

Revised
BESTSELLER
Expanded

VISUAL QUICKSTART GUIDE



JAVASCRIPT & AJAX

SIXTH EDITION

*Learn JavaScript
and Ajax the Quick
and Easy Way!*

**TOM NEGRINO
DORI SMITH**

FOR THE WEB



VISUAL QUICKSTART GUIDE

JAVASCRIPT AND AJAX

FOR THE WEB

Sixth Edition

**Tom Negrino
Dori Smith**

 **Peachpit Press**

Visual QuickStart Guide

JavaScript and Ajax for the Web, Sixth Edition

Tom Negrino and Dori Smith

Peachpit Press

1249 Eighth Street

Berkeley, CA 94710

(510) 524-2178

(800) 283-9444

(510) 524-2221 (fax)

Find us on the Web at www.peachpit.com

To report errors, send a note to errata@peachpit.com

Peachpit Press is a division of Pearson Education

Copyright © 2007 by Tom Negrino and Dori Smith

Editor: Nancy Davis

Production Coordinator: Tracey Croom

Proofreader: Tiffany Taylor

Compositor: Danielle Foster

Indexer: Ron Strauss

Cover design: Peachpit Press

Notice of rights

All rights reserved. No part of this book may be reproduced or transmitted in any form by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. For information on getting permission for reprints and excerpts, contact permissions@peachpit.com

Notice of liability

The information in this book is distributed on an "As is" basis, without warranty. While every precaution has been taken in the preparation of the book, neither the authors nor Peachpit Press, shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained in this book or by the computer software and hardware products described in it.

Trademarks

Visual QuickStart Guide is a registered trademark of Peachpit Press, a division of Pearson Education.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Peachpit Press was aware of a trademark claim, the designations appear as requested by the owner of the trademark. All other product names and services identified throughout this book are used in editorial fashion only and for the benefit of such companies with no intention of infringement of the trademark. No such use, or the use of any trade name, is intended to convey endorsement or other affiliation with this book.

ISBN 0-321-43032-8

0987654321

Printed in the United States of America

TABLE OF CONTENTS

Dedication:

To the memory of Bill Horwitz and
Dorothy Negrino, because they loved
learning.

Special Thanks to:

Big thanks to our editor Nancy Davis; her expert touch, warm compassion, and fierce dedication always make our work better.

Thanks to Tracey Croom for her excellent production work.

Our heartfelt thanks to Danielle Foster, the book's compositor, who laid out the book under incredible time pressure and pulled off the job with grace and aplomb.

We're grateful to Peachpit's Nancy Ruenzel for her support.

Thanks to our son Sean, for understanding when his parents got cranky, and for making himself dinner on too many occasions.

We'd like to express our special thanks to all of the high school, college, and university instructors who chose to use the previous editions of this book as a textbook for their classes.

Dori would like to thank the wonderful ladies in the W&S group for their loving kindness and virtual hugs. And thanks are also due to the Wise Women's Web Design community (<http://www.wise-women.org>) for their patience, support, and inspiration as role models.

Music is the fuel for Tom's writing, and this book is no exception. The soundtrack and musical inspiration for this sixth edition was graciously provided by lots of uncool bouncy pop music that will remain unidentified, Wolfgang Mozart, Patty Griffin, Richard Shindell, KT Tunstall, Bruce Springsteen, Rosanne Cash, REM, Antonio Vivaldi, Pandora Radio, and the shuffle feature in iTunes.

TABLE OF CONTENTS

	Introduction	xi
Chapter 1:	Getting Acquainted with JavaScript	1
	What JavaScript Is	2
	JavaScript Isn't Java	3
	Where JavaScript Came From	5
	What JavaScript Can Do	6
	What JavaScript Can't Do	7
	What Is Ajax?	8
	The Snap-Together Language	11
	Handling Events	14
	Values and Variables	15
	Assignments and Comparisons	16
	Writing JavaScript-Friendly HTML	17
	What Tools to Use?	20
Chapter 2:	Start Me Up!	21
	Where to Put Your Scripts	23
	About Functions	25
	Using External Scripts	26
	Putting Comments in Scripts	29
	Alerting the User	31
	Confirming a User's Choice	33
	Prompting the User	35
	Redirecting the User with a Link	37
	Using JavaScript to Enhance Links	39
	Working with Referrer Pages	42
Chapter 3:	Language Essentials	45
	Around and Around with Loops	46
	Passing a Value to a Function	51
	Detecting Objects	53
	Working with Arrays	55
	Working with Functions That Return Values	57
	Updating Arrays	58
	Using Do/While Loops	60
	Calling Scripts Multiple Ways	62

Using Multi-level Conditionals.....	64
Handling Errors.....	67
Chapter 4: Working with Images	69
Creating Rollovers.....	71
Creating More Effective Rollovers.....	73
Building Three-State Rollovers.....	81
Triggering Rollovers from a Link.....	83
Making Multiple Links Change a Single Rollover.....	86
Working with Multiple Rollovers.....	88
Creating Cycling Banners.....	92
Adding Links to Cycling Banners.....	94
Building Wraparound Slideshows.....	96
Displaying a Random Image.....	99
Cycling Images with a Random Start.....	101
Chapter 5: Frames, Frames, and More Frames	103
Keeping a Page out of a Frame.....	105
Forcing a Page into a Frame.....	107
Forcing a Site into a Frame.....	108
Setting a Target.....	112
Creating and Loading a Dynamic Frame.....	113
Sharing Functions between Frames.....	116
Storing Information in Frames.....	119
Loading Multiple Frames at Once.....	122
Working with Iframes.....	124
Loading Iframes with JavaScript.....	127
Chapter 6: Working with Browser Windows	129
Opening a New Window.....	130
Loading Different Contents into a Window.....	134
Opening Multiple Windows.....	136
Opening Multiple Windows Simultaneously.....	138
Updating One Window from Another.....	140
Creating New Pages with JavaScript.....	143
Closing a Window.....	146
Opening Windows in a Specified Location.....	149
Moving Windows to a Specified Location.....	152
Chapter 7: Form Handling	155
Select-and-Go Navigation.....	157
Changing Menus Dynamically.....	162
Making Fields Required.....	165
Checking Fields against Each Other.....	170
Identifying Problem Fields.....	172

	Putting Form Validation into Action.....	175
	Working with Radio Buttons.....	179
	Setting One Field with Another.....	182
	Validating Zip Codes.....	185
	Validating Email Addresses.....	189
Chapter 8:	Forms and Regular Expressions	195
	Validating an Email Address with Regular Expressions.....	197
	Validating a File Name.....	202
	Extracting Strings.....	204
	Formatting Strings.....	207
	Formatting and Sorting Strings.....	211
	Formatting and Validating Strings.....	213
	Replacing Elements using Regular Expressions.....	216
Chapter 9:	Handling Events	219
	Handling Window Events.....	220
	Mouse Event Handling.....	228
	Form Event Handling.....	236
	Key Event Handling.....	240
Chapter 10:	JavaScript and Cookies	243
	Baking Your First Cookie.....	245
	Reading a Cookie.....	248
	Showing Your Cookies.....	249
	Using Cookies as Counters.....	251
	Deleting Cookies.....	254
	Handling Multiple Cookies.....	256
	Displaying "New to You" Messages.....	258
Chapter 11:	Introducing CSS	263
	Saying It with Style.....	265
	Styles with Class.....	268
	Changing Fonts with CSS.....	270
	Checking Your ID.....	271
	Distinguished Links.....	273
	Embedding Styles in Styles.....	275
	Adding Background Images.....	277
	Positioning Absolutely.....	279
Chapter 12:	Objects and the DOM	281
	About Node Manipulation.....	282
	Adding Nodes.....	284
	Deleting Nodes.....	286
	Deleting Specific Nodes.....	288

Inserting Nodes	292
Replacing Nodes	295
Chapter 13: Making Your Pages Dynamic	299
Putting the Current Date into a Web Page	300
Working with Days	302
Customizing a Message for the Time of Day	303
Displaying Dates by Time Zone	304
Converting Military Time to A.M./P.M.	310
Creating a Countdown	312
Moving an Object in the Document	316
Date Methods	319
Chapter 14: Applied JavaScript	321
Using Sliding Menus	322
Adding Pull-Down Menus	325
A Slideshow with Captions	330
A Silly Name Generator	334
A Bar Graph Generator	340
Style Sheet Switcher	348
Chapter 15: Introducing Ajax	357
Ajax: Pinning It Down	359
Reading Server Data	363
Parsing Server Data	371
Refreshing Server Data	375
Previewing Links with Ajax	378
Auto-Completing Form Fields	382
Chapter 16: Ajax Toolkits	389
Dragging and Dropping Page Elements	391
Adding a Calendar to Your Page	397
Adding a 2-up Calendar to Your Page	401
Using the Container Utility	407
Adding Animation Effects	412
Implementing the Logger Control for Debugging	415
Chapter 17: Bookmarklets	419
Your First Bookmarklet	420
Changing a Page's Background Color	425
Changing a Page's Styles	426
Web-safe Colors	428
Word Lookups	430
Viewing Images	433
Displaying ISO Latin Characters	435
Converting RGB Values to Hex	436

Converting Values	438
A Bookmarklet Calculator	439
Validating Pages	441
Appendix A: JavaScript Genealogy and Reference	443
JavaScript Versions	444
ECMAScript	447
Browsers and JavaScript	449
Object Flowchart	450
The Big Object Table	455
Appendix B: JavaScript Reserved Words	467
Appendix C: Cascading Style Sheets Reference	471
Appendix D: Where to Learn More	479
Finding Help on the Web	480
Books	486
Index	489



INTRODUCTION

Welcome to JavaScript! Using this easy-to-learn programming language, you'll be able to add pizzazz to your Web pages and make them more useful for you and for your site's visitors. We've written this book as a painless introduction to JavaScript, so you don't have to be a geek or a nerd to write a script. Pocket protectors will not be necessary at any time. As a friend of ours says, "We're geeky, so you don't have to be!"

INTRODUCTION

We wrote this book for you

We figure that if you're interested in JavaScript, then you've already got some experience in creating HTML pages and Web sites, and you want to take the next step by adding some interactivity to your sites. We don't assume that you know anything about programming or scripting. We also don't assume that you are an HTML expert (though if you are, that's just fine). We do assume that you've got at least the basics of building Web pages down, and that you have some familiarity with common HTML, such as links, images, and forms.

We include some extra explanation of HTML in sidebars called "Just Enough HTML." You won't find these sidebars in every chapter, just the ones where we think you'll need a quick reference. Having this HTML information handy means you won't need two books open just to remember the syntax of a particular HTML attribute. We also think that you should have at least a nodding familiarity with CSS (Cascading Style Sheets), but we've included just about all the information you might need to get going if you haven't worked with CSS before.

If you already know something about programming, you should be aware that we don't take the same approach to JavaScript as you might have seen in other books. We don't delve deeply into JavaScript's syntax and structure, and we don't pretend that this book is a comprehensive language reference (though you'll find some valuable reference material in Appendix A, the color section in the back of the book). There are several other books on the market that do that job admirably, and we list them at the end of this book, in Appendix D. The difference between those books and this one is that instead of getting bogged down in formalism, we concentrate on showing you how to get useful

tasks done with JavaScript without a lot of extraneous information.

New in this edition, we've added an introduction to Ajax, a technique that uses JavaScript and other common Web technologies to add extra interactivity to Web pages, and to improve the user experience of your Web sites. We give you Ajax basics and then add practical examples that will allow you to Ajax-ify your sites without getting an advanced degree in Web programming.

How to use this book

Throughout the book, we've used some devices that should make it easier for you to work with both the book and with JavaScript itself.

In the step-by-step instructions that make up most of the book, we've used a special type style to denote either HTML or JavaScript code, like this:

```
<div align="center">
window.onload = initLinks;
```

You'll also notice that we show the HTML and the JavaScript in lowercase. We've done that because all of the scripts in this edition are compliant with the XHTML, 1.0 Transitional standard from the W3C, the World Wide Web Consortium. Whenever you see a quote mark in a JavaScript, it is always a straight quote (like ' or "), never curly quotes (aka "smart" quotes, like ' or "). Curly quotes will prevent your JavaScript from working, so make sure that you avoid them when you write scripts.

