Explorer")
{
  // write IE-specific content
  document.write("Welcome, Internet Explorer user!");
}
else
{
  // write unknown browser content
  document.write("Welcome, user of a fancy unknown browser!");
}
```

This code reads in English as: "If they're using Netscape, send them Netscape-specific content; if they're using Internet Explorer, send them IE-specific content. Otherwise send them a message about having a mysterious browser."

When and Where to Place Curly Brackets

Notice in the examples above that curly brackets (braces) mark the beginning and end of the body of an `if-then` statement, enclosing the part where you tell JavaScript what action(s) to take. You'll also notice that I place my beginning and ending curly brackets on their own lines, like this:

```
if (something == something_else)
{
  blah_blah_blah;
}
```

This is my style, one that I think makes it easier to align pairs of beginning and ending brackets. Other people prefer this slightly more compact style:

```
if (something == something_else) {
  blah_blah_blah;
}
```

It's up to you to choose where you put the curly brackets. Many studies have tried to figure out which formatting style is most readable or which avoids bugs. When you get right down to it, just decide what you think looks good and go with that.

Sometimes curly brackets are not needed in an `if-then` statement, such as when the body of the statement has only one line. For example, this is legal:

```
if (something == something_else)
  alert("they're equal");
else
  alert("they're different!");
```

Since each of the "then" parts of the clause is only one line (the `alert` functions), the curly brackets around these statements are optional. However, it's always a good idea to include the braces anyway, because you might want to add a second line to that `else` clause. If you do add a second line to the `else` clause and forget to put the brackets around the two lines, your script won't work.

With curly brackets, the previous example would look like this:

```
if (something == something_else)
{
  alert("they're equal");
}
else
{
  alert("they're different!");
}
```

Or, if you prefer the more compact style:

```
if (something == something_else) {
  alert("they're equal");
} else {
  alert("they're different!");
}
```

OR and AND

The `if-then` statements we've seen so far are pretty simple. You might, however, want to add more conditions to an `if-then` statement (for example, "If Joe is in high school *and* is not doing his homework, then tell him to get to work"). To add more conditions to an `if-then` statement, use the *OR* and *AND operators*.

OR

Suppose you want to give different greetings to people who come to your site, depending on who they are. You could, as in Figure 3-6, use a prompt box to ask for a visitor's name (Figure 3-7) and then use an `if-then` statement to determine which greeting to give.

```
<script type = "text/javascript">
<!-- hide me from older browsers

var the_name = prompt("What's your name?", "");
if (the_name == "thau")
{
  document.write("Welcome back, thau! Long time no see!");
} else {
  document.write("Greetings, " + the_name + ". Good to see you.");
}

// show me -->
</script>
```

Figure 3-6: Asking for a visitor's name with the prompt box

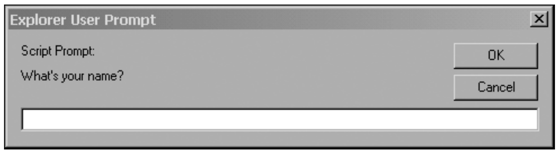
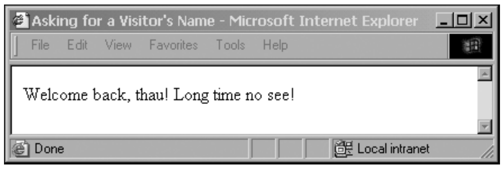


Figure 3-7: The prompt box asking for a visitor's name

This example greets thau with "Welcome back, thau! Long time no see!" (Figure 3-8) and everyone else with "Greetings, *Name*. Good to see you."


Asking for a Visitor's Name - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Done
Local intranet

Welcome back, thau! Long time no see!

Figure 3-8: thau's greeting

To greet others the same way you greet thau, you could use a series of `if-then` statements as in Figure 3-9.

```
if (the_name == "thau")
{
  document.write("Welcome back, thau! Long time no see!");
}
else if (the_name == "dave")
{
  document.write("Welcome back, dave! Long time no see!");
}
```

```
else if (the_name == "pugsly")
{
  document.write("Welcome back, pugsly! Long time no see!");
}
else if (the_name == "gomez")
{
  document.write("Welcome back, gomez! Long time no see!");
}
else
{
  document.write("Greetings, " + the_name + ". Good to see you.");
}
```

Figure 3-9: Personalized greetings with a series of `if-then` *statements*

This would work, but there's a lot of waste here: We repeat basically the same `document.write()` line four times. What we really want to say is something like: "If `the_name` is *thau*, or *dave*, or *pugsly*, or *gomez*, give the 'Long time no see' greeting." JavaScript has a feature called the OR operator, which comes in handy here. Figure 3-10 shows OR in use:

```
if ((the_name == "thau") || (the_name == "dave") ||
    (the_name == "pugsly") || (the_name == "gomez"))
{
  document.write("Welcome back, " + the_name + "! Long time no see!");
}
```

Figure 3-10: The OR operator

The OR operator is represented by two vertical lines (`||`), called *bars*. You will usually be able to type the bar (`|`) character as the shifted backslash (`\`) key on your keyboard.

NOTE *Although each of the Boolean tests in Figure 3-10 (for example,* `the_name == "thau"`*) has its own parentheses, these aren't strictly necessary. However, the set of parentheses around all four Boolean tests* is *required, and it's a good idea to include the other parentheses for legibility's sake.*

AND

AND, another important operator, is represented by two ampersands (`&&`). Figure 3-11 shows this operator in use.

