O'REILLY®

"The Missing Manual series is simply the most intelligent and usable series of guidebooks..."
—KEVIN KELLY, CO-FOUNDER OF *WIRED*

JavaScript & jQuery

the missing manual®

The book that should have been in the box®

3rd Edition
Covers jQuery UI

David Sawyer McFarland

O'REILLY®

Answers found here!

JavaScript lets you supercharge your web pages with animation, interactivity, and visual effects, but learning the language isn't easy. This fully updated and expanded guide takes you step-by-step through JavaScript basics, then shows you how to save time and effort with jQuery—the library of prewritten JavaScript code—and the newest innovations from the jQuery UI plug-in.

The important stuff you need to know

- **Make your pages come alive.** Use jQuery to create interactive elements that respond to visitor input.
- **Get acquainted with jQuery UI.** Expand your interface with tabbed panels, dialog boxes, date pickers, and other widgets.
- **Display good forms.** Get information from visitors, help shoppers buy goods, and let members post their thoughts.
- **Go beyond the browser with Ajax.** Communicate with the web server to update your pages without reloading.
- **Put your new skills right to work.** Create a simple application step-by-step, using jQuery and jQuery UI widgets.
- **Dive into advanced concepts.** Use ThemeRoller to customize your widgets; avoid common errors that new programmers often make.

David Sawyer McFarland, president of Sawyer McFarland Media, Inc., has spent nearly 20 years building and managing websites. Having served as webmaster at UC Berkeley, he's also taught at the UC Berkeley Graduate School of Journalism and the Portland State University multimedia program. David lives in Portland and has written bestselling Missing Manual titles on Adobe Dreamweaver and CSS

Web Authoring and Design

US $49.99 CAN $52.99

ISBN: 978-1-491-94707-4

the missing manual®

missingmanuals.com
twitter: @missingmanuals
facebook.com/MissingManuals

JavaScript & jQuery

the missing manual®

The book that should have been in the box®

David Sawyer McFarland

O'REILLY®

Beijing | Cambridge | Farnham | Köln | Sebastopol | Tokyo

JavaScript & jQuery: The Missing Manual

by David Sawyer McFarland

Published by O'Reilly Media, Inc.,
1005 Gravenstein Highway North, Sebastopol, CA 95472.

July 2008: First Edition.
October 2011: Second Edition.
September 2014: Third Edition.

Revision History for the Third Edition:

2014-09-10 First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491947074* for release details.

ISBN-13: 978-1-491-94707-4

[LSI]

Contents

Part Four: Advanced jQuery and JavaScript

Part Five: Tips, Tricks, and Troubleshooting

Part Six: **Appendix**

The Missing Credits

ABOUT THE AUTHOR

David Sawyer McFarland is president of Sawyer McFarland Media, Inc., a web development and training company in Portland, Oregon. He's been building websites since 1995, when he designed his first site—an online magazine for communication professionals. He's served as webmaster at the University of California at Berkeley and the Berkeley Multimedia Research Center, and oversaw a complete CSS-driven redesign of Macworld.com.

In addition to building websites, David is also a writer, trainer, and instructor. He's taught web design at UC Berkeley Graduate School of Journalism, the Center for Electronic Art, the Academy of Art College, Ex'Pressions Center for New Media, and Portland State University. He's written articles about the web for *Practical Web Design, MX Developer's Journal, Macworld* magazine, and CreativePro.com.

He welcomes feedback about this book by email: *missing@sawmac.com*. (If you're seeking technical help, however, please refer to the sources listed in Appendix A.)

ABOUT THE CREATIVE TEAM

Nan Barber (editor) is associate editor for the Missing Manual series. She lives in Massachusetts with her husband and various electronic devices. Email: *nanbarber@gmail.com*.

Melanie Yarbrough (production editor) works and plays in Cambridge, Massachusetts, where she bakes up whatever she can imagine and bikes around the city. Email: *myarbrough@oreilly.com*.

Jennifer Davis (technical reviewer) is an engineer with years of experience improving platform development efficiency. As a Chef Automation engineer, she helps companies discover their own best practices to improving workflow reducing mean time to deploy. She is an event organizer for Reliability Engineering, the Bay Area Chef user group.