In the illustrations accompanying the step-by-step instructions, we've highlighted the part of the scripts that we're discussing in red, so you can quickly find what we're talking about. We often also highlight parts of the screen shots of Web browser windows in red, to indicate the most important part of the picture.

Because book pages are narrower than computer screens, some of the lines of JavaScript code are too long to fit on the page. When this happens, we've broken the line of code up into one or more segments, inserted this gray arrow → to indicate that it's a continued line, and indented the rest of the line. Here's an example of how we show long lines in scripts.

```
dtString = "Hey, just what are you  
→ doing up so late?";
```

You say browser, we say Kumbaya

We've made a big change in this edition; we have ended our support for browsers that are very old or that don't do a good job of supporting Web standards. Since our last edition, virtually all Web users have upgraded and are enjoying the benefits of modern browsers, ones that do a good-to-excellent job of supporting commonly accepted Web standards like XHTML, CSS2, and the Document Object Model. That covers Internet Explorer 6 or later; Firefox 1.0 or later; Netscape 7 or later; all versions of Safari; and Opera 7 or later.

If you do need to support older browsers, you'll need to use some older techniques. We covered those browser-specific techniques in previous editions, but we decided to make a clean break from the old ways of doing things this time around and took that information out of the printed book. We didn't want to lose that content entirely, so we are making it available as Chapter 18, "Working with Older Browsers," in the form of a downloadable PDF on the book's companion Web site. Just go to www.javascriptworld.com, click the Resources link, and you'll find the Chapter 18 file waiting for you. While you're there at the Web site, take a look around; chances are you'll find lots of other interesting things.

We (along with our crack team of testers) have tested our scripts in a wide variety of browsers, on several different operating systems, including Windows XP, Mac OS X, and Ubuntu Linux. Windows Vista had not arrived in final form as we went to press, and the public beta versions were still too subject to change, so we passed on using Vista for this edition.

We used the 600-pound gorilla of the browser world, Microsoft Internet Explorer 6 for Windows, to test virtually everything in the book, and did extensive testing with public beta versions of IE 7 (we went to press before the final version of IE 7 was released). We also tested the scripts with Firefox 1.5 for Windows, Mac, and Linux. We didn't leave out the Mac users, of course; all of the scripts were tested in Safari 2. That means that they should also work in other browsers derived from the WebKit engine that drives Safari (such as the Omni Group's OmniWeb), and on browsers based on the open-source KHTML rendering engine from which Safari got its start (such as Konqueror for Linux).

Don't type that code!

Some JavaScript books print the scripts, and expect you to type in the examples. We think that's way too retro for this day and age. It was tough enough for us to do all that typing, and there's no reason you should have to repeat that work. So we've prepared a companion Web site for this book, one that includes all of the scripts in the book, ready for you to just copy and paste into your own Web pages. The site also includes additional tips and scripts. If we discover any mistakes in the book that got through the editing process, we'll list the updates on the site, too. You can find our companion site at:

<http://www.javascriptworld.com/>

If for some reason you do plan to type in some script examples, you might find that the examples don't seem to work, because you don't have the supporting files that we used to create the examples. For example, in a task where an on-screen effect happens to an image, you'll need image files. No problem. We've put all of those files up on the book's Web site, nicely packaged for you to download. You'll find one downloadable file that contains all of the scripts, HTML files, and any media files we used. If you have any questions, please check the FAQ (Frequently Asked Questions) page on the companion Web site. It's clearly marked.

If you've read the FAQ, and your question isn't answered there, you can contact us via email at: js6@javascriptworld.com. We regret that because of the large volume of email that we get, we cannot, and will not, answer email about the book sent to our personal email addresses. We can only guarantee that messages sent to the js6@javascriptworld.com address will be answered.

Time to get started

One of the best things about JavaScript is that it's easy to start with a simple script that makes cool things happen on your Web page then add more complicated stuff as you need it. You don't have to learn a whole book's worth of information before you can start improving your Web pages. But by the time you're done with the book, you'll be adding advanced interactivity to your sites with Ajax.

Of course, every journey begins with the first step, and if you've read this far, your journey into JavaScript and Ajax has already begun. Thanks for joining us; please keep your hands and feet inside the moving vehicle. And please, no flash photography.

GETTING ACQUAINTED WITH JAVASCRIPT

1

GETTING ACQUAINTED WITH JAVASCRIPT

For Web site creators, the evolution of HTML has been a mixed blessing. In the early days of the World Wide Web, HTML was fairly simple, and it was easy to learn most everything you needed to learn about putting together Web pages.

As the Web grew, page designers' aspirations grew as well, and their demand for greater control over the look of the page forced HTML to change and become more complex.

Because the Web is a dynamic medium, page designers also wanted their pages to interact with the user, and it soon became obvious that HTML was insufficient to handle the demand. Netscape invented JavaScript as a way to control the browser and add pizzazz and interactivity to Web pages.

Since its creation, JavaScript has evolved quite a bit (although occasionally in different directions, depending on the browser). Later in this book, we'll discuss JavaScript's evolution in detail.

In this chapter, you'll learn what JavaScript is (and what it isn't); what it can do (and what it can't); and some of the basics of the JavaScript language; and you'll get an introduction to Ajax, the exciting combination of JavaScript and other technologies that is enabling the next wave of interactivity and creativity for Web sites.

1

What JavaScript Is

JavaScript is a programming language that you can use to add interactivity to your Web pages. But if you're not a programmer, don't panic; there are lots of JavaScripts available on the Web that you can copy and modify for your own use with a minimum of effort. In fact, standing on the shoulders of other programmers in this way is a great technique for getting comfortable with JavaScript.

To make it easier for you to get up and running with JavaScript, we have set up a Web site that supplements this book. We've included all the scripts in the book (so you don't have to type them in yourself!), as well as additional notes, addenda, and updates. You can find our site at <http://www.javascriptworld.com>.

You'll often see JavaScript referred to as a "scripting language," with the implication that it is somehow easier to script than to program. It's a distinction without a difference, in this case. A JavaScript script is a program that either is contained internally in an HTML page (the original method of scripting) or resides in an external file (the now-preferred method). On HTML pages, because it is enclosed in the `<script>` tag, the text of the script doesn't appear on the user's screen, and the Web browser knows to run the JavaScript program. The `<script>` tag is most often found within the `<head>` section of the HTML page, though you can, if you wish, have scripts in the `<body>` section. Internal scripts that write text to the screen or that write HTML are best put in the `<body>` section, as in **Script 1.1**. If you're unfamiliar with these HTML concepts and you need more information about HTML, we suggest that you check out Elizabeth Castro's *HTML, XHTML, and CSS, Sixth Edition: Visual QuickStart Guide*, also available from Peachpit Press.

Script 1.1 This very simple script types "Hello, Cleveland!" into the browser window.



```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
+ Transitional//EN">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Barely a script at all</title>
</head>
<body bgcolor="#FFFFFF">
<h1>
  <script language="JavaScript"
  + type="text/javascript">
    document.write("Hello, Cleveland!")
  </script>
</h1>
</body>
</html>

```

JavaScript Isn't Java

Despite the name, JavaScript and Java have almost nothing to do with one another. Java is a full-featured programming language developed and marketed by Sun Microsystems. With Java, a descendant of the C and C++ programming languages, programmers can create entire applications and control consumer electronic devices. Unlike other languages, Java holds out the promise of cross-platform compatibility; that is, a programmer should be able to write one Java program that can then run on any kind of machine, whether that machine is running Windows, Mac OS X, or any of the different flavors of Unix. In practice, Java hasn't always realized that dream, due in no small part to bickering between Sun and Microsoft as to the direction of the language. Microsoft got involved because it first wanted to integrate Java into Windows in its own way (a way that Sun said would make Java work one way on Windows, and another way on other machines, thereby defeating Java's main purpose); then Microsoft dropped Sun's Java from Windows altogether, after creating its own Java-like language, C#. After a flurry of lawsuits between the two companies (and a big settlement in favor of Sun), Microsoft removed its Java from Windows, and you can now install the latest version of Sun's Java for Windows (or Linux) at <http://www.java.com/getjava/>. Mac OS X comes with Java installed as part of the operating system.

JAVASCRIPT ISN'T JAVA

Chapter 1

Besides standalone applications, Java's main use on the *client side*, that is, in the user's browser, is to create *applets*, small programs that download over the Internet and run inside Web browsers. Because of Java's cross-platform nature, these applets should run identically on any Java-enabled browser. In recent years, we've seen many Java applets for browsers replaced by Macromedia Flash animations, which are generally easier to create than Java applets.

You embed Java applets in your Web pages using the `<object>` HTML tag, with additional information specifying the applet. When the browser sees the `<object>` tag, it downloads the Java applet from the server, and the applet then runs in the area of the screen specified in the tag (**Figure 1.1**).

To learn more about Java, we recommend *Java 2 for the World Wide Web: Visual QuickStart Guide* by Dori Smith.



Figure 1.1 This Java applet plays a mean game of checkers.

Where JavaScript Came From

If JavaScript isn't related to Java, then why do they have such similar names? It's another example of one of the computer industry's most annoying traits: the triumph of marketing over substance.

When Netscape added some basic scripting abilities to its Navigator Web browser, it originally called that scripting language LiveScript. Around the same time, Java was getting lots of press as the Next Big Thing In Computing. When Netscape revised Navigator to run Java applets in Navigator 2, it also renamed LiveScript to JavaScript, hoping that some of Java's glitter would rub off. The mere fact that JavaScript and Java were very different programming languages didn't stop Netscape's marketing geniuses, and ever since then, writers like us have made good money explaining that JavaScript and Java are very different things. Come to think of it, maybe we should be thanking those marketeers.

When Microsoft saw that JavaScript was becoming popular among Web developers, it realized that it would have to add some sort of scripting capabilities to Internet Explorer. It could have adopted JavaScript, but as is so often the case, Microsoft instead built its own language that works much like JavaScript but is not exactly the same. This Microsoft version of JavaScript is called JScript.

What JavaScript Can Do

There are many things that you can do with JavaScript to make your Web pages more interactive and provide your site's users with a better, more exciting experience. JavaScript lets you create an active user interface, giving the users feedback as they navigate your pages. For example, you've probably seen sites that have buttons that highlight as you move the mouse pointer over them. That's done with JavaScript, using a technique called a *rollover* (**Figure 1.2**).

You can use JavaScript to make sure that your users enter valid information in forms, which can save your business time and money. If your forms require calculations, you can do them in JavaScript on the user's machine without any server-side processing. That's a distinction you should know: programs that run on the user's machine are referred to as *client-side* programs; programs running on the server (including things called CGIs; more on them later) are called *server-side* programs.

With JavaScript, you have the ability to create custom HTML pages on the fly, depending on actions that the user takes. Let's say that you are running a travel site, and the user clicks Hawaii as a destination. You can have the latest Hawaii travel deals appear in a new window. JavaScript controls the browser, so you can open up new windows, display alert boxes, and put custom messages in the status bar of the browser window. Because JavaScript has a set of date and time features, you can generate clocks, calendars, and timestamp documents.

You can also deal with forms, set cookies, build HTML pages on the fly, and create Web-based applications.

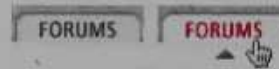


Figure 1.2 A rollover is an image that changes when you move the mouse pointer over it.

What JavaScript Can't Do

JavaScript is a *client-side* language; that is, it is designed to do its work on your machine, not on the server. Because of this, JavaScript has some limitations built-in, mostly for security reasons:

- ◆ JavaScript does not allow the reading or writing of files on client machines. That's a good thing, because you certainly don't want a Web page to be able to read files off of your hard disk, or be able to write viruses onto your disk, or be able to manipulate the files on your computer. The only exception is that JavaScript can write to the browser's cookie file, and even then there are limitations (for more information about cookies, see Chapter 10).
- ◆ JavaScript does not allow the writing of files on server machines. There are a number of ways in which this would be handy (such as storing page hit counts or filled-out form data), but JavaScript isn't allowed to do that. Instead, you'll need to have a program on your server to handle and store this data. This can be, for example, a CGI written in a language such as Perl or PHP, or a Java program.
- ◆ JavaScript cannot close a window that it hasn't opened. This is to avoid a situation where a site takes over your browser, closing windows from any other sites.
- ◆ JavaScript cannot read information from an opened Web page that came from another server. In other words, a Web page can't read any information from other open windows and find out what else a surfer visiting the site is up to.