```
var age = prompt("How old are you?", "");
var drinking = prompt("Are you drinking alcohol (yes or no)?", "yes");

if ((age < 21) && (drinking == "yes"))
{
  document.write("Beat it!");
}
else
```

```
{
  document.write("Enjoy the show!");
}
```

Figure 3-11: The AND operator

When bars start using robot bouncers that run on JavaScript, this is the kind of code they'll be running. The script asks a person's age and whether he or she is drinking alcohol (Figure 3-12).

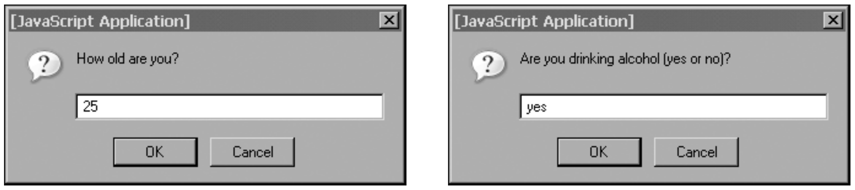
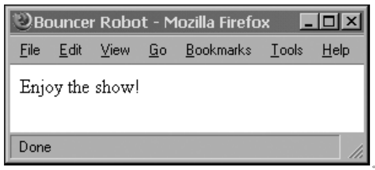


Figure 3-12: The bouncer's questions

If the person is under 21 and is drinking alcohol, the bouncer tells him or her to beat it. Otherwise, the visitor is perfectly legal and is welcome to stay (Figure 3-13). (Never mind the fake IDs for now.)


Bouncer Robot - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Done

Enjoy the show!

Figure 3-13: The bouncer's response

Putting It All Together

Here's a script containing most of what's been presented in the chapter so far. The script in Figure 3-14 redirects users to one page if they're using an older version of Netscape (version 4 or earlier), another page if they're using an older version of Internet Explorer (version 5.5 or earlier), a third page for browsers it's unfamiliar with, and a fourth page for modern browsers it knows about.

I've broken the code into two blocks of `<script>` tags. The first sets up the variables and the second does the redirection.

NOTE *It's a good idea to declare variables at the top of your script. That way, if you want to change a variable later, you won't have to go hunting through a lot of HTML and JavaScript to find it.*

```
<html><head><title>Redirection</title>
<script type = "text/javascript" src = "brwsniff.js"></script>
<script type = "text/javascript">
<!-- hide me from older browsers
```

```
var browser_info = getBrowser();
var browser_name = browser_info[0];
var browser_version = browser_info[1];
var this_browser = "unknown";
if (browser_name == "msie")
{
  if (browser_version < 5.5)
  {
    this_browser = "old Microsoft";
  }
  else
  {
    this_browser = "modern";
  }
} // end if browser_name == Microsoft
if (browser_name == "netscape")
  {
  if (browser_version < 6.0)
  {
    this_browser = "old Netscape";
  }
  else
  {
    this_browser = "modern";
  }
} // end if browser_name == Netscape

// show me -->
</script>
</head><body>
<SCRIPT type = "text/javascript">
<!-- hide me from older browsers
if (this_browser == "old Netscape")
{
  window.location = "archaic_netscape_index.html";
} else if (this_browser == "old Microsoft")  {

  window.location.href = "archaic_ie.html";
} else if (this_browser == "modern")
{
  window.location.href = "modern_browser.html";
}

// show me -->
</script>
<h1>Unknown Browser</h1>
Sorry, but this page only works for browsers Netscape 6.0 and later, and
Internet Explorer 5.5 and later.
</body>
</html>
```

Figure 3-14: Complete redirection code

A Few More Details About Boolean Expressions

There are just a few more things you need to know about Boolean expressions before you can call yourself a Boolean master. You already know that you can create an `if-then` statement using code like this:

```
if (name == "thau") {
  alert("Hello, thau!");
}
```

This says, "If it is true that the variable `name` contains the string `thau`, put up an alert saying *Hello, thau!*" What you may not know is that you can store the value `true` or `false` in a variable and use it later. So, I could have done this instead:

```
var thisIsThau = (name == "thau");
if (thisIsThau == true) {
  alert("Hello, thau!");
}
```

The first line tests to see whether the variable `name` contains the string `"thau"`. If it does, the test is `true`. This `true` value is stored in the variable `thisIsThau`. You can then test to see whether the variable `thisIsThau` is `true`, as seen in the subsequent `if-then` statement. This can be shortened a bit to this:

```
var thisIsThau = (name == "thau");
if (thisIsThau) {
  alert("Hello, thau!");
}
```

Notice that I'm not explicitly checking to see whether `thisIsThau` contains the value `true`. Instead, I'm just putting the variable inside the `if-then` test parentheses. The `if-then` rule states, "If the thing inside the parentheses is true, do the action in the curly brackets." In this case, the variable `isThisThau` will be true if the variable `name` contains the value `"thau"`.

If you wanted to do something in the case where the string stored in `name` was something other than `"thau"` you could do this:

```
var thisIsThau = (name == "thau");
if (thisIsThau == false) {
  alert("Hello, somebody other than thau!");
}
```

Here, we're checking to see whether the value stored inside `thisIsThau` is false, which it will be if the comparison of `name` and `"thau"` turned out to be false in the line above (for example, if `name` equaled `"pugsly"`).

The final shortcut involves using the special character `!`, which means *not*.

```
var thisIsThau = (name == "thau");
if (!thisIsThau) {
  alert("Hello, somebody other than thau!");
}
```

The expression means "if `thisIsThau` is not true, then do the stuff in the curly brackets." These Boolean shortcuts are used quite frequently in the scripts I've seen on the Web, so you should take some time to get used to them.

How Netscape Provides Browser-Specific Content

Now we've covered just about everything you need to know to understand how Netscape serves up the browser-specific content illustrated at the beginning of the chapter (Figures 3-1 and 3-2). Here is a somewhat simplified and modified version of the JavaScript on Netscape's home page:

```
  <script type = "text/javascript">
❶ var agent = navigator.userAgent.toLowerCase();
❷ var major = parseInt(navigator.appVersion);
  var minor = parseFloat(navigator.appVersion);
❸ var ns = ((agent.indexOf('mozilla') != -1) &&
      (agent.indexOf('compatible') == -1));
❹ var ns4 = (ns && (major == 4));
  var ns7 = (ns && (agent.indexOf('netscape/7') != -1) );
  var ie = (agent.indexOf("msie") != -1);
  var ie4 = (ie && (this.major >= 4));
  var ie6 = (ie && (agent.indexOf("msie 6.0") != -1));
  var op3 = (agent.indexOf("opera") != -1);
  </script>
```

Next comes all of the HTML. Inside the HTML, when you want to decide whether or not to write something based on the browser being used, you do something like this:

```
    <script type = "text/javascript">
❺  if (!ns4) document.write('<td>the stuff that puts in the numbers</td>');
     </script>
```

The script starts by using the `userAgent` and `appVersion` variables to determine the type of browser being used. Notice the use of `parseInt()` in ❷. This function works just like `parseFloat()`, except that it pulls the first integer out of a string, rather than the first floating-point number. This will set the variable `major` to a number like 4, 5, or 6.

The next line (❸) is jam-packed with information, so take it slow. The first thing to notice is the use of the `indexOf()` function. We'll see more of `indexOf()` in Chapter 11 when we work with strings. The main thing to know here is that `indexOf()` checks to see whether a string contains another string. To see if the word mozilla is part of the string stored in agent, we use agent.indexOf('mozilla'). If mozilla is in the agent string, indexOf() will return some number other than –1. If mozilla is not part of the agent string, indexOf() will return –1. This can get a little confusing, so make sure you understand that last rule.

Now, looking at ❸, we see that there are two main parts. The first part checks to see whether some application of the indexOf() function gives a result different from –1. The next part checks to see if another application of the indexOf() function gives a result that equals –1. If the first part is true, *and* the second part is also true, then the whole thing is true, and the value true is stored in the variable ns. If either of the comparisons is false, then the whole thing will be false, and the value false will be stored in ns. Remember the bouncer's test:

```
if ((age < 21) && (drinking == "yes"))
```

If both statements were true—the person was under 21, *and* the person was drinking—the person got bounced. If either part was not true, then they were okay.

With all that in mind, let's look to see what the two comparisons in ❸ are. The first one will return the value true if indexOf() finds the string mozilla in the variable agent. Take a long, hard look at the expression:

```
agent.indexOf('mozilla') != -1
```

Remember, if the string stored in variable agent contains the string mozilla, indexOf() will return a value not equal to –1. So this test will be true if the navigator.userAgent has the word mozilla (upper- or lowercase) in it.

The next part makes sure that the navigator.userAgent does not contain the string compatible. This is because many browsers say they are Mozilla compatible, and they'll have both the words mozilla and compatible in their navigator.userAgent string. Netscape just has the word mozilla in its string. The end result of ❸ is that the variable ns will be true if the navigator.userAgent contains the string mozilla but not the string compatible.

The next lines figure out which version of Netscape this might be. Consider ❹:

```
var ns4 = (ns && (major == 4));
```

This line says, "If the variable ns is true, and the variable major has a value of 4, then put the value true in the variable ns4." If it's not true *both* that the variable ns is true *and* that the variable major is 4, then ns4 will be false. The other lines perform similar tests for Navigator 7 and other browsers. Each one is a little different from the others, so make sure you take some time to understand all of them.

Once the browser is known, the decision whether or not to display the browser-specific feature (namely, the page number navigation links) happens later in the code. Right at the place where you either write something to the web page or not, depending on the browser being used, you use a line like ❺:

```
if (!ns4) document.write('<td>the stuff that puts in the numbers</td>');
```

This says, "If this is not a Netscape 4 browser, write the code that puts in the navigation element." The variable `ns` will be `true` if the earlier code determined that it was a Netscape 4 browser being used, and `false` otherwise. Remember that this code must go between `<script>` and `</script>` tags.

Except for the part of the script that determines the type of browser being used, the Netscape code is fairly simple. If you want to avoid the complexities involved in determining the browser being used, use one of the browser sniffer packages available for free on the Web, incorporating the software into your page using JavaScript statements similar to those shown in the section "More Accurate Browser Detection" on page 36.

Summary

Here are the things you should remember from this chapter:

- JavaScript's tools for identifying a visitor's browser (`navigator.appName`, `navigator.appVersion`, and `navigator.userAgent`)
- How `if-then`, `if-then-else`, and `if-then-else-if` statements work
- How Boolean expressions work
- How to redirect your visitors to other web pages
- How to import JavaScript from another file

Did you get all that? If so, here's an assignment for you.

Assignment

Write a web page that asks for a visitor's name. If the visitor is someone you like, send him to your favorite page. If it's someone you don't know, send him to a different page. And if it's someone you don't like, send him to yet another page.

4

WORKING WITH ROLLOVERS

You've seen rollovers a million times. You mouse over an image, and the image changes. You mouse off the image, and the image changes back to its original state. Rollovers are an easy way to make your site more interactive.

This chapter will show you how to create a good rollover. This involves:

- Telling JavaScript to detect the mouse event that will trigger an image swap
- Telling JavaScript which of several images to swap in, based on the mouse event
- Replacing the old image with a new one

I'll also teach you a new way to detect which browser a visitor is using.

A Real-World Example of Rollovers

To begin, let's take a look at rollovers in action. *Tin House* (http://www.tinhouse.com), one of my favorite literary journals, has a little house on its home page that helps you navigate the site. When you first come to the page, all the lights in the house are off (Figure 4-1); rolling over different parts of the house lights those areas up (Figure 4-2). It may be a little silly, but I like it.

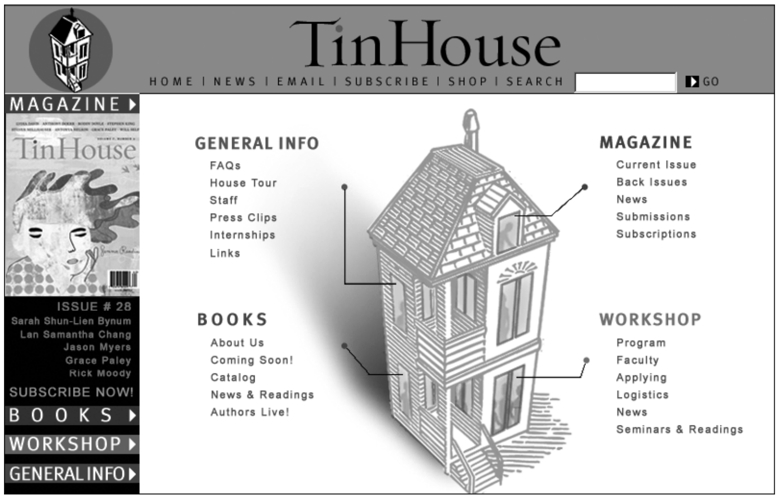


Figure 4-1: Tin House *home page before mousing over the house*

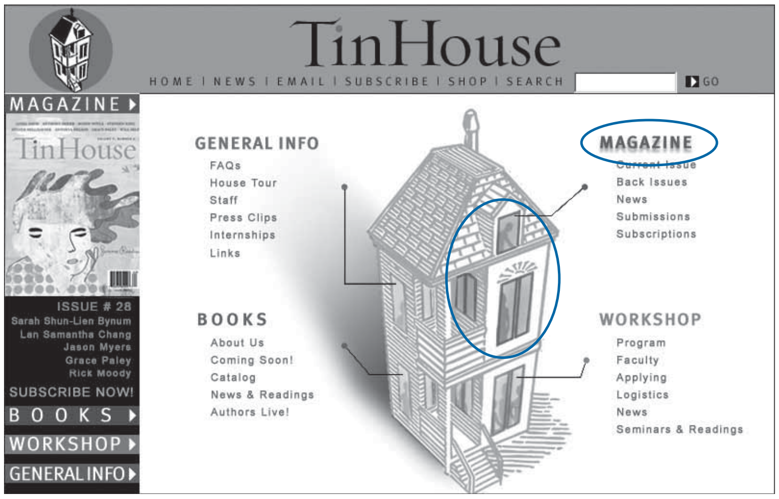


Figure 4-2: Tin House *home page with mouse over the house*

The *Book of JavaScript* home page also has a relatively straightforward and uncomplicated implementation of an image swap. If you mouse over the graphic that says *Turn it over!* the image of the front cover of the book will change to show the back of the book (see Figures 4-3 and 4-4). Mouse off the *Turn it over!* image again and the book image switches back to the front cover.

There are many ways to script a rollover. Because rollovers don't work in old browsers, or when people turn JavaScript off, creating them also involves browser detection, so in this chapter you'll learn more ways to tailor JavaScripts to the visitor's browser.

You'll also learn how quotation marks are handled in JavaScript and how the hierarchical framework of a web page, known as the *Document Object Model (DOM)*, is reflected in JavaScript syntax.

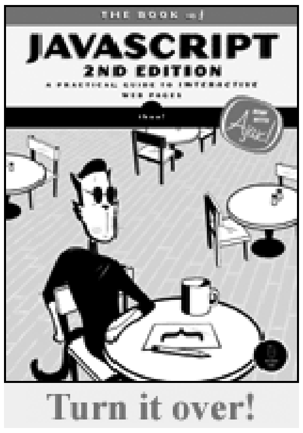


Figure 4-3: An image from the *Book of JavaScript* home page before mouseover

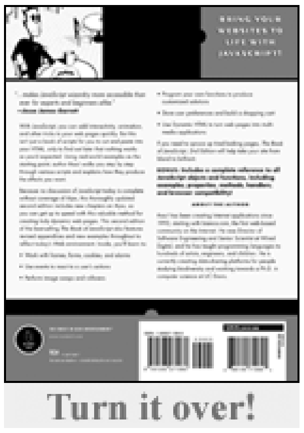


Figure 4-4: The same image after mouseover

Triggering Events

So far all the JavaScript we've seen is triggered when a web page loads into a browser. But JavaScript can also be *event driven.*

Event-driven JavaScript waits for your visitor to take a particular action, such as mousing over an image, before it reacts. The key to coding event-driven JavaScript is to know the names of events and how to use them.

Event Types

With JavaScript's help, different parts of your web page can detect different events. For example, a pull-down menu can know when it has changed (see Chapter 7); a window when it has closed (see Chapter 5); and a link when a visitor has clicked on it. In this chapter I'll focus on link events.

A link can detect many kinds of events, all of which involve interactions with the mouse. The link can detect when your mouse moves over it and when your mouse moves off of it. The link knows when you click down on it, and whether, while you're over the link, you lift your finger back off the button after clicking down. The link also knows whether the mouse moves while over the link.

Like the other kinds of interactions that we'll cover in later chapters, all of these events are captured in the same way: using an *event handler.*

onClick

Figure 4-5 shows the basic format of a link that calls an alert after a visitor clicks it.

Before adding JavaScript:

```
<a href = "http://www.bookofjavascript.com/">Visit the Book of JavaScript
    website</a>
```

After adding JavaScript:

```
<a href = "http://www.bookofjavascript.com/"
  onClick = "alert('Off to the Book of JavaScript!');">Visit the Book of
    JavaScript website</a>
```

Figure 4-5: A link that calls an alert

Try putting the link with the `onClick` into one of your own web pages. When you click the link, an alert box should come up and say *Off to the Book of JavaScript!* (Figure 4-6). When you click OK in the box, the page should load the *Book of JavaScript* website.

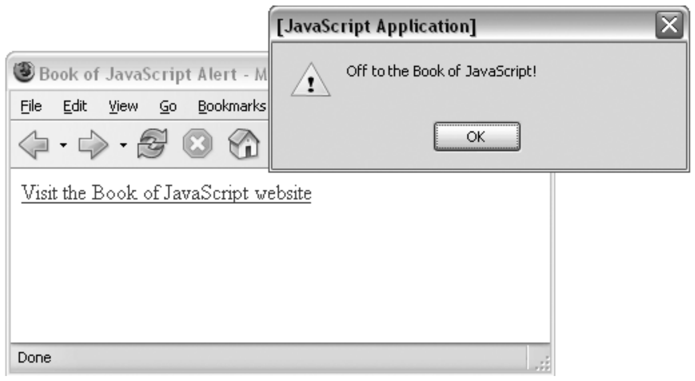

[JavaScript Application]
Off to the Book of JavaScript!
OK
Book of JavaScript Alert - M
File Edit View Go Bookmarks
Visit the Book of JavaScript website
Done

Figure 4-6: The event-driven "Off to the Book of JavaScript!" alert box

Notice that, aside from the addition of `onClick`, this enhanced link is almost exactly like the normal link. The `onClick` event handler says, "When this link is clicked, pop up an alert."

onMouseOver and onMouseOut

Two other link events are `onMouseOver` and `onMouseOut`. Moving the mouse over a link triggers `onMouseOver`, as shown in Figure 4-7.