Alex Stangl (technical reviewer) has developed software professionally for 25+ years, using a myriad of languages and technologies. He enjoys challenging problems and puzzles, learning new languages (currently Clojure), doing technical reviews, and being a good dad and husband. Email: *alex@stangl.us*.

Jasmine Kwityn (proofreader) is a freelance copyeditor and proofreader. She lives in New Jersey with her husband, Ed, and their three cats, Mushki, Axle, and Punky. Email: *jasminekwityn@gmail.com*.

Bob Pfahler (indexer) is a freelance indexer who indexed this book on behalf of Potomac Indexing, LLC, an international indexing partnership at *www.potomacindexing.com*. Besides the subject of computer technology, he specializes in business, management, biography, and history. Email: *bobpfahler@hotmail.com*.

ACKNOWLEDGMENTS

Many thanks to all those who helped with this book, including Jennifer Davis and Alex Stangl, whose watchful eyes saved me from potentially embarrassing mistakes. Thanks also to my many students at Portland State University who have sat through my long JavaScript lectures and struggled through my programming assignments—especially the members of Team Futzbit (Combination Pizza Hut and Taco Bell) for testing the tutorials: Julia Hall, Amber Brucker, Kevin Brown, Josh Elliott, Tracy O'Connor, and Blake Womack. Also, we all owe a big debt of gratitude to John Resig and the jQuery team for creating the best tool yet for making JavaScript fun.

Finally, thanks to David Pogue for getting me started; Nan Barber for making my writing sharper and clearer; my wife, Scholle, for putting up with an author's crankiness; and thanks to my kids, Graham and Kate, because they're just awesome.

—David Sawyer McFarland

THE MISSING MANUAL SERIES

Missing Manuals are witty, superbly written guides to computer products that don't come with printed manuals (which is just about all of them). Each book features a handcrafted index and cross-references to specific pages (not just chapters). Recent and upcoming titles include:

Access 2010: The Missing Manual by Matthew MacDonald

Access 2013: The Missing Manual by Matthew MacDonald

Adobe Edge Animate: The Missing Manual by Chris Grover

Buying a Home: The Missing Manual by Nancy Conner

Creating a Website: The Missing Manual, Third Edition by Matthew MacDonald

CSS3: The Missing Manual, Third Edition by David Sawyer McFarland

David Pogue's Digital Photography: The Missing Manual by David Pogue

Dreamweaver CS6: The Missing Manual by David Sawyer McFarland

Dreamweaver CC: The Missing Manual by David Sawyer McFarland and Chris Grover

Excel 2010: The Missing Manual by Matthew MacDonald

Excel 2013: The Missing Manual by Matthew MacDonald

Facebook: The Missing Manual, Third Edition by E. A. Vander Veer

FileMaker Pro 13: The Missing Manual by Susan Prosser and Stuart Gripman

Flash CS6: The Missing Manual by Chris Grover

Galaxy Tab: The Missing Manual by Preston Gralla

Galaxy S4: The Missing Manual by Preston Gralla

Galaxy S5: The Missing Manual by Preston Gralla

Google+: The Missing Manual by Kevin Purdy

HTML5: The Missing Manual, Second Edition by Matthew MacDonald

iMovie '11 & iDVD: The Missing Manual by David Pogue and Aaron Miller

iPad: The Missing Manual, Sixth Edition by J.D. Biersdorfer

iPhone: The Missing Manual, Seventh Edition by David Pogue

iPhone App Development: The Missing Manual by Craig Hockenberry

iPhoto '11: The Missing Manual by David Pogue and Lesa Snider

iPod: The Missing Manual, Eleventh Edition by J.D. Biersdorfer and David Pogue

Kindle Fire HD: The Missing Manual by Peter Meyers

Living Green: The Missing Manual by Nancy Conner

Microsoft Project 2010: The Missing Manual by Bonnie Biafore

Microsoft Project 2013: The Missing Manual by Bonnie Biafore

Motorola Xoom: The Missing Manual by Preston Gralla

NOOK HD: The Missing Manual by Preston Gralla

Office 2010: The Missing Manual by Nancy Conner and Matthew MacDonald

Office 2011 for Macintosh: The Missing Manual by Chris Grover

Office 2013: The Missing Manual by Nancy Conner and Matthew MacDonald

OS X Mountain Lion: The Missing Manual by David Pogue

OS X Mavericks: The Missing Manual by David Pogue

OS X Yosemite: The Missing Manual by David Pogue

Personal Investing: The Missing Manual by Bonnie Biafore

Photoshop CS6: The Missing Manual by Lesa Snider

Photoshop CC: The Missing Manual by Lesa Snider

Photoshop Elements 12: The Missing Manual by Barbara Brundage

PHP & MySQL: The Missing Manual, Second Edition by Brett McLaughlin

QuickBooks 2014: The Missing Manual by Bonnie Biafore

QuickBooks 2015: The Missing Manual by Bonnie Biafore

Switching to the Mac: The Missing Manual, Mavericks Edition by David Pogue

Switching to the Mac: The Missing Manual, Yosemite Edition by David Pogue

Windows 7: The Missing Manual by David Pogue

Windows 8: The Missing Manual by David Pogue

WordPress: The Missing Manual, Second Edition by Matthew MacDonald

Your Body: The Missing Manual by Matthew MacDonald

Your Brain: The Missing Manual by Matthew MacDonald

Your Money: The Missing Manual by J.D. Roth

For a full list of all Missing Manuals in print, go to *www.missingmanuals.com/library.html.*

Introduction

The Web was a pretty boring place in its early days. Web pages were constructed from plain old HTML, so they could display information, and that was about all. Folks would click a link and then wait for a new web page to load. That was about as interactive as it got.

These days, most websites are almost as responsive as the programs on a desktop computer, reacting immediately to every mouse click. And it's all thanks to the subjects of this book—JavaScript and its sidekick, jQuery.

What Is JavaScript?

JavaScript is a programming language that lets you supercharge your HTML with animation, interactivity, and dynamic visual effects.

JavaScript can make web pages more useful by supplying immediate feedback. For example, a JavaScript-powered shopping cart page can instantly display a total cost, with tax and shipping, the moment a visitor selects a product to buy. JavaScript can produce an error message immediately after someone attempts to submit a web form that's missing necessary information.

JavaScript also lets you create fun, dynamic, and interactive interfaces. For example, with JavaScript, you can transform a static page of thumbnail images into an animated slideshow. Or you can do something more subtle like stuff more information on a page without making it seem crowded by organizing content into bite-size panels that visitors can access with a simple click of the mouse (page 326). Or add something useful and attractive, like pop-up tooltips that provide supplemental information for items on your web page (page 321).

Another one of JavaScript's main selling points is its immediacy. It lets web pages respond instantly to actions like clicking a link, filling out a form, or merely moving the mouse around the screen. JavaScript doesn't suffer from the frustrating delay associated with server-side programming languages like PHP, which rely on communication between the web browser and the web server. Because it doesn't rely on constantly loading and reloading web pages, JavaScript lets you create web pages that feel and act more like desktop programs than web pages.

If you've visited Google Maps (*http://maps.google.com*), you've seen JavaScript in action. Google Maps lets you view a map of your town (or pretty much anywhere else for that matter), zoom in to get a detailed view of streets and bus stops, or zoom out to get a bird's-eye view of how to get across town, the state, or the nation. While there were plenty of map sites before Google, they always required reloading multiple web pages (usually a slow process) to get to the information you wanted. Google Maps, on the other hand, works without page refreshes—it responds immediately to your choices.

The programs you create with JavaScript can range from the really simple (like popping up a new browser window with a web page in it) to full-blown web applications like Google Docs (*http://docs.google.com*), which lets you create presentations, edit documents, and build spreadsheets using your web browser with the feel of a program running directly on your computer.

A Bit of History

Invented in 10 days by Brendan Eich at Netscape back in 1995, JavaScript is nearly as old as the Web itself. While JavaScript is well respected today, it has a somewhat checkered past. It used to be considered a hobbyist's programming language, used for adding less-than-useful effects such as messages that scroll across the bottom of a web browser's status bar like a stock ticker, or animated butterflies following mouse movements around the page. In the early days of JavaScript, it was easy to find thousands of free JavaScript programs (also called *scripts*) online, but many of those scripts didn't work in all web browsers, and at times even crashed browsers.

NOTE JavaScript has little to do with the Java programming language. JavaScript was originally named LiveScript, but a quick deal by marketers at Netscape eager to cash in on the success of Sun Microsystem's then-hot programming language led to this long-term confusion. Don't make the mistake of confusing the two...especially at a job interview!