What Is Ajax?

The short answer is that it's a way to create interactive Web applications. All right, now what does *that* mean? Let's think about things that you might want to do with a Web site. For example, you might want to go out to dinner with your spouse. (If you don't have a spouse, feel free to substitute "friend," "relative," or "pet" for "spouse" in the previous sentence. Though most people don't go to dinner with their pets. But we digress.) So you want to know how to get from your place to that fabulous new restaurant downtown that everyone's told you about. You decide to look up directions on a Web site that provides maps. You go to the site, type in the restaurant's address, and the site shows you a map with the restaurant marked. This particular site shows you the map, with a border that you can click if you want to change the map view (**Figure 1.3**). You click a border, wait five to ten seconds or so for the map to redraw, and, if you want to change the view again, repeat. It's a slow process and not very responsive to you. Wouldn't it be better to just click the map and drag it in the direction you want, and the map view moves as you drag?

That's the kind of dynamic responsiveness you can provide to your users with a Web application built with Ajax (**Figure 1.4**). There's almost no waiting; the user is in control; and it allows you to create Web-based applications with the kind of user experience found on a traditional desktop application. Plus, it makes it a lot faster and easier to find out how to get from home to that great dinner.



Figure 1.3 This MapQuest map takes a long time to respond to user clicks, because it needs a lot of refreshing from the server to make changes.



Figure 1.4 The Ajax-ified Google Maps allows a much more fluid and responsive user experience.

Ajax is shorthand for Asynchronous JavaScript and XML, and it is a term that was first coined in early 2005 by Jesse James Garrett, a Web developer and author. Strictly speaking, Ajax is just a small (although particularly popular) part of JavaScript. As commonly used, though, the term no longer refers to a technology by itself (like, say, Java or JavaScript).

In the larger scheme of things, what's generally referred to as Ajax is the combination of these technologies:

- ◆ XHTML
- ◆ CSS (Cascading Style Sheets)
- ◆ The DOM (Document Object Model) accessed using JavaScript
- ◆ XML, the format of the data being transferred between the server and the client
- ◆ XMLHttpRequest to retrieve data from the server

Whew. That's quite a list, especially if you don't have much experience with JavaScript or other Web programming. But you shouldn't worry; throughout this book, we'll introduce each of these technologies. By the time you get to the Ajax chapters, the pieces that make it up should be old hat.

The benefit to Ajax is that most of the processing for the application is happening within the user's browser, and requests to the server for data are usually short. So with Ajax, you can give users the kind of rich applications that depend on Web-based data, without the performance penalty of older approaches, which required that the server send back entire pages of HTML in response to user actions.

Chapter 1

Some companies have made huge investments in Ajax, notably Google, which has built several major Ajax applications, including Gmail (its Web-based email), Google Calendar, Google Suggest, and Google Maps. Another big supporter of Ajax is Yahoo!, which uses Ajax to enhance its personalized My Yahoo! portal, the Yahoo! front page, Yahoo! Mail, and more. Both companies have made interfaces to their Web applications public, so that people can use them to make interesting new applications. For example, many people have created *mashups* for Google Maps, which take a map and overlay interesting, useful, or just wacky information on the map, such as the location of all Japanese restaurants, or the filming locations for movies in the Los Angeles area.

✓ Tip

- For a nice listing of many Google Maps mashups, see Google Maps Mania (googlemapsmania.blogspot.com).



Figure 1.5 The cat object (this one's name is Pixel).

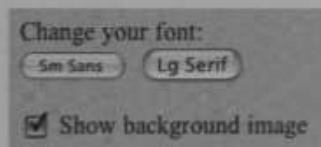


Figure 1.6 The buttons and check box are browser objects, which can be manipulated by JavaScript.

The Snap-Together Language

Here's another buzzword that we should get out of the way: JavaScript is an *object-oriented* language. So what does that mean?

Objects

First, let's think about objects. An *object* is some kind of a thing. A cat, a computer, and a bicycle are all objects (Figure 1.5) in the physical world. To JavaScript, there are objects it deals with in Web browsers, such as windows and forms, and the elements of the form, such as buttons and check boxes (Figure 1.6).

Because you can have more than one cat, or more than one window, it makes sense to give them names. While you could refer to your pets as Cat #1 and Cat #2, it's a bad idea for two reasons: first, it's easier to tell the cats apart if they have unique names, and second, it's just plain impolite. In the same way, all the examples in this book will give objects their own unique names.

✓ Tip

- Be aware that scripts you might see on the Internet will refer to objects like `window[0]` and `form[1]`. This is poor style for the reasons given above, and you'll find that it's much easier for you to keep track of the different objects in your scripts if you give them names instead of numbers.

Properties

Objects have *properties*. A cat has fur, the computer has a keyboard, and the bicycle has wheels. In the JavaScript world, a window has a title, and a form can have a check box.

Changing a property of an object modifies that object, and the same property name can be a part of completely different objects. Let's say that you have a property called *empty*. It's okay to use *empty* wherever it applies, so you could say that both the cat's tummy is empty and the cat's bowl is empty.

Note that the computer's keyboard and the bicycle's wheels aren't only properties; they are also objects in their own right, which can have their own properties. So objects can have sub-objects.

Methods

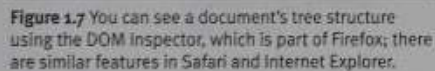
The things that objects can do are called *methods*. Cats purr, computers crash, and bicycles roll. JavaScript objects also have methods: buttons `click()`, windows `open()`, and text can be `selected()`. The parentheses signal that we're referring to a method, rather than a property.

✓ Tip

- It might help to think of objects and properties as nouns, and methods as verbs. The former are things, and the latter are actions that those things can do, or have done to them.

Putting the pieces together

You can put together objects, properties, and methods to get a better description of an object, or to describe a process. In JavaScript, these pieces are separated by periods (also known as dots, as in Internet addresses). This is called *dot syntax*. Here are some examples of objects and their properties written in this way:



And here are some examples of objects and methods written in dot syntax:

Introducing the Document Object Model

Each of the objects on the tree is also called a *node* of the tree. We can use JavaScript to modify any aspect of the tree, including adding, accessing, changing, and deleting nodes on the tree. Each object on the tree is a node. If the node contains an HTML tag, it's referred to as an *element node*. Otherwise, it's referred to as a *text node*. Of course, element nodes can contain text nodes. That's all you need to know about the DOM and nodes for now; you'll see more about them throughout the book, especially in Chapter 12.

Handling Events

Events are actions that the user performs while visiting your page. Submitting a form and moving a mouse over an image are two examples of events.

JavaScript deals with events using commands called *event handlers*. An action by the user on the page triggers an event handler in your script. The 12 most common JavaScript event handlers are listed in **Table 1.1**. We deal with other, more advanced event handlers in Chapter 9.

For example, let's say that our cat handles the event `onpetting` by performing the actions `purrr` and `stretch`.

In JavaScript, if the user clicks a button, the `onclick` event handler takes note of the action and perform whatever duties it was assigned.

When you write a script, you don't have to anticipate every possible action that the user might take, just the ones where you want something special to occur. For instance, your page will load just fine without an `onload` event handler. But you use the `onload` command if you want to trigger a script as soon as the page loads.

Table 1.1

Event Handlers

EVENT	WHAT IT HANDLES
<code>onabort</code>	The user aborted loading the page
<code>onblur</code>	The user left the object
<code>onchange</code>	The user changed the object
<code>onclick</code>	The user clicked an object
<code>onerror</code>	The script encountered an error
<code>onfocus</code>	The user made an object active
<code>onload</code>	The object finished loading
<code>onmouseover</code>	The cursor moved over an object
<code>onmouseout</code>	The cursor moved off an object
<code>onselect</code>	The user selected the contents of an object
<code>onsubmit</code>	The user submitted a form
<code>onunload</code>	The user left the page

Table 1.2

Value Types		
TYPE	DESCRIPTION	EXAMPLE
Number	Any numeric value	3.141592654
String	Characters inside quote marks	"Hello, world!"
Boolean	True or False	true
Null	Empty and meaningless	
Object	Any value associated with the object	
Function	Value returned by a function	

Table 1.3

Operators	
OPERATOR	WHAT IT DOES
$x + y$ (Numeric)	Adds x and y together
$x + y$ (String)	Concatenates x and y together
$x - y$	Subtracts y from x
$x * y$	Multiplies x and y together
x / y	Divides x by y
$x \% y$	Modulus of x and y (i.e., the remainder when x is divided by y)
$x++, ++x$	Adds one to x (same as $x = x + 1$)
$x--, --x$	Subtracts one from x (same as $x = x - 1$)
$-x$	Reverses the sign on x

Values and Variables

In JavaScript, a piece of information is a *value*. There are different kinds of values; the kind you're most familiar with are numbers. A *string* value is a word or words enclosed in quotes. Other kinds of JavaScript values are listed in **Table 1.2**.

Variables contain values. For example, the variable `myName` is assigned the string "Dori". Another way to write this is `myName="Dori"`. The equals sign can be read as "is set to." In other words, the variable `myName` now contains the value "Dori".

✓ Tips

- JavaScript is case sensitive. This means that `myname` is not the same as `myName`, and neither is the same as `MyName`.
- Variable names cannot contain spaces or other punctuation, or start with a digit. They also can't be one of the JavaScript reserved words. See Appendix B for a list of JavaScript reserved words.

Operators

Operators are the symbols used to work with variables. You're already familiar with operators from simple arithmetic; plus and minus are operators. See **Table 1.3** for the full list of operators.

✓ Tips

- While both `x++` and `++x` add one to x , they are not identical; the former increments x after the assignment is complete, and the latter before. For example, if x is 5, `y=x++` results in y set to 5 and x set to 6, while `y=++x` results in both x and y set to 6. The operator `--` (minus sign) works similarly.
- If you mix numeric and string values when adding two values together, the result is a string. For example, `cat + 5` results in `cat5`.

Assignments and Comparisons

When you put a value into a variable, you are assigning that value to the variable, and you use an assignment operator to do the job. For example, you use the equals operator to make an assignment, such as `hisName="Tom"`. There are a whole set of assignment operators as listed in **Table 1.4**.

Other than the equals sign, the other assignment operators serve as shortcuts for modifying the value of variables. For example, a shorter way to say `x=x+5` is to say `x+=5`. For the most part, we've used the longer version in this book for clarity's sake.

Comparisons

You'll often want to compare the value of one variable with another, or the value of a variable against a literal value (i.e., a value typed into the expression). For example, you might want to compare the value of the day of the week to "Tuesday", and you can do this by checking if `today'sDate=="Tuesday"`. A complete list of comparisons is in **Table 1.5**.

✓ Tip

- If you are comparing strings, be aware that "a" is greater than "A" and that "abracadabra" is less than "be".

Table 1.4

Assignments	
ASSIGNMENT	WHAT IT DOES
<code>x = y</code>	Sets x to the value of y
<code>x += y</code>	Same as <code>x = x + y</code>
<code>x -= y</code>	Same as <code>x = x - y</code>
<code>x *= y</code>	Same as <code>x = x * y</code>
<code>x /= y</code>	Same as <code>x = x / y</code>
<code>x %= y</code>	Same as <code>x = x % y</code>

Table 1.5

Comparisons	
COMPARISON	WHAT IT DOES
<code>x == y</code>	Returns true if x and y are equal
<code>x != y</code>	Returns true if x and y are not equal
<code>x > y</code>	Returns true if x is greater than y
<code>x >= y</code>	Returns true if x is greater than or equal to y
<code>x < y</code>	Returns true if x is less than y
<code>x <= y</code>	Returns true if x is less than or equal to y
<code>x && y</code>	Returns true if both x and y are true
<code>x y</code>	Returns true if either x or y is true
<code>!x</code>	Returns true if x is false

Writing JavaScript-Friendly HTML

Because you'll be using JavaScript to manipulate the objects within a document, you want to write your HTML in a way that can be easily used by your scripts. That basically means writing modern, standards-compliant XHTML and using CSS to separate the document's structure from its presentation.