```
<a href = "#" onMouseOver = "alert('Mayday! Mouse overboard!');">board</a>
```

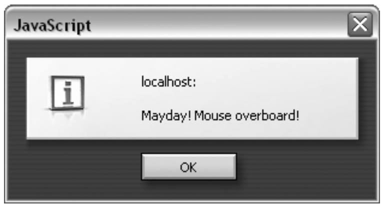
Figure 4-7: `onMouseOver`

As you can see, moving the mouse over the link triggers `onMouseOver`. The code for `onMouseOut` looks like the `onMouseOver` code (except for the handler name) and is triggered when the mouse moves off of the link. You can use `onMouseOut`, `onMouseOver`, and `onClick` in the same link, as in Figure 4-8.

```
<a href = "#"
  onMouseOver = "alert('Mayday! Mouse overboard!');"
  onMouseOut = "alert('Hooray! Mouse off of board!!');"
  onClick = "return false;">
board
</a>
```

Figure 4-8: `onMouseOut`, `onMouseOver`, and `onClick` in the same link

Mousing over this link results in an alert box showing the words *Mayday! Mouse overboard!* (Figure 4-9). Pressing ENTER to get rid of the first alert and moving your mouse off the link results in another alert box that contains the words *Hooray! Mouse off of board!!* If you click the link instead of moving your mouse off it, nothing will happen, because of the `return false;` code in the `onClick`.



localhost:

Mayday! Mouse overboard!

Figure 4-9: An alert box produced by mousing over a link

onMouseMove, onMouseUp, and onMouseDown

The `onMouseMove`, `onMouseUp`, and `onMouseDown` event handlers work much like the others. Try them yourself and see. The `onMouseMove` event handler is called whenever the mouse is moved while it is over the link. The `onMouseDown` event handler is triggered when a mouse button is pressed down while the mouse is over a link. Similarly, the `onMouseUp` event handler is triggered when the mouse button is lifted up again. An `onClick` event handler is triggered whenever an `onMouseDown` event is followed by an `onMouseUp` event.

Quotes in JavaScript

This example also demonstrates a new wrinkle in JavaScript syntax. Inside the double quotes of the `onClick` (Figure 4-8) is a complete line of JavaScript, semicolon and all. In previous chapters, we've placed all of our JavaScript between opening `<script>` and closing `</script>` tags. The only exception to this rule is when JavaScript is inside the quotes of an event. Your browser will assume that anything within these quotes is JavaScript, so you shouldn't put `<script>` and `</script>` tags in there.

Also note that the quotes in the alert are single quotes ('). If these were double quotes ("), JavaScript wouldn't be able to figure out which quotes go with what. For example, if you wrote

```
onClick = "alert("Off to the Book of JavaScript!");"
```

JavaScript would think that the second double quote closed the first one, which would confuse it and result in an error. Make sure that if you have quotes inside quotes, one set is double and the other is single.

Apostrophes can also pose problems. For example, let's say you want the alert in Figure 4-7 to say

```
Here's the Book of JavaScript page. You're gonna love it!
```

You would want the JavaScript to resemble this:

```
onClick = "alert('Here's the Book of JavaScript page. You're gonna love it!');"
```

Unfortunately, JavaScript reads the apostrophes in `Here's` and `You're` as single quotes inside single quotes and gets confused. If you really want those apostrophes, *escape* them with a backslash (\), like this:

```
onClick = "alert('Here\'s the Book of JavaScript page. You\'re gonna love it!');"
```

Putting a backslash before a special character, such as a quote, tells JavaScript to print the item rather than interpret it.

Clicking the Link to Nowhere

You may have noticed that the links in Figures 4-7 and 4-8 have an unusual form for the `href` attribute:

```
<a href = '#'>
```

This hash mark (#) in an `href` means, "Go to the top of this page." I've included it there because most browsers expect something to be inside the quotes after the `href`, usually a URL. In Figure 4-5, for example, the tag is

```
<a href = "http://www.bookofjavascript.com/">
```

In HTML, `href` is a required attribute of the anchor (`<a>`) tag, or link. `href` is an abbreviation for *hypertext reference*, and it's required because, as far as HTML is concerned, the whole purpose of a link is to send the user somewhere else when the link is clicked, so the browser needs to be told where to go. Usually that's another page, but in this case you are not trying to go anywhere. I might have just put nothing inside the quotes (`href = ""`), but different browsers will do different things in that case, and it's usually something weird. Give it a try in your favorite browser. To avoid weird behaviors, it's best to put the `#` sign inside an `href` when you don't want the link to go anywhere when clicked.

The link in Figure 4-8 had a second way of ensuring that the link didn't go anywhere when clicked: `onClick = "return false;"`. Placing `return false;` in the quotes after an `onClick` tells JavaScript to prevent the browser from following the URL inside the link's `href`. This can be quite useful for dealing with people who have JavaScript turned off in their browsers. For example, if someone with JavaScript turned off clicks the link in Figure 4-10, the browser will ignore the `onClick` and happily follow the URL inside the `href`. This URL might go to a web page that describes the wonders of JavaScript and tells the user how to turn JavaScript on. People who already have JavaScript turned on will be treated to the contents of the `onClick`. They will see an alert box, and then the `return false` inside the `onClick` will prevent the browser from following the URL in the `href`. Although very few people turn JavaScript off (fewer than 1 percent of browsers), it never hurts to take them into consideration.

```
<a href = "please_turn_js_on.html" onClick =
    "alert('I\'m glad you have JavaScript turned on!'); return false;">Click me</a>
```

Figure 4-10: Links for people with JavaScript turned off

More Interesting Actions

You can do more with event handlers than simply triggering alert boxes. Figure 4-11, for instance, uses an event handler to customize a page's background color.