In the early days, JavaScript also suffered from incompatibilities between the two prominent browsers, Netscape Navigator and Internet Explorer. Because Netscape and Microsoft tried to outdo each other's browsers by adding newer and (ostensibly) better features, the two browsers often acted in very different ways, making it difficult to create JavaScript programs that worked well in both.

NOTE After Netscape introduced JavaScript, Microsoft introduced jScript, their own version of JavaScript included with Internet Explorer.

Fortunately, the worst of those days is nearly gone and contemporary browsers like Firefox, Safari, Chrome, Opera, and Internet Explorer 11 have standardized much of the way they handle JavaScript, making it easier to write JavaScript programs that work for most everyone. (There are still a few incompatibilities among current web browsers, so you'll need to learn a few tricks for dealing with cross-browser problems. You'll learn how to overcome browser incompatibilities in this book.)

In the past several years, JavaScript has undergone a rebirth, fueled by high-profile websites like Google, Yahoo!, and Flickr, which use JavaScript extensively to create interactive web applications. There's never been a better time to learn JavaScript. With the wealth of knowledge and the quality of scripts being written, you can add sophisticated interaction to your website—even if you're a beginner.

NOTE JavaScript is also known by the name ECMAScript. ECMAScript is the "official" JavaScript specification, which is developed and maintained by an international standards organization called Ecma International: *www.ecmascript.org.*

JavaScript Is Everywhere

JavaScript isn't just for web pages, either. It's proven to be such a useful programming language that if you learn JavaScript you can create Yahoo! Widgets and Google Apps, write programs for the iPhone, and tap into the scriptable features of many Adobe programs like Acrobat, Photoshop, Illustrator, and Dreamweaver. In fact, Dreamweaver has always offered clever JavaScript programmers a way to add their own commands to the program.

In the Yosemite version of the Mac OS X operating system, Apple lets users automate their Macs using JavaScript. In addition, JavaScript is used in many helpful front end web development tools like Gulp.js (which can automatically compress images and CSS and JavaScript files) and Bower (which makes it quick and easy to download common JavaScript libraries like jQuery, jQuery UI, or AngularJS to your computer).

JavaScript is also becoming increasingly popular for server-side development. The Node.js platform (a version of Google's V8 JavaScript engine that runs JavaScript on the server) is being embraced eagerly by companies like Walmart, PayPal, and eBay. Learning JavaScript can even lead to a career in building complex server-side applications. In fact, the combination of JavaScript on the *frontend* (that is, JavaScript running in a web browser) and the *backend* (on the web server) is known as *full stack JavaScript development*.

In other words, there's never been a better time to learn JavaScript!

What Is jQuery?

JavaScript has one embarrassing little secret: writing it can be hard. While it's simpler than many other programming languages, JavaScript is still a programming language. And many people, including web designers, find programming difficult.

To complicate matters further, different web browsers understand JavaScript differently, so a program that works in, say, Chrome may be completely unresponsive in Internet Explorer 9. This common situation can cost many hours of testing on different machines and different browsers to make sure a program works correctly for your site's entire audience.

That's where jQuery comes in. jQuery is a JavaScript library intended to make JavaScript programming easier and more fun. A JavaScript library is a complex set of JavaScript code that both simplifies difficult tasks and solves cross-browser problems. In other words, jQuery solves the two biggest JavaScript headaches: complexity and the finicky nature of different web browsers.

jQuery is a web designer's secret weapon in the battle of JavaScript programming. With jQuery, you can accomplish tasks in a single line of code that could take hundreds of lines of programming and many hours of browser testing to achieve with your own JavaScript code. In fact, an in-depth book solely about JavaScript would be at least twice as thick as the one you're holding; and, when you were done reading it (if you could manage to finish it), you wouldn't be able to do half of the things you can accomplish with just a little bit of jQuery knowledge.

That's why most of this book is about jQuery. It lets you do so much, so easily. Another great thing about jQuery is that you can add advanced features to your website with thousands of easy-to-use jQuery plug-ins. For example, the jQuery UI plug-in (which you'll meet on page 299) lets you create many complex user interface elements like tabbed panels, drop-down menus, pop-up date-picker calendars—all with a single line of programming!