When we say modern XHTML, we don't just mean documents that pass W3C validation using the Web tool at validator.w3.org. We also recommend thinking ahead to what you are likely to do with a page and adding appropriate tags and attributes that will make it easy to access objects with JavaScript. What sort of markup, you wonder? Glad you asked.

Structure, presentation, and behavior

CSS (Cascading Style Sheets) is a standard layout language for the Web that controls typography, colors, and the size and placement of elements and images. Your XHTML documents should have external style sheets defining the styles used within the document. Your JavaScript should also be in an external document, one that contains only JavaScript code.

When split up this way, your sites will contain three types of text files:

- ◆ XHTML: contains the content and structure of the page
- ◆ CSS: controls the appearance and presentation of the page
- ◆ JavaScript: controls the behavior of the page

When you do this, it becomes straightforward to make changes to your site—even changes with site-wide effects.

Divs and spans

If you're used to the classic style of HTML, where you threw everything into tables and played with spacer GIFs until it all fell into more-or-less the layout you wanted, some of this will be new to you—so here's a quick overview to bring you up to date.

XHTML contains two tags that are just starting to get the attention they deserve: `<div>` and `<span>`. They're used to break up your content into *semantic* chunks, that is, chunks that have a similar *meaning*. Things inside a single table cell or paragraph may or may not have anything in common, but the content within each `<div>` and `<span>` should.

Why use one over the other? A `<div>` is a block-level element, that is, there's a physical break between it and the elements above and below it. A `<span>` isn't block-level; instead, it's *inline*, so you can apply it to, for instance, a single phrase within a sentence.

We're not saying that you need to junk everything you've ever learned about HTML—far from it! But add these two tags to your toolkit of solutions, and you'll be surprised how often you use them.

Classes and ids

Inside your XHTML document, you'll mark up your content by breaking it down into those meaningful chunks. From there, you'll still need to identify those pieces of content where you want to change their presentation or behavior. For that, you'll primarily use two attributes: `class` and `id`. These attributes can be used by both CSS and JavaScript; a CSS style sheet uses those attributes as part of rules to define the appearance of a page, and the JavaScript file can use those attributes in code that affects the behavior of elements on the page.

- ◆ A *class* identifies an element that you may want to use more than once. For example, let's say that you're doing a page for a movie theater. You can define a class for the movie titles, specifying that the titles should be 14 pixel, bold, and dark blue.

```
.movieTitle {
    font: bold 14px;
    color: #000099;
}
```

You should then wrap each movie title on your page with a tag specifying the class of the title style, like so:

```
<p>We're currently showing
→ <span class="movieTitle">
→ The Aviator</span> and
→ <span class="movieTitle">
→ The Outlaw</span>.</p>
```

- ◆ An *id* identifies an element that is unique to that document. For example, if you only use the name of the movie theater once on your page, you can create a style rule using an *id* to define how the theater's name will look, like this:

```
#theaterName {
    font: bold 28px;
    color: #FF0000;
}
```

Then, when it's time to show the name of the theater, all you do is add that *id* attribute to the tag to get the effect:

```
<h1 id="theaterName">The Raven
→ Theater Presents:</h1>
```

What goes for CSS in the above examples also applies to JavaScript. Once we've assigned classes and *ids* to our *divs* and *spans* (and to any other elements as well), we can then modify those elements: not just their appearance with CSS, but also their behavior with JavaScript. And that's a topic that will take up the rest of this book.

✓ Tip

- Is that CSS above a little daunting? Chapter 11 should give you all the CSS help you'll need to know what's going on in this book.

What Tools to Use?

Since JavaScript is just plain text, you could use almost any kind of text editor. You could even use a word processor like Microsoft Word, though you would always have to make sure that Word saved the file as Text Only, instead of in its native file format. HTML, JavaScript, and CSS files must always be in plain text format so Web servers can understand them.

You're better off using a program that has plain text as its standard format. On Windows, many people get away with using Notepad (**Figure 1.8**). On the Mac, you can use TextEdit, though a favorite of professionals is BBEdit, by Bare Bones Software (**Figure 1.9**). On Unix machines, Emacs is one of the best text editors available. No matter what program you use, don't forget to save your plain text files with the extension .html or .htm so that things will go smoothly when you upload the file to a Web server.

You can also use some of the WYSIWYG (What You See Is What You Get) HTML editors available, such as Macromedia Dreamweaver, Adobe GoLive, or Microsoft FrontPage (on the PC; there's no Mac version of FrontPage). Just switch to their HTML Source modes, and script away.

✓ Tip

- If you're a Mac user, try TextWrangler, also from the Bare Bones folks (www.barebones.com). It's not as full-featured as BBEdit, but it has a big point in its favor: it's free.



Figure 1.8 Notepad on Windows XP.



Figure 1.9 BBEdit on Mac OS X.

15

INTRODUCING AJAX

The Web is always changing, and for Web and JavaScript developers the ground shifted under their feet beginning in early 2005. New and immediately popular Web applications appeared, some of them from Google, such as Gmail and Google Maps, and some from others, such as Flickr and the MyYahoo! portal. The common denominator of all of these new sites was that they acted more like desktop applications, with fast, responsive user interfaces. Instead of the traditional Web application, where the user clicked, waited some number of seconds for the server to respond and refresh the page, and then repeated the process, these new sites were more reactive, updating pages right away, providing superior interaction, and making for a better user experience.

There was something new (actually not so new, as we'll see) powering these sites, and it is called *Ajax*. You can use Ajax techniques to make your sites more responsive and attractive, which makes your site's users happier in the process. Best of all, you don't have to learn a completely new technology, because Ajax is made from building blocks that you already know (and that we've covered earlier in this book).

continues on next page

INTRODUCING AJAX

In this chapter, you'll learn how to request information from the server in the background and turn it into a form your Ajax application can use; automatically refresh the information from the server; build a cool previewing effect for objects on your page; and build an Ajax application that auto-completes form fields, just like a desktop application. Let's get to it.



Figure 15.1 This is the article that launched a zillion Ajax sites.

Ajax: Pinning It Down

One of the interesting things about Ajax is that there is some confusion and even a little disagreement as to what Ajax really is. We know that it is important and that it's very popular; heck, we even changed the name of this book from previous editions to hitch onto Ajax's popularity. So here is our take on what Ajax is and isn't, and what we mean when we use the term.

First, a little history. In February 2005, Jesse James Garrett, a founder of Adaptive Path (a Web interface and design shop in San Francisco), coined the term Ajax in an article on their site. He said Ajax was shorthand (but *not* an acronym) for "Asynchronous JavaScript and XML." You can read the article for yourself at www.adaptivepath.com/publications/essays/archives/000385.php (Figure 15.1).

According to Garrett, Ajax is not in itself a new technology; it's a technique that combines several long-standing Web technologies:

- ◆ Using XHTML and CSS for structure and presentation
- ◆ Displaying and manipulating pages using the Document Object Model
- ◆ Using the browser's XMLHttpRequest object to transfer data between the client and the server
- ◆ Using XML as the format for the data flowing between the client and server
- ◆ And finally, using JavaScript to dynamically display and interact with all of the above

An Ajax application places an intermediary between the user and the server. This *Ajax engine* (also known as the JavaScript part of a Web page) provides an interface to the user (in concert, of course, with XHTML and CSS), and if a user action doesn't require a request to the server (for example, displaying data that is already local), the Ajax engine responds. This allows the browser to react immediately to many user actions and makes the Web page act with the snappiness we've come to expect from our desktop programs. If the user action does require a server call, the Ajax engine performs it *asynchronously*, that is, without making the user wait for the server response; the user can continue to interact with the application, and the engine updates the page when the requested data arrives. The important part is that the user's actions don't come to a screeching halt while waiting for the server.

As the technique evolved, not all of the pieces had to be in place to call something an Ajax application, and this is where the confusion and disagreements set in. In fact, even the authors disagree about this:

Tom says, "I'm fine with just manipulating the page with the DOM, XHTML and CSS, and JavaScript and calling it Ajax. There are tons of effects that people are referring to as Ajax, and the whole look of modern sites has changed because of this approach. The change from the static Web to the dynamic Web page, which is sometimes called Web 2.0, owes its look and feel to the Ajax approach, whether or not there's a server call behind the scenes. Maybe calling it Ajax won't please the purists, but it's good enough for me."

Dori, who's the real JavaScript programmer in the family, says: "To call it Ajax, you need to use XMLHttpRequest and transfer some data between the client and server. Otherwise, what's so new about it?"

Dori's writing the code, so for the most part in this chapter we're sticking to her sensibilities as to what an Ajax application is and what it should do. But in the next chapter, we'll show you how to add some great (but still useful, not just flashy) Web 2.0-style eye candy to your sites.

Now, let's talk a little about what's not Ajax. Because you can do some cool visual effects on Web pages using Ajax, some people think that Ajax is *anything* you can do that looks good on a page, leading them to refer to things like interfaces made in Macromedia Flash as "Ajax." But just saying it doesn't make it so. Ajax is not about loading up your sites with cute user interface widgets and adding user interface tweaks that are cool but that change or break behaviors that people are used to with Web pages.

That leads us to problems with Ajax, and they can be significant. For example, to work correctly, an Ajax site needs to be running in a modern browser (for a list, see the back cover of this book). It also requires JavaScript. So what do you do about people using older browsers or who have turned JavaScript off? Or what about disabled users, or people who may be browsing your site with limited-capability handheld devices such as mobile phones or PDAs? The answer is that you must write your sites to degrade gracefully, meaning that users with less-capable browsers get a subset of your site's functionality, or, at the minimum, get a meaningful error message explaining why they can't use your site.

Chapter 15

Another potential problem with Ajax applications is that they may break the expected behavior of the browser's back button. With a static page, users expect that clicking the back button will move the browser to the last page it loaded. But because Ajax-enabled pages are dynamically updated, that might not be a valid expectation. There are solutions for the "back button problem," and before you dive wholeheartedly into Ajax, you should take the problem and its solutions into account.

Additionally, Ajax is not dependent on specific server-side technologies. There are a number of companies that are using the Ajax boom to try to sell their own server-side solutions, and that's what they're in business to do—but there's no reason why their products are required. So long as what's on the back end is something that your JavaScript can read (XML, ideally), you're fine. Just because the guys in the snappy suits (*cough* IBM *cough* Sun *cough*) want to hitch their buzzword-compliant products to Ajax's success in order to get you to buy doesn't mean that you have to fall for it.

Script 15.1 The HTML for the text and XML file request example.

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
+ Transitional//EN">
<html>
<head>
  <title>My First Ajax Script</title>
  <script src="script01.js"
    + type="text/javascript"
    + language="JavaScript">
  </script>
</head>
<body>
  <p><a id="makeTextRequest"
    + href="gAddress.txt">Request a text
    + file</a><br />
    <a id="makeXMLRequest"
    + href="us-states.xml">Request an XML
    + file</a></p>
  <div id="updateArea">&nbsp;</div>
</body>
</html>

```

Script 15.2 This JavaScript gets the files from the server.

```

window.onload = initAll;

var xhr = false;

function initAll() {
  document.getElementById("makeTextRequest").
    + onclick = getNewFile;
  document.getElementById("makeXMLRequest").
    + onclick = getNewFile;
}

function getNewFile() {
  makeRequest(this.href);
  return false;
}

function makeRequest(url) {
  if (window.XMLHttpRequest) {
    xhr = new XMLHttpRequest();
    (script continues on next page)

```

Reading Server Data

We begin our exploration of Ajax with the basics: using the XMLHttpRequest object to retrieve and display information from a server.

To get the job done, we'll use **Scripts 15.1** (HTML) and **15.2** (JavaScript). There are two possible files that can be read: the XML file that is shown in **Script 15.3** and the plain text file that is **Script 15.4**.

To request server data:

1. var xhr = false;

In Script 15.2, the xhr variable is one that you'll be seeing a lot of in this chapter. It's an XMLHttpRequest object (or it will be later, after it's initialized). At this point, we just need to create it outside any functions in order to make it globally available.

2. function initAll() {
 document.getElementById
 + ("makeTextRequest").onclick =
 + getNewFile;
 document.getElementById
 + ("makeXMLRequest").onclick =
 + getNewFile;
 }

When the page is first loaded, it knows to call the initAll() function. Here, we set two onclick handlers so that when a user clicks either of the links, the getNewFile() function is triggered.

continues on next page

```

3. function getNewFile() {
    makeRequest(this.href);
    return false;
}

```

Someone's clicked a link, so it's time to do something. Here, that something is to call `makeRequest()`—but that function needs to know which file was requested. Thankfully, we know that that information is tucked away in `this.href`, so we can pass it along. When we come back, we know we're done, so we return a value of `false`, telling the browser that no, we don't really want to load up a new Web page.