```
<a href = "#"
  onClick = "var the_color = prompt('red or blue?','');
    window.document.bgColor = the_color;
    return false;">
change background</a>
```

Figure 4-11: Customizing background color

When you click this link, a prompt box asks whether you want to change the background to red or blue. When you type your response, the background changes to that color. In fact, you can type whatever you want into that prompt box, and your browser will try to guess the color you mean. (You can even do a kind of personality exam by typing your name into the prompt and seeing what color your browser thinks you are. When I type `thau` into the prompt, the background turns pea green.)

This example demonstrates two new facts about JavaScript. First, notice that the `onClick` triggers three separate JavaScript statements. You can put as many lines of JavaScript as you want between the `onClick`'s quotes, although if you put too much in there, the HTML starts to look messy.

Second, notice that you can change the background color of a page by setting `window.document.bgColor` to the color you desire. To make the background of a page red, you'd type:

```
window.document.bgColor = 'red';
```

In the example, we're setting the background color to any color the user enters into the prompt box. I'll say more about `window.document.bgColor` soon.

Swapping Images

Using JavaScript, you can change or swap images on your web pages, making buttons light up, images animate, and features explain themselves. Before you tell JavaScript to swap an image, you have to tell it what image to swap by naming the image. Figure 4-12 shows you how.

Before JavaScript:

```
<img src = "happy_face.gif">
```

After JavaScript:

```
<img src = "happy_face.gif" name = "my_image">
```

Figure 4-12: Naming an image

In this example, I've put an image of a happy face on the page and named it `my_image`.

NOTE *You can name an image whatever you like, but the name can't contain spaces.*

Once you've named an image, it's easy to tell JavaScript to swap it with a new one. Let's say you have an image named `my_image`. To create an image swap, tell JavaScript you want to change the `src` of that image to `another.gif`:

```
window.document.my_image.src = "another.gif";
```

Figure 4-13 shows the code for a very basic page with an image and a link; click the link, and the image changes to `happy_face.gif` (Figure 4-14).

```
<html><head><title>Simple Image Swap</title></head>
<body>
<img src = "sad_face.gif" name = "my_image">
<br>
<a href = "#"
  onClick = "window.document.my_image.src = 'happy_face.gif';
```

```
    return false;">make my day!</a>
</body>
</html>
```

Figure 4-13: JavaScript for a basic image swap

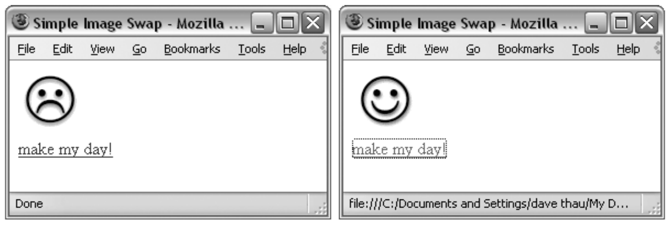

Simple Image Swap - Mozilla ...
File Edit View Go Bookmarks Tools Help
make my day!
Done
Simple Image Swap - Mozilla ...
File Edit View Go Bookmarks Tools Help
make my day!
file:///C:/Documents and Settings/dave thau/My D...

Figure 4-14: Swapping a sad face for a happy one

Working with Multiple Images

If you have more than one image on a page, you should give each one a different name. Figure 4-15 has two images and two links. The first link tells JavaScript to swap the image called `my_first_image` (the sad face) with `happy_face.gif`. The second link tells JavaScript to swap the image called `my_second_image` (a circle) with `square.gif`. The result is shown in Figure 4-16.

NOTE *When using more than one image, you must name your images differently. If you accidentally give two images the same name, the swap won't work.*

```
<html><head><title>Two Image Swaps</title></head>
<body>
<img src = "sad_face.gif" name = "my_first_image"><br>
<img src = "circle.gif" name = "my_second_image">
<br>
<a href = "#"
  onClick = "window.document.my_first_image.src = 'happy_face.gif';
    return false;">make my day!</a>
<br>
<a href = "#"
  onClick = "window.document.my_second_image.src = 'square.gif';
    return false;">square the circle!</a>
</body>
</html>
```

Figure 4-15: JavaScript for swapping two images

NOTE *Image swapping doesn't work in browsers earlier than Internet Explorer 4.0 or Netscape 3.0. Furthermore, if you're trying to replace a small image with a bigger one, or a big image with a smaller one, browsers earlier than Netscape 4.61 and Internet Explorer 4.0 will squash or stretch the new image to fit the space the old one occupied. Later versions of these browsers adjust the page to fit the bigger or smaller image.*

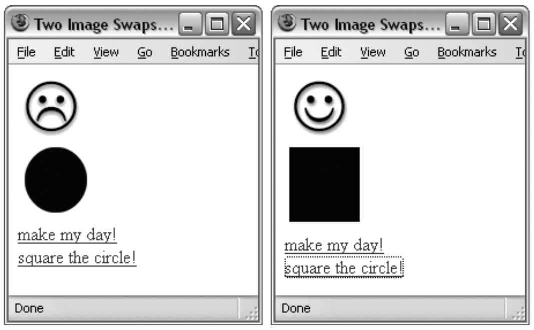


Figure 4-16: Swapping two images

What's with All the Dots?

You may wonder why JavaScript refers to `my_image` as `window.document.my_image` and not just as `my_image`. You may also wonder why you would use `window.document.my_image.src` when you want to change the `src` of that image. In short, what's with all the dots?

The answer has to do with how your browser looks at your web page.

Figure 4-17 shows the hierarchical organization of a web page as JavaScript understands it—through the Document Object Model (DOM). At the top of the DOM is the `window` that contains the web page you're viewing. That window contains the `navigator`, `document`, and `location` objects. Each of these objects has a lot of information in it, and by changing one you can change what happens on your web page.

The dots in a line of JavaScript separate hierarchical levels of objects. When JavaScript sees a series of objects separated by dots, it goes to the last object in the series. So, for example, the phrase `window.location` tells JavaScript to find the `location` object inside the current `window`. Similarly, the line `window.document.my_image.src` tells JavaScript to find the source file (`src`) of the image named `my_image` within the `document` object in the current `window`. The current window is the one in which the JavaScript is located.

The document Object

The `document` object lists all the images, links, forms, and other stuff on a web page. To code an image swap, we must tell JavaScript to find the `document` object in the window, then locate the `image` object we would like to change in the `document` object's list, and finally change the image's `src`. In JavaScript terms (where `happy_face.gif` is the image we're swapping in), this is how it looks:

```
window.document.my_image.src = "happy_face.gif";
```

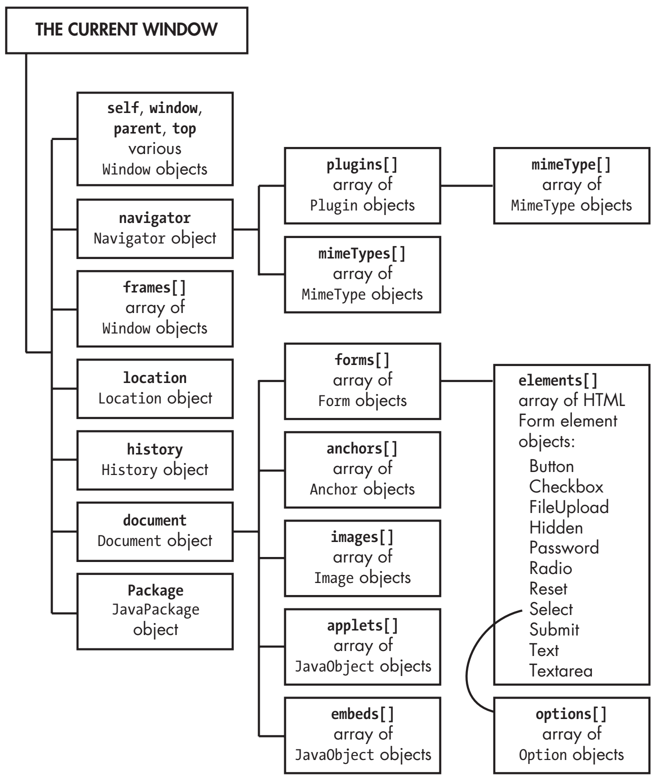

THE CURRENT WINDOW
self, window, parent, top various Window objects
navigator Navigator object
frames[] array of Window objects
location Location object
history History object
document Document object
Package JavaPackage object
plugins[] array of Plugin objects
mimeTypes[] array of MimeType objects
mimeType[] array of MimeType objects
forms[] array of Form objects
anchors[] array of Anchor objects
images[] array of Image objects
applets[] array of JavaObject objects
embeds[] array of JavaObject objects
elements[] array of HTML Form element objects: Button Checkbox FileUpload Hidden Password Radio Reset Select Submit Text Textarea
options[] array of Option objects

Figure 4-17: DOM's hierarchical organization

Object Properties

An object's *properties* are the bits of information that describe the object, such as its height, width, and `src` (the name of the file that the image displays). Some properties, such as the `src` of an image, can be changed, and others can't. As we've seen, changing the `src` property of an `image` object changes which file is displayed:

```
window.document.my_image.src = "happy_face.gif";
```

Other properties, like the image's height and width, are read-only and cannot be changed.

The `document` object contains the `image` objects, and it has its own properties. For example, the background color of the `document` object is called `bgColor`. That's why we could change the background color of our document using `window.document.bgColor = 'red'`. The `image` and `document` objects are just two of many objects we'll be seeing throughout the book. Each JavaScript object has its own set of properties. Appendix C provides a list of many JavaScript objects and their properties.

Finally, Rollovers!

Now that we know how to tell JavaScript how to do an image swap and how to trigger JavaScript based on a user event with onClick, onMouseOver, and onMouseOut, we can create a rollover. Just stick an onMouseOver and onMouseOut inside an image tag, like this:

```
<img src = "sad_face.gif" name = "my_first_image"
  onMouseOver = "window.document.my_first_image.src = 'happy_face.gif';"
  onMouseOut = "window.document.my_first_image.src = 'sad_face.gif';"
>
```

See how that works? When first loaded, the image shows the sad_face.gif because that's what the image tag calls.

```
<img src = "sad_face.gif" name = "my_first_image"...
```

Then, when the mouse moves over the image, the link around it captures the onMouseOver, and the image swaps to happy_face.gif, like so:

```
onMouseOver = "window.document.my_first_image.src = 'happy_face.gif';"
```

When the mouse moves off the image again, the link captures the onMouseOut event, which causes JavaScript to swap sad_face.gif back into the image:

```
onMouseOut = "window.document.my_first_image.src = 'sad_face.gif';"
```

Alternatively, the onMouseOut and onMouseOver could have gone inside an HTML link, as we've done with onClick in earlier examples. Because there are still a few people using browsers that don't allow onMouseOut and onMouseOver handlers inside <img> tags, it's not a bad idea to put them in a link surrounding the image:

```
<a href = "#"
  onMouseOver = "window.document.my_first_image.src = 'happy_face.gif';"
  onMouseOut = "window.document.my_first_image.src = 'sad_face.gif';">
<img src = "sad_face.gif" name = "my_first_image" border = "0">
</a>
```

Image Preloading

That's pretty much all there is to your basic image swap. As usual, there's something that makes the process a little more difficult. When you do an image swap as I've described, the image that's swapped in downloads only when your visitor mouses over the image. If your network connection is slow or the image is big, there's a delay between the mouseover and the image swap.

The way around this potential download delay is to *preload* your images—grabbing them all before they're needed and saving them in the browser's cache. When the mouse moves over a rollover image, the browser first looks to see whether the swap image is in its cache. If the image is there, the browser doesn't need to download the image, and the swap occurs quickly.

There are hundreds of image preloading scripts, and they're all basically the same. Rather than write your own, you can download one of the free ones and paste it into your page (Webmonkey has a good one at http://www.hotwired.com/webmonkey/reference/javascript_code_library/wm_pl_img). Let's go over the basics of how preloads work so you'll recognize them when you see them.

There are two parts to a preload. First, you create a new `image` object. The line

```
var new_image = new Image();
```

creates a new `image` object that has no information. It doesn't have a GIF or JPEG associated with it, nor does it have a height or width. If you know the height and width of the image, you can do this:

```
var new_image = new Image(width, height);
```

Giving JavaScript information about the size of the image helps the browser allocate memory for the image; it doesn't have much impact on how users experience your web page.

Once you've created this new object,

```
new_image.src = "my_good_image.gif";
```

forces the browser to download an image into its cache by setting the `image` object's `src`. When the image is in the browser's cache, it can be swapped for another image without any download delays. Figure 4-18 incorporates a preload with the rollover we saw in the last example.