Unsurprisingly, jQuery is used on millions of websites (*http://trends.builtwith.com/javascript/jQuery*). It's baked right into popular content management systems like Drupal and WordPress. You can even find job listings for "jQuery Programmers" with no mention of JavaScript. When you learn jQuery, you join a large community of fellow web designers and programmers who use a simpler and more powerful approach to creating interactive, powerful web pages.

HTML: The Barebones Structure

JavaScript isn't much good without the two other pillars of web design—HTML and CSS. Many programmers talk about the three languages as forming the "layers" of a web page: HTML provides the *structural* layer, organizing content like pictures and words in a meaningful way; CSS (Cascading Style Sheets) provides the *presentational* layer, making the content in the HTML look good; and JavaScript adds a *behavioral* layer, bringing a web page to life so it interacts with web visitors.

In other words, to master JavaScript, you need to have a good understanding of both HTML and CSS.

NOTE For a full-fledged introduction to HTML and CSS, check out *Head First HTML with CSS and XHTML* by Elisabeth Robson and Eric Freeman. For an in-depth presentation of the tricky subject of Cascading Style Sheets, pick up a copy of *CSS3: The Missing Manual* by David Sawyer McFarland (both from O'Reilly).

HTML (Hypertext Markup Language) uses simple commands called *tags* to define the various parts of a web page. For example, this HTML code creates a simple web page:

```
<!DOCTYPE html>
<html>
<head>
<meta charset=utf-8>
<title>Hey, I am the title of this web page.</title>
</head>
<body>
Hey, I am some body text on this web page.
</body>
</html>
```

It may not be exciting, but this example has all the basic elements a web page needs. This page begins with a single line—the document type declaration, or *doctype* for short—that states what type of document the page is and which standards it conforms to. HTML actually comes in different versions, and you use a different doctype with each. In this example, the doctype is for HTML5; the doctype for an HTML 4.01 or XHTML document is longer and also includes a URL that points the web browser to a file on the Internet that contains definitions for that type of file.

In essence, the doctype tells the web browser how to display the page. The doctype can even affect how CSS and JavaScript work. With an incorrect or missing doctype, you may end up banging your head against a wall as you discover lots of cross-browser differences with your scripts. If for no other reason, always include a doctype in your HTML.

Historically, there have been many doctypes—HTML 4.01 Transitional, HTML 4.01 Strict, XHTML 1.0 Transitional, XHTML 1.0 Strict—but they required a long line of confusing code that was easy to mistype. HTML5's doctype—`<!DOCTYPE html>`—is short, simple, and the one you should use.

How HTML Tags Work

In the example in the previous section, as in the HTML code of any web page, you'll notice that most instructions appear in pairs that surround a block of text or other commands. Sandwiched between brackets, these *tags* are instructions that tell a web browser how to display the web page. Tags are the "markup" part of the Hypertext Markup Language.

The starting (*opening*) tag of each pair tells the browser where the instruction begins, and the ending tag tells it where the instruction ends. Ending or *closing* tags always include a forward slash (/) after the first bracket symbol (<). For example, the tag

 marks the start of a paragraph, while marks its end. Some tags don't have closing tags, like <img>, <input>, and <br> tags, which consist of just a single tag.

For a web page to work correctly, you must include at least these three tags:

- The <html> tag appears once at the beginning of a web page (after the doctype) and again (with an added slash) at the end. This tag tells a web browser that the information contained in this document is written in HTML, as opposed to some other language. All of the contents of a page, including other tags, appear between the opening and closing <html> tags.

  If you were to think of a web page as a tree, the <html> tag would be its root. Springing from the root are two branches that represent the two main parts of any web page—the *head* and the *body*.

- The *head* of a web page, surrounded by <head> tags, contains the title of the page. It may also provide other, invisible information (such as search keywords) that browsers and web search engines can exploit.

  In addition, the head can contain information that's used by the web browser for displaying the web page and for adding interactivity. You put Cascading Style Sheets, for example, in the head of the document. The head of the document is also where you often include JavaScript programming and links to JavaScript files.

- The *body* of a web page, as set apart by its surrounding <body> tags, contains all the information that appears inside a browser window: headlines, text, pictures, and so on.

Within the <body> tag, you commonly find tags like the following:

- You tell a web browser where a paragraph of text begins with