```

4. if (window.XMLHttpRequest) {
    xhr = new XMLHttpRequest();
}

```

Now, we're inside `makeRequest()`, and it's here that things get interesting. Modern browsers support a native `XMLHttpRequest` object as a property of `window`. So, we check to see if that property exists, and if it does, we create a new `XMLHttpRequest` object.

```

5. if (window.ActiveXObject) {
    try {
        xhr = new ActiveXObject
        ("Microsoft.XMLHTTP");
    }
    catch (e) { }
}

```

However, there's a browser that supports `XMLHttpRequest` that doesn't have a native version of the object, and that's Microsoft Internet Explorer (versions 5.5 and 6). In that case, we have to check to see if the browser supports ActiveX. If it does, we then check (using a try/catch error check) to see if we can create an `XMLHttpRequest` object based on ActiveX. If we can, great.

continues on page 366

Script 15.2 continued

```

}
else {
    if (window.ActiveXObject) {
        try {
            xhr = new ActiveXObject
            ("Microsoft.XMLHTTP");
        }
        catch (e) { }
    }
}

if (xhr) {
    xhr.onreadystatechange = showContents;
    xhr.open("GET", url, true);
    xhr.send(null);
}
else {
    document.getElementById("updateArea").
    .innerHTML = "Sorry, but I couldn't
    + create an XMLHttpRequest";
}
}

function showContents() {
    if (xhr.readyState == 4) {
        if (xhr.status == 200) {
            var outMsg = (xhr.responseXML &&
            + xhr.responseXML.contentType ==
            + "text/xml") ? xhr.responseXML.
            + getElementsByTagName("choices")
            + [0].textContent : xhr.responseText;
        }
        else {
            var outMsg = "There was a problem
            + with the request " + xhr.status;
        }
        document.getElementById("updateArea").
        .innerHTML = outMsg;
    }
}
}

```

Script 15.3 This is the XML file that is requested.

```
<?xml version="1.0"?>
<choices xml:lang="EN">
  <item><label>Alabama</label><value>AL</value></item>
  <item><label>Alaska</label><value>AK</value></item>
  <item><label>Arizona</label><value>AZ</value></item>
  <item><label>Arkansas</label><value>AR</value></item>
  <item><label>California</label><value>CA</value></item>
  <item><label>Colorado</label><value>CO</value></item>
  <item><label>Connecticut</label><value>CT</value></item>
  <item><label>Delaware</label><value>DE</value></item>
  <item><label>Florida</label><value>FL</value></item>
  <item><label>Georgia</label><value>GA</value></item>
  <item><label>Hawaii</label><value>HI</value></item>
  <item><label>Idaho</label><value>ID</value></item>
  <item><label>Illinois</label><value>IL</value></item>
  <item><label>Indiana</label><value>IN</value></item>
  <item><label>Iowa</label><value>IA</value></item>
  <item><label>Kansas</label><value>KS</value></item>
  <item><label>Kentucky</label><value>KY</value></item>
  <item><label>Louisiana</label><value>LA</value></item>
  <item><label>Maine</label><value>ME</value></item>
  <item><label>Maryland</label><value>MD</value></item>
  <item><label>Massachusetts</label><value>MA</value></item>
  <item><label>Michigan</label><value>MI</value></item>
  <item><label>Minnesota</label><value>MN</value></item>
  <item><label>Mississippi</label><value>MS</value></item>
  <item><label>Missouri</label><value>MO</value></item>
  <item><label>Montana</label><value>MT</value></item>
  <item><label>Nebraska</label><value>NE</value></item>
  <item><label>Nevada</label><value>NV</value></item>
  <item><label>New Hampshire</label><value>NH</value></item>
  <item><label>New Jersey</label><value>NJ</value></item>
  <item><label>New Mexico</label><value>NM</value></item>
  <item><label>New York</label><value>NY</value></item>
  <item><label>North Carolina</label><value>NC</value></item>
  <item><label>North Dakota</label><value>ND</value></item>
  <item><label>Ohio</label><value>OH</value></item>
  <item><label>Oklahoma</label><value>OK</value></item>
  <item><label>Oregon</label><value>OR</value></item>
  <item><label>Pennsylvania</label><value>PA</value></item>
  <item><label>Rhode Island</label><value>RI</value></item>
  <item><label>South Carolina</label><value>SC</value></item>
  <item><label>South Dakota</label><value>SD</value></item>
  <item><label>Tennessee</label><value>TN</value></item>
  <item><label>Texas</label><value>TX</value></item>
```

(script continues on next page)

```

6. if (xhr) {
    xhr.onreadystatechange =
        showContents;
    xhr.open("GET", url, true);
    xhr.send(null);
}

```

Either way, we should have a new `xhr` object, and if we do, we need to do something (in fact, three somethings) with it. Here are the three things that we always do with `xhr`:

- ▲ Set the `xhr`'s `onreadystatechange` event handler. Any time the `xhr.readyState` property changes its value, this handler is triggered.
- ▲ We call `open()` and pass in three parameters: an HTTP request method (e.g., "GET", "POST", or "HEAD"), a URL to a file on the server, and a Boolean telling the server if the request is asynchronous (that is, if we're sitting around waiting for it).
- ▲ And finally, we `send()` the request we just created. If we were requesting a POST, the parameters would be passed here.

```

7. else {
    document.getElementById
    ("updateArea").innerHTML =
    "Sorry, but I couldn't create an
    XMLHttpRequest";
}

```

If we end up here, we couldn't create an `XMLHttpRequest` for some reason, and there's nothing else that can be done.

Script 15.3 continued

```

<item><label>Utah</label><value>
  UT</value></item>
<item><label>Vermont</label><value>
  VT</value></item>
<item><label>Virginia</label><value>
  VA</value></item>
<item><label>Washington</label><value>
  WA</value></item>
<item><label>West Virginia</label><value>
  WV</value></item>
<item><label>Wisconsin</label><value>
  WI</value></item>
<item><label>Wyoming</label><value>
  WY</value></item>
</choices>

```

Script 15.4 The requested text file.

Script

Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived in liberty, and dedicated to the proposition that all men are created equal. Now we are engaged in a great civil war, testing whether that nation, on any nation so conceived and so dedicated, can long endure. We are met on a great battle-field of that war. We have come to dedicate a portion of that field, as a final resting place for those who here gave their lives that that nation might live. It is altogether fitting and proper that we should do this.

But, in a larger sense, we can not dedicate -- we can not consecrate -- we can not hallow -- this ground. The brave men, living and dead, who struggled here, have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember what we say here, but it can never forget what they did here. It is for us the living, rather, to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us -- that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion -- that we here highly resolve that these dead shall not have died in vain -- that this nation, under God, shall have a new birth of freedom -- and that government of the people, by the people, for the people, shall not perish from the earth.

```
8. if (xhr.readyState == 4) {
    if (xhr.status == 200) {
```

Now we're down in the `showContents()` function. The `readyState` property can have one of several values (see **Table 15.1**), and every time the server changes its value, the `showContents()` function is triggered. However, we don't actually want to do anything (at least not here) until the request is finished, so we start off by checking to see if `readyState` is 4. If it is, we're good to go, and we can check to see what the request returned.

The first thing to check is the request's `status`, which will be a result code returned by the server (servers routinely return these codes behind the scenes for every file served, although browsers only show them to you if there's an error). A status code of 200 means that everything's fine. The status here is the same status that is returned by any server call; for instance, if you ask for a file that doesn't exist you'll get a 404 error from the Web server.

continues on next page

Table 15.1**readyState Property Values**

VALUE	WHAT IT MEANS
0	Uninitialized; object contains no data
1	Loading; object is currently loading its data
2	Loaded; object has finished loading its data
3	Interactive; user may interact with the object even though it is not fully loaded
4	Complete; object has finished initializing

```

9. var outMsg = (xhr.responseXML &&
  → xhr.responseXML.contentType
  → == "text/xml") ? xhr.responseXML.
  → getElementsByTagName("choices")[0].
  → textContent : xhr.responseText;

```

If we're here, that means that everything is fine, and we want to look at what the server actually gave us. There were two different types of files we could be reading, so we need to check what type of data we got back. The `responseXML` property contains the data if it's XML. If the `contentType` property (which can also be referred to as the MIME type) contains "text/xml", then we know that we've got a properly formatted DOM object back, and we can use commands we've seen before (such as `getElementsByTagName()`) to traverse its nodes. But here, we're just trying to see if it worked, so we'll take everything and dump it all into `outMsg`.

If what we got back isn't valid XML, then it's our text file. In that case, we want to put `xhr.responseText` property into `outMsg`.

```

10. else {
    var outMsg = "There was a problem
    → with the request " + xhr.status;
  }

```

If what we got back had a status other than 200, we've got a problem, so we set `outMsg` to say that and append the status error so we can try to figure out what the problem is.

```

11. document.getElementById
  → ("updateArea").innerHTML = outMsg;

```

And finally, we take `outMsg` and dump it onto the screen, as shown in **Figure 15.2**.

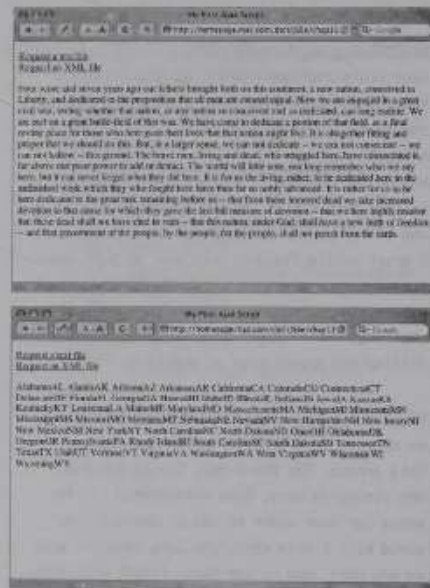


Figure 15.2 By clicking the appropriate link, you can fetch either a text file of the Gettysburg Address (top) or an XML file of U.S. states and their abbreviations (bottom).

✓ Tips

- Because of the way that Ajax works, when you are doing your development and testing, the files that you're reading must reside on a server; they can't just be local files.
- Back in step 5, we said that IE 5.5 and 6 used an ActiveX control to create the XMLHttpRequest object. Thankfully, IE7 has a native object so that's no longer required. However, this means that you always have to check for the existence of a native object first—if you check for `window.ActiveXObject` first, that will be true for IE7, and then you'll be going down the wrong path. A considerable amount of older, pre-IE7 Ajax code has this problem.
- If it matters deeply to your code which version of Microsoft's ActiveX object you actually get, here's a code snippet for you to use instead:

```
if (window.ActiveXObject) {
    try {
        xhr = new ActiveXObject
            ("Msxml2.XMLHTTP");
    }
    catch (e) {
        try {
            xhr = new ActiveXObject
                ("Microsoft.XMLHTTP");
        }
        catch (e) { }
    }
}
```

This approach attempts to use the IE6 version (`Msxml2.XMLHTTP`) of the XMLHttpRequest object first and only falls back to the older version if it can't find it. However, the `Microsoft.XMLHTTP` version should always give you the latest version available on the PC, so we'll just be using that in this chapter—because eventually, the older code will be going away.

✓ Tips

- One drawback of Ajax calls is that they can be cached; that is, it looks like your application is contacting the server and getting new data, but it's really just looking at stuff it read previously. If that's the case, setting the headers of the request can help. Adding one or more of these can help force recalcitrant servers to fork over the goods:

```
xhr.setRequestHeader  
→ ("If-Modified-Since",  
→ "Wed, 15 Jan 1995 01:00:00 GMT");  
xhr.setRequestHeader  
→ ("Cache-Control", "no-cache");  
xhr.setRequestHeader  
→ ("Cache-Control",  
→ "must-revalidate");  
xhr.setRequestHeader  
→ ("Cache-Control", "no-store");  
xhr.setRequestHeader  
→ ("Pragma", "no-cache");  
xhr.setRequestHeader  
→ ("Expires", "0");
```

- You can force the call to return XML data by overriding the MIME type:
`xhr.overrideMimeType("text/xml");`
However, this may cause problems with certain browsers and configurations, so use it with care.

Script 15.5 The additional JavaScript in this script allows you to parse the data you previously requested.