```
<html><head><title>Preloaded Rollover</title>
<script type = "text/javascript">
<!-- hide me from older browsers

var some_image = new Image();
some_image.src = "happy_face.gif";

// show me -->
</script>
```

```
</head>
<body>
<img src = "sad_face.gif" name = "my_first_image"
  onMouseOver = "window.document.my_first_image.src = 'happy_face.gif';"
  onMouseOut = "window.document.my_first_image.src = 'sad_face.gif';">
</body>
</html>
```

Figure 4-18: Image preload and rollover

How the *Tin House* Rollovers Work

At the beginning of the chapter, I mentioned the home page of *Tin House*. Its image swap JavaScript is quite simple and will give you an idea of how easy it is to add a little JavaScript to your site. The Tin House rollover involves four images: the top, middle, left, and right parts of the bottom floor of the house. These images are placed in an HTML table to create the complete image of the house. Figure 4-19 shows you the (abbreviated) code for the top floor.

```
❶ <a href = "general_info/submission.html"
❷   onMouseOver = "attic3.src='images/index/home_attic3b.gif';"
❸   onMouseOut = "attic3.src='images/index/home_attic3.gif';">
❹     <img src = "images/index/home_attic3.gif" name =
      "attic3" width = "289" height = "68" border = "0" alt =
      "General Information: Our history, our glory, and our guidelines.">
  </a>
```

Figure 4-19: Rollover from the Tin House *home page*

This should look very familiar by now. Line ❹ describes the image. Notice that *Tin House* puts `width`, `height`, `border`, and `alt` attributes inside their image tag as well as the `name` attribute used to do the image swap. The `height` and `width` attributes tell web browsers how much space to reserve for the image. The `alt` attribute does two important things. First, some browsers don't display images. This might be because the person is on a slow connection and has turned images off in their browser, or because they using are a device that can read web pages to them, or perhaps it's a search engine visiting your web page, looking for stuff to add to its index. The `alt` attribute in an image provides information about that image in all of these situations. In addition, the `alt` attribute is used by some browsers even when images are being displayed. Recent versions of Internet Explorer, for example, will display the `alt` text in a yellow box when you leave your mouse over an image for more than a second or two.

Getting back to the JavaScript, you can see that *Tin House* has put its `onMouseOver` (❷) and `onMouseOut` (❸) inside an HTML link (❶). As we've seen, the `onMouseOver` and `onMouseOut` event handlers can go either in the image itself, or in a link surrounding the image, as *Tin House* has done.

Summary

In this chapter you've learned:

- How to trigger events, such as `onMouseOver` and `onMouseOut`
- How to nullify a link with `return false` inside `onClick`
- How to change the background color of a page
- How to swap images
- How to preload images so that they'll swap in more quickly
- How the DOM uses dots to separate objects into hierarchies

Now that you know the basics of image swapping, you can perform lots of tricks. You can make an image vanish by swapping in one that's the same color as the page background. You can make images composed of explanatory text and place them next to the feature they describe. There's no end to the fun you can have with image swaps.

As always, we'll be revisiting many of these points in later chapters, so you'll become more and more familiar with them. To make sure you understand how they work, try the following assignment.

Assignment

Figures 4-20 and 4-21 show you a page which does two image swaps simultaneously. Notice that mousing over the text on the bottom of the screen changes the words from *turn over* to *turn back* and swaps the book's front cover with its back cover. The words, like the book covers, are images, and they are swapped using the techniques we've learned in this chapter. Your assignment is to write a similar page where mousing over one image causes two images to change.

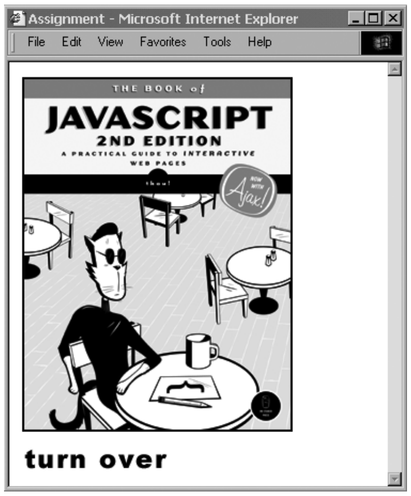

Assignment - Microsoft Internet Explorer
File Edit View Favorites Tools Help
THE BOOK of
JAVASCRIPT
2ND EDITION
A PRACTICAL GUIDE TO INTERACTIVE WEB PAGES
thau!
NOW WITH Ajax!
turn over

Figure 4-20: The Chapter 4 assignment page before the rollover

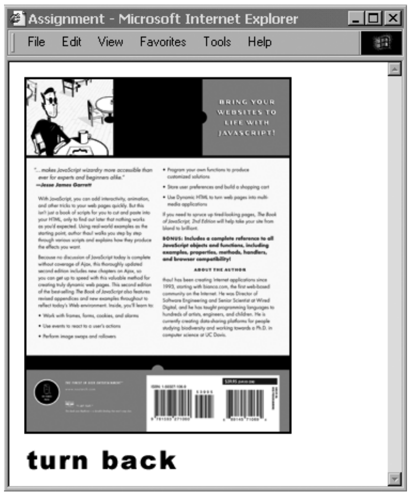

Assignment - Microsoft Internet Explorer
File Edit View Favorites Tools Help
BRING YOUR WEBSITES TO LIFE WITH JAVASCRIPT!
turn back

Figure 4-21: The Chapter 4 assignment page after the rollover