```

window.onload = initAll;
var xhr = false;
function initAll() {
    if (window.XMLHttpRequest) {
        xhr = new XMLHttpRequest();
    }
    else {
        if (window.ActiveXObject) {
            try {
                xhr = new ActiveXObject
                    ("Microsoft.XMLHTTP");
            }
            catch (e) { }
        }
    }

    if (xhr) {
        xhr.onreadystatechange = showPictures;
        xhr.open("GET", "flickrfeed.xml", true);
        xhr.send(null);
    }
    else {
        alert("Sorry, but I couldn't create an
            XMLHttpRequest");
    }
}

function showPictures() {
    var tempDiv = document.createElement
        ("div");
    var pageDiv = document.getElementById
        ("pictureBar");

    if (xhr.readyState == 4) {
        if (xhr.status == 200) {
            tempDiv.innerHTML = xhr.responseText;
            var allLinks = tempDiv.
                getElementsByTagName("a");

            for (var i=1; i<allLinks.length;
                i+=2) {

                (script continues on next page)
            }
        }
    }
}

```

Parsing Server Data

Now that we have data from the server, we need to find the exact information we need and make sure that it is in a format that our Ajax application can use. To do that, we'll first need to examine the information. Because the data is a nicely structured XML document, our script walks the XML document tree to find and extract the particular data we need and store it in variables. Then, if needed, the script can reformat the data for later use.

The HTML is unchanged from the last example, so all we need to do is add more code in the JavaScript file. The result is **Script 15.5**. For this task, the XML file is data about photographs stored on Flickr; a portion of the XML can be seen in **Script 15.6**.

To parse information from the server:

1. `xhr.onreadystatechange = showPictures;`
`xhr.open("GET", "flickrfeed.xml", true);`

Every time `readyState` changes, we want to call the `showPictures()` function. The file name we want to read off the server is `flickrfeed.xml`. Both those values are set here.

2. `var tempDiv = document.createElement("div");`
`var pageDiv = document.
 getElementById("pictureBar");`

Down in `showPictures()` is where the real work is done. We start by creating variables to store two elements: `tempDiv`, a temporary div placeholder, and `pageDiv`, a reference to the `pictureBar` div on the page.

continues on next page

Chapter 15

```

3. tempDiv.innerHTML =
   → xhr.responseText;
   var allLinks = tempDiv.
   → getElementsByTagName("a");

```

The response back from the server contained XML, but for the sake of convenience here, we're taking the data from `responseText` (that is, the text version) and throwing it into `tempDiv`'s `innerHTML` property.

Take a look at the XML in Script 15.6, and you'll see that there's a lot of stuff there that we don't care about at all—in fact, all we want is what's in the `<a>` tags (and really, only half of those). So, we start to narrow down to just what we want by creating the `allLinks` array, which will contain the contents of all the `<a>` nodes in the document.

```

4. for (var i=1; i<allLinks.length;
   → i+=2) {

```

As previously mentioned, we only want half the `<a>` nodes, so we winnow out the ones here that we don't want. In a file with information about 20 photos, there will be 20 `<entry>` nodes, each of which contains two links. Each `<entry>` node contains the photographer's name (linked to their Flickr page), followed by a thumbnail image which is linked to the full-size version. We want just the latter, so we can go through the loop starting with 1 (rather than 0) and increment `i` by 2 each go-round.

The end result: we'll get links 1, 3, 5, and so on, each of which is just the thumbnail image inside a link.

continues on page 374

Script 15.5 continued

```

Script
pageDiv.appendChild(allLinks[i].
→ cloneNode(true));
}
}
else {
  alert("There was a problem with the
→ request " + xhr.status);
}
}
}
}

```

Script 15.6 This is an edited and shortened version of the XML file that Flickr provides; the original was approximately 500 lines in length!

```

Script
<?xml version="1.0" encoding="utf-8" standalone=
→ "yes"?>
<feed xmlns="http://www.w3.org/2005/Atom" xmlns:
→ dc="http://purl.org/dc/elements/1.1/">
  <title>Dori Smith's Photos</title>
  <link rel="self" href="http://www.flickr.
→ com/services/feeds/photos_public.gne?id=
→ 23922109@N00" />
  <link rel="alternate" type="text/html"
→ href="http://www.flickr.com/photos/
→ dorismith/" />
  <id>tag:flickr.com,2005:/photos/public/
→ 116078</id>
  <icon>http://static.flickr.com/5/
→ buddyicons/23922109@N00.jpg?1113973282
→ </icon>
  <subtitle>A feed of Dori Smith's Photos
→ </subtitle>
  <updated>2006-03-22T20:12:44Z</updated>
  <generator uri="http://www.flickr.com/"
→ ">Flickr</generator>
  <entry>
    <title>Mash note</title>
    <link rel="alternate" type="text/html"
→ href="http://www.flickr.com/photos/
→ dorismith/116463569/" />

```

(script continues on next page)

Script 15.6 continued

```

<id>tag:flickr.com,2005:/photo/116463569</id>
<published>2006-03-22T20:12:44Z</published>
<updated>2006-03-22T20:12:44Z</updated>
<dc:date.Taken>2006-03-22T12:12:44-08:00</dc:date.Taken>
<content type="html"><p><a href="http://www.flickr.com/people/dorismith/">Dori Smith</a>
  posted a photo:</p>
<p><a href="http://www.flickr.com/photos/dorismith/116463569/" title="Mash note"></a></p></content>
<author>
  <name>Dori Smith</name>
  <uri>http://www.flickr.com/people/dorismith/</uri>
</author>
<category term="dorismith" scheme="http://www.flickr.com/photos/tags/" />
<category term="ncmug" scheme="http://www.flickr.com/photos/tags/" />
</entry>
<entry>
  <title>My odometer reading</title>
  <link rel="alternate" type="text/html" href="http://www.flickr.com/photos/dorismith/69558528/" />
  <id>tag:flickr.com,2005:/photo/69558528</id>
  <published>2005-12-03T03:48:40Z</published>
  <updated>2005-12-03T03:48:40Z</updated>
  <dc:date.Taken>2005-11-07T13:04:54-08:00</dc:date.Taken>
  <content type="html"><p><a href="http://www.flickr.com/people/dorismith/">Dori Smith</a> posted
    a photo:</p>
<p><a href="http://www.flickr.com/photos/dorismith/69558528/" title="My odometer reading"><img src=
  "http://static.flickr.com/20/69558528_d6b9bb4e08_s.jpg" width="75" height="75" alt="My odometer
  reading" style="border: 5px solid #ddd;" /></a></p>
<p>Why yes, I am a geek.</p></content>
  <author>
    <name>Dori Smith</name>
    <uri>http://www.flickr.com/people/dorismith/</uri>
  </author>
  <category term="32768" scheme="http://www.flickr.com/photos/tags/" />
  <category term="dori" scheme="http://www.flickr.com/photos/tags/" />
  <category term="geek" scheme="http://www.flickr.com/photos/tags/" />
  <category term="dorismith" scheme="http://www.flickr.com/photos/tags/" />
</entry>
</feed>

```

5. `pageDiv.appendChild(allLinks[i].cloneNode(true));`

Inside the loop, we take the node that we want, clone it, and then append it onto the HTML page's `pageDiv`. The end result is as we see it in **Figure 15.3**, where every thumbnail image on the page is a link back to the full-size version.

We need to use `cloneNode` and not simply `append` `allLinks[i]` because using `append` actually moves the nodes from one to another—instead, we want to duplicate what we've already got. The value `true` passed as a parameter tells JavaScript to clone not just this node, but also any child nodes of this node, and that's exactly what we want to do to get the `<img>` node along with the `<a>`.

✓ Tips

- While you can't read a data file that's stored on another server (see the "Getting Your Data" sidebar for more about why that's the case), you can always have your HTML file load information from another server. Here, your Web page, no matter where it is, is able to display images from flickr.com's servers.
- One of the best things about the new crop of Web 2.0 dot-coms is that they understand that people want access to data—and not just their own data, but other people's data (when they've agreed to make it public) as well. For instance, it's possible to search Flickr for all the photographs containing the tags "Hawaii" and "sunset," and then get the result as an XML file. Combine that with this script or the next, and you'll always have new and lovely photos on your page.



Figure 15.3 These thumbnail images were read from Flickr.

Getting Your Data

One of the things people want to do when they first hear about Ajax is write JavaScript that reads in all kinds of XML files (including RSS and Atom feeds), mash them up, and then put the result on their own Web page.

The bad news: it doesn't quite work that way—a script can only read a file that comes from the same server as the one that the script is on itself. If you think about it for a while, you'll start to figure out why; after all, if a script could read anything, then that would open up all kinds of possible security scams and fake sites.

The sort-of good news: you can have a program on your server that goes out periodically, grabs an XML file, and then stores it locally. Once you've done that, your Ajax application will have no problems reading it. In this example and the next, it's assumed that you've got something running that grabs the Flickr data file of your choice periodically and saves it on your server. How to do that, though, is beyond the scope of this book.

Script 15.7 Use this script to automatically refresh server information.

```

window.onload = initAll;

var xhr = false;

function initAll() {
    if (window.XMLHttpRequest) {
        xhr = new XMLHttpRequest();
    }
    else {
        if (window.ActiveXObject) {
            try {
                xhr = new ActiveXObject(
                    "Microsoft.XMLHTTP");
            }
            catch (e) {}
        }
    }

    if (xhr) {
        getPix();
    }
    else {
        alert("Sorry, but I couldn't create an
            XMLHttpRequest");
    }
}

function getPix() {
    xhr.open("GET", "flickrfeed.xml", true);
    xhr.onreadystatechange = showPictures;
    xhr.send(null);

    setTimeout("getPix()", 5 * 1000);
}

function showPictures() {
    var tempDiv = document.createElement(
        "div");
    var tempDiv2 = document.createElement(
        "div");

    if (xhr.readyState == 4) {
        (script continues on next page)
    }
}

```

Refreshing Server Data

Our Ajax application has fetched information from the server and then parsed the data and acted upon it. Now we'll show you how to make the application retrieve a new version of the data from the server, which automatically refreshes the page. **Script 15.7** contains the necessary JavaScript.

To refresh server information:

1. function getPix() {
 xhr.open("GET", "flickrfeed.xml",
 true);
 xhr.onreadystatechange =
 showPictures;
 xhr.send(null);

 setTimeout("getPix()", 5 * 1000);
 }
2. for (var i=1; i<allLinks.length;
 i+=2) {
 tempDiv2.appendChild(allLinks[i].
 cloneNode(true));
 }

This is almost the same loop that's in the previous task, with one difference: instead of appending the nodes to the Web page, it adds them to another temporary div.

continues on next page

```

3. allLinks = tempDiv2.
   → getElementsByTagName("a");
   var randomImg = Math.floor
   → (Math.random() * allLinks.length);
   document.getElementById
   → ("pictureBar").innerHTML =
   → allLinks[randomImg].innerHTML;

```

When the loop completes, we want to figure out one random image out of all of them to display. We start by creating another array of links (now with just the linked photographs instead of all the links). We then calculate a random number between zero and one less than the number of images, using `Math.random()` and `Math.floor()` as we did back in Chapter 4 in "Displaying a Random Image." And finally, we use that random number as an index into the `allLinks` array to grab the one image, and we put that into our Web page (Figure 15.4).

Script 15.7 continued

```

if (xhr.status == 200) {
    tempDiv.innerHTML = xhr.responseText;
    var allLinks = tempDiv.
    → getElementsByTagName("a");

    for (var i=1; i<allLinks.length;
    → i+=2) {
        tempDiv2.appendChild(allLinks[i].
        → cloneNode(true));
    }

    allLinks = tempDiv2.
    → getElementsByTagName("a");
    var randomImg = Math.floor
    → (Math.random() * allLinks.length);
    document.getElementById
    → ("pictureBar").innerHTML =
    → allLinks[randomImg].innerHTML;
}
else {
    alert("There was a problem with the
    → request." + xhr.status);
}
}
}

```

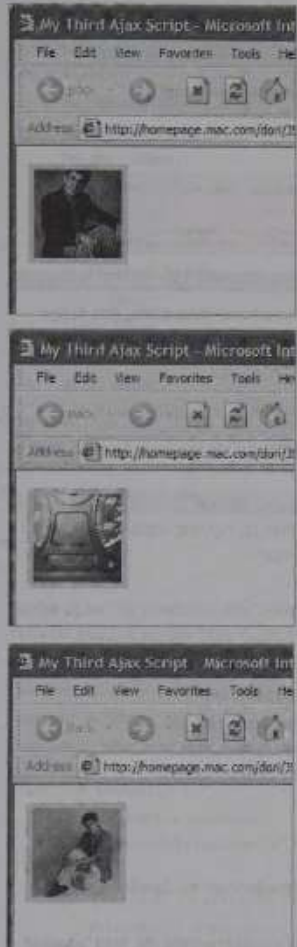


Figure 15.4 The script fetches one image after another.

✓ Tips

- You might wonder why this script bothers to read from the same XML file every time—after all, if the file isn't changing, why not just keep the data in variables after the first time through? If you keep in mind the technique referred to in the previous sidebar ("Getting Your Data"), you'll then realize that the XML file could be changing at any point. Say your server-side program grabs a new version of the XML file every few minutes—why should anyone have to wait to see the latest pictures? This way, your site's visitors always get the latest possible version.
- If you take the approach just mentioned, you're likely to run into the Ajax drawback covered earlier in this chapter: caching. Different browsers (and different versions, and different platforms) all have their own unique caching peculiarities, most of which are solved by modifying the headers as discussed earlier. Another solution many recommend is to change the GET to a POST. But here's what we've found that works: swap the order of the `open()` and `onreadystatechange`, as shown above in step 1.

Previewing Links with Ajax

There's a handy and great-looking visual effect that many sites are using now, where, when you hover the mouse pointer over a link, the first few lines of the page that is the link's destination appear in a floating window under the cursor (**Figure 15.5**). This turns out to be a fairly easy-to-create Ajax application. You'll find the HTML in **Script 15.8**, the CSS in **Script 15.9**, and the JavaScript in **Script 15.10**.

To use Ajax to preview links:

1.

```
var allLinks = document.  
  → getElementsByTagName("a");  
for (var i=0; i< allLinks.length;  
  → i++) {  
    allLinks[i].onmouseover =  
    → showPreview;  
}
```

Here's our `initAll()` function, which simply plows through all the links on the page and adds an `onmouseover` event handler to each. This event handler will (as you'll see, below) read the destination page and display a preview for the (possible) visitor.

2.

```
function showPreview(evt) {  
    getPreview(evt);  
    return false;  
}
```

The `showPreview()` event handler function passes the event triggering the function call to `getPreview()`. That's where the real work will be done.



Figure 15.5 When you hover over a link, this script reads the HTML file on the server and gives you an overlay containing a preview of the first few lines of the file.

Script 15.8 This HTML builds the page for the preview example.

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0  
→ Transitional//EN">  
<html>  
<head>  
    <title>My Fourth Ajax Script</title>  
    <link rel="stylesheet" rev="stylesheet"  
    → href="script04.css" />  
    <script src="script04.js"  
    → type="text/javascript"  
    → language="JavaScript">  
    </script>  
</head>  
<body>  
<h2>A Gentle Introduction to JavaScript</h2>  
<ul>  
    <li><a href="jsintro/2000-08.html">August  
    → column</a></li>  
    <li><a href="jsintro/2000-09.html">  
    → September column</a></li>  
    <li><a href="jsintro/2000-10.html">  
    → October column</a></li>  
    <li><a href="jsintro/2000-11.html">  
    → November column</a></li>  
</ul>  
<div id="previewWin"> </div>  
</body>  
</html>
```

Script 15.9 This CSS styles the preview pop-up.

```

#previewWin {
    background-color: #FF9;
    width: 400px;
    height: 100px;
    font: .8em arial, helvetica, sans-serif;
    padding: 5px;
    position: absolute;
    visibility: hidden;
    top: 10px;
    left: 10px;
    border: 1px #CC0 solid;
    clip: auto;
    overflow: hidden;
}
#previewWin h1, #previewWin h2 {
    font-size: 1.0em;
}

```

Script 15.10 The JavaScript that allows the server request and the appearance of the pop-up.

```

window.onload = initAll;
var xhr = false;
var xPos, yPos;

function initAll() {
    var allLinks = document.
    →getElementsByName("a");

    for (var i=0; i<allLinks.length; i++) {
        allLinks[i].onmouseover = showPreview;
    }
}

function showPreview(evt) {
    getPreview(evt);
    return false;
}

function hidePreview() {
    (script continues on next page)
}

```

```

3. function hidePreview() {
    document.getElementById
    → ("previewWin").style.visibility
    += "hidden";
}

```

If we have a `showPreview()` function, we need a `hidePreview()`, right? Here's ours, and all it does is set the preview window back to its hidden level of visibility.

```

4. if (evt) {
    var url = evt.target;
}
else {
    evt = window.event;
    var url = evt.srcElement;
}
xPos = evt.clientX;
yPos = evt.clientY;

```

Here in `getPreview()`, the first thing we need to do is figure out what file we want to read, and that's done by looking at the properties of the event. Depending on which browser your visitor is using, the URL is in either `evt.target` or `window.event.srcElement`. Once we've got that, we can grab the x and y positions of the mouse for later use.

```

5. var prevWin = document.
    →getElementById("previewWin");

```

```

if (xhr.readyState == 4) {
    Having used Ajax to read the file,
    we're now down in the showContents()
    function. We store the previewWin ele-
    ment for later use in prevWin, and when
    xhr.readyState is 4, it's time to show off.
}

```

continues on next page

Chapter 15

```

6. prevWin.innerHTML = (xhr.status ==
   - 200) ? xhr.responseText :
   - "There was a problem with the
   - request " + xhr.status;
prevWin.style.top = parseInt(yPos)+
   - 2 + "px";
prevWin.style.left =
   - parseInt(xPos)+2 + "px";
prevWin.style.visibility =
   - "visible";
prevWin.onmouseout = hidePreview;

```

If everything's fine, then `xhr.status` is 200 and the data we want to put into `prevWin.innerHTML` is in `xhr.responseText`. If not, we put the error message there instead.

Once that's done, it's simply a matter of figuring out where to place the preview window, and that's where those `x` and `y` mouse coordinates come in handy. It's a pop-up, so we put it just below and to the right (2 pixels over and 2 down) of the cursor position that triggered this call. Lastly, we set `prevWin` to be visible, and we let JavaScript know that `prevWin` should be hidden when the cursor moves off the preview.

Script 15.10 continued

```

document.getElementById("previewWin").
  - style.visibility = "hidden";
}

function getPreview(evt) {
  if (evt) {
    var url = evt.target;
  }
  else {
    evt = window.event;
    var url = evt.srcElement;
  }
  xPos = evt.clientX;
  yPos = evt.clientY;

  if (window.XMLHttpRequest) {
    xhr = new XMLHttpRequest();
  }
  else {
    if (window.ActiveXObject) {
      try {
        xhr = new ActiveXObject(
          - ("Microsoft.XMLHTTP"));
      }
      catch (e) { }
    }
  }

  if (xhr) {
    xhr.onreadystatechange = showContents;
    xhr.open("GET", url, true);
    xhr.send(null);
  }
  else {
    alert("Sorry, but I couldn't create an
      - XMLHttpRequest");
  }
}

function showContents() {
  (script continues on next page)

```

Script 15.10 continued

```

var prevWin = document.getElementById
+ ("previewWin");

if (xhr.readyState == 4) {
    prevWin.innerHTML = (xhr.status == 200)
    + ? xhr.responseText : "There was a
    + problem with the request " +
    + xhr.status;
    prevWin.style.top = parseInt(yPos)+2 +
    + "px";
    prevWin.style.left = parseInt(xPos)+2 +
    + "px";
    prevWin.style.visibility = "visible";
    prevWin.onmouseout = hidePreview;
}
}

```

✓ Tips

- The data being read is in HTML format. Putting `xhr.responseText` into `innerHTML` tells the browser that when the preview window displays, it should interpret the HTML as, well, HTML. If you wanted something else to display (say, for instance, that you wanted to see the actual source of the page), you could modify what's in `innerHTML` before displaying the preview.
- Ajax requires that the file being read reside on the same server—but it doesn't require that it be in the same directory. If the page you're reading in is in a different directory, and the page contains relative links, then those links will not work. If your pages refer to a particular CSS file, or images, or JavaScript, you won't be able to preview those particular parts of the file. The same solution applies here as well: modify `prevWin.innerHTML` before displaying it.

Auto-Completing Form Fields

A first-rate way to help your site's visitors is to lessen the drudgery of data entry into fields. Helping them fill out forms that have a large number of choices saves them time and effort, and additionally helps provide your site with valid data.

For this example, **Script 15.11** (HTML), **Script 15.12** (CSS), and **Script 15.13** (JavaScript) automatically show a list of U.S. states that match the letters the user types into a form field (**Figure 15.6**). As the user continues typing, the list shrinks until there is only one state left; this is then automatically put into the entry field, and the list goes away.

To build auto-completing form fields:

1. Please enter your state:

→ autocomplete="off" />

<div id="popups"> </div>
Here's the bit of HTML that we care about. The one tricky (and non-standards compliant) part: that autocomplete attribute. It tells browsers not to do any auto-completion on this field, as we'll be handling it with the script. While it isn't part of any W3C recommendations, autocomplete, like XMLHttpRequest itself, has excellent cross-browser support.

2. document.getElementById
→ ("searchField").onkeyup =
→ searchSuggest;
In order to grab and process each key-stroke, we need an event handler, and here's ours, set in initAll().

Script 15.11: This simple HTML provides the form field that will be auto-completed.

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
→ Transitional//EN"
<html>
<head>
  <title>My Fifth Ajax Script</title>
  <link rel="stylesheet" rev="stylesheet"
→ href="script05.css" />
  <script src="script05.js"
→ type="text/javascript"
  <language="JavaScript">
  </script>
</head>
<body>
  <form action="#">
    Please enter your state:<br />
    <input type="text" id="searchField"
→ autocomplete="off" /><br />
    <div id="popups"> </div>
  </form>
</body>
</html>

```

Script 15.12 The CSS here styles the search field and the pop-up menu.

```

body, #searchfield {
  font: 1.2em arial, helvetica, sans-serif;
}

.suggestions {
  background-color: #FFF;
  padding: 2px 6px;
  border: 1px solid #000;
}

.suggestions:hover {
  background-color: #69F;
}

#popups {
  position: absolute;
}

#searchField.error {
  background-color: #FFC;
}

```

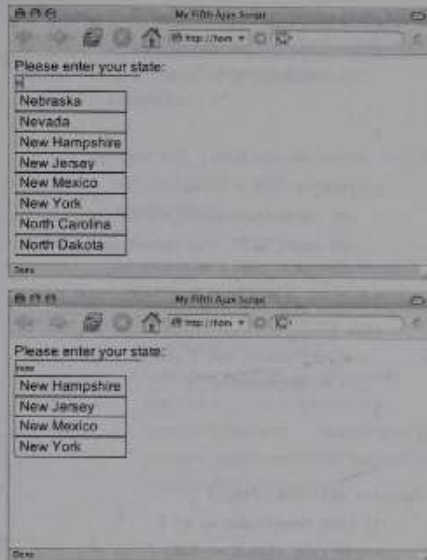


Figure 15.6 As you type, the number of possible choices narrows.

Script 15.13 This JavaScript handles the server request and the pop-up display.

```

<script>
window.onload = initAll;
var xhr = false;
var statesArray = new Array();

function initAll() {
    document.getElementById("searchField").
        .onkeyup = searchSuggest;

    if (window.XMLHttpRequest) {
        xhr = new XMLHttpRequest();
    }
    else {
        if (window.ActiveXObject) {
            try {
                xhr = new ActiveXObject
                    ("Microsoft.XMLHTTP");
            }
        }
    }
}

```

(script continues on next page)

```

3. xhr.onreadystatechange =
    → setStatesArray;
   xhr.open("GET", "us-states.xml",
    → true);
   xhr.send(null);

```

Unlike those photographs earlier in this chapter, the names of the United States aren't likely to change. We can read the XML file (Script 15.3) in once, initialize our array, and safely assume that our list will still be valid at the end of this session.

```

4. if (xhr.responseXML) {
    var allStates = xhr.responseXML.
        .getElementsByTagName("item");
    for (var i=0; i<allStates.length;
        → i++) {
        statesArray[i] = allStates[i].
            .getElementsByTagName("label")
            .firstChild;
    }
}

```

Here's where we read that file in, looking at each item node, finding the label node inside, and then storing label's firstChild—the name of the state itself. Each of them goes into a slot in the StatesArray array.

```

5. var str = document.getElementById
    → ("searchField").value;
   document.getElementById
    → ("searchField").className = "";

```

When you start typing in the field, you'll end up here, in the searchSuggest() event handler function. We start off by getting the value of searchField, which is whatever has been typed so far. Next, we clear that field's class attribute.

continues on next page

Chapter 15

```
6. if (str != "") {
    document.getElementById("popups").
    →innerHTML = "";
```

If nothing's been entered, we don't want to do anything, so there's a check here to make sure that the user's entry has a value before we start popping up possibilities. If there's something there, we then blank out the previous list of possibilities.

```
7. for (var i=0; i<statesArray.length;
    →i++) {
    var thisState = statesArray[i].
    →nodeValue;
```

Now, we loop through the list of states, storing the current state we're looking at in `thisState`.

```
8. if (thisState.toLowerCase().indexOf
    →(str.toLowerCase()) == 0) {
```

We want to see if what they've entered so far is part of a state name—but that alone isn't sufficient; we also have to make sure that what they've entered is at the beginning of the name. If you type in Kansas, you don't want to see a drop-down box asking if you want Arkansas or Kansas, after all. And so long as we're doing that check, we'll also force the comparison to be lowercase on both sides before checking `indexOf()`.

If `indexOf()` returns 0—that is, the entered string was found starting at position 1 of `thisState`—then we know we have a hit.

Script 15.13 continued

```
catch (e) { }

}

if (xhr) {
    xhr.onreadystatechange = setStatesArray;
    xhr.open("GET", "us-states.xml", true);
    xhr.send(null);
}
else {
    alert("Sorry, but I couldn't create an
    →XMLHttpRequest");
}

function setStatesArray() {
    if (xhr.readyState == 4) {
        if (xhr.status == 200) {
            if (xhr.responseXML) {
                var allStates = xhr.responseXML.
                →getElementsByTagName("item");
                for (var i=0; i<allStates.length;
                →i++) {
                    statesArray[i] = allStates[i].
                    →getElementsByTagName
                    →("label")[0].firstChild;
                }
            }
        }
        else {
            alert("There was a problem with the
            →request " + xhr.status);
        }
    }
}

function searchSuggest() {
    var str = document.getElementById
    →("searchField").value;
    document.getElementById("searchField").
    →className = "";
    if (str != "") {
```

(script continues on next page)

Script 15.13 continued

```

document.getElementById("popups").
    .innerHTML = "";

for (var i=0; i<statesArray.length; i++) {
    var thisState = statesArray[i].
        .nodeValue;

    if (thisState.toLowerCase().indexOf
        (str.toLowerCase()) == 0) {
        var tempDiv = document.
            .createElement("div");
        tempDiv.innerHTML = thisState;
        tempDiv.onclick = makeChoice;
        tempDiv.className = "suggestions";
        document.getElementById("popups").
            .appendChild(tempDiv);
    }
}

var foundCt = document.getElementById
    ("popups").childNodes.length;
if (foundCt == 0) {
    document.getElementById
        ("searchField").className = "error";
}
if (foundCt == 1) {
    document.getElementById
        ("searchField").value =
        document.getElementById("popups").
            .firstChild.innerHTML;
    document.getElementById("popups").
        .innerHTML = "";
}
}

function makeChoice(evt) {
    var thisDiv = (evt) ? evt.target :
        window.event.srcElement;
    document.getElementById("searchField").
        .value = thisDiv.innerHTML;
    document.getElementById("popups").
        .innerHTML = "";
}

```

```

9. var tempDiv = document.
    .createElement("div");
tempDiv.innerHTML = thisState;
tempDiv.onclick = makeChoice;
tempDiv.className = "suggestions";
document.getElementById("popups").
    .appendChild(tempDiv);

```

Because this state is a possibility, we want to add it to the list that will display. That's done by creating a temporary div, setting its innerHTML to the name of the state, adding an onclick handler and className, and then appending the whole to the popups div. Adding each state as a separate div allows us to manipulate each using JavaScript and CSS.

```

10. var foundCt = document.
    .getElementById("popups").
    .childNodes.length;

```

When we've looped through all the states, we're done setting up the popups—but how many do we have? We calculate that, the foundCt, here.

```

11. if (foundCt == 0) {
    document.getElementById
        ("searchField").className =
        "error";
}

```

If foundCt is 0, they've entered something unexpected. We let them know that by setting the className to "error", which causes the entry field to display with a pale yellow background (based on a CSS style rule in Script 15.12).

continues on next page

```

12. if (foundCt == 1) {
    document.getElementById
    → ("searchField").value =
    → document.getElementById
    → ("popups").firstChild.
    → innerHTML;
    document.getElementById
    → ("popups").innerHTML = "";
}

```

If foundCt is one, however, we know that they've got a unique hit, so we can then put that state into the entry field. If they've typed in ca, they shouldn't have to type in li for nia also; we already know which state they want. We give them the full state by using the single div in popups to fill in the entry field, and then we blank out the popups div.

```

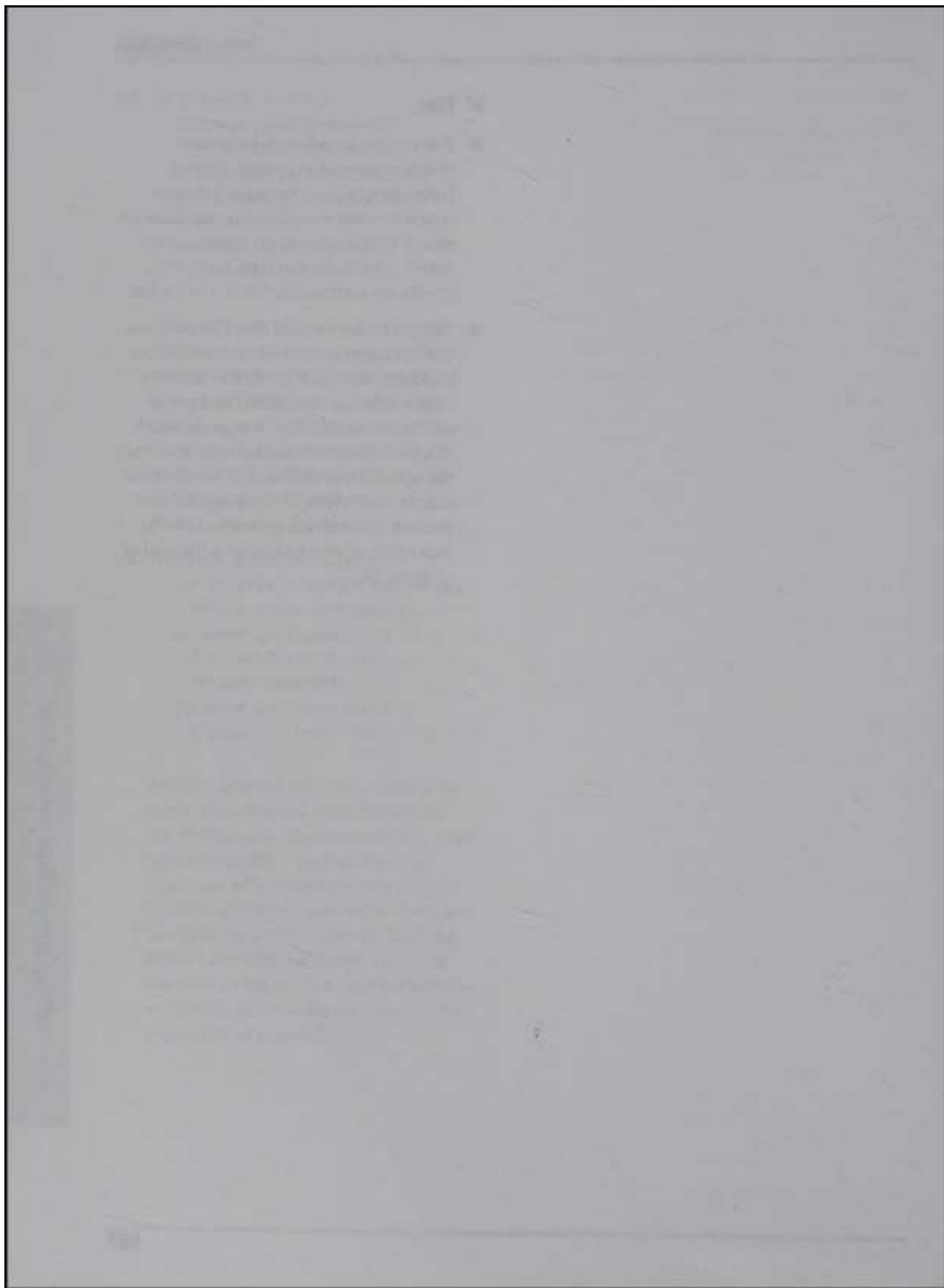
13. function makeChoice(evt) {
    var thisDiv = (evt) ? evt.target :
    → window.event.srcElement;
    document.getElementById
    → ("searchField").value =
    → thisDiv.innerHTML;
    document.getElementById
    → ("popups").innerHTML = "";
}

```

Another way the user can enter a state name is to click one from the pop-up list. In that case, the makeChoice() event handler function is called. First, we figure out which state the user clicked by looking at the target of the event, and that gives us a particular div. Looking at the innerHTML for that div gives us the state name, and we put that into the entry field. And finally, we clear out the pop-up list of possibilities.

✓ Tips

- You can see an advanced example of this technique at Google Suggest (labs.google.com/suggest/). As you type into what looks like the usual Google search field, a pop-up list appears with search results. As you type, the search results are continually filtered in the list.
- You may have noticed that this task and the last spent a lot more time and effort making things look good than actually demonstrating Ajax, XML, and server-side technologies. That's because much of what has come to be known (or at least thought of by some) as Ajax involves not just the underlying technology but also the way the technology works. How to make that all much simpler is the goal of Chapter 16.





VISUAL QUICKSTART GUIDE

BOOK LEVEL



✓ beginning
✓ intermediate
advanced

JAVASCRIPT & AJAX

FOR THE WEB, SIXTH EDITION

Visual QuickStart Guide—the quick and easy way to learn!

- **Easy visual approach** uses pictures to guide you through JavaScript and Ajax and show you what to do.
- **Concise steps and explanations** let you get up and running in no time.
- **Page for page, the best content and value around.**
- **Companion Web site** at www.javascriptworld.com offers sample scripts, updates, and more.
- **Tom Negrino** is the author of dozens of books including Visual QuickStart Guides covering Keynote and Macromedia Contribute, and Visual QuickProject Guides on PowerPoint, Quicken, and upgrading Mac OS X. **Dori Smith** is the author of *Java 2 for the World Wide Web: Visual QuickStart Guide*. She is a frequent speaker at industry conferences, publisher of the Wise-Web community, and a member of the Web Standards Project. Together they have written the best-selling Visual QuickStart Guides on JavaScript and Dreamweaver. They also publish numerous print and online articles. They also publish a popular weblog.



978-0466-071



Peachpit Press

1249 Eighth Street, Berkeley, CA 94710
800 283-9444 • 510 524-2178 • fax 510 524-2221
Find us on the Web at: www.peachpit.com

FOR COMPUTERS USING: Internet Explorer 6 or later; Firefox 1.5 or later; Safari 2 or later; Netscape 7 or later

COMPUTER BOOKS SHELF CATEGORY: JavaScript/Programming Languages

US \$24.99 • Canada \$29.99 • UK £17.99

ISBN 0-321-43032-8



9 780321 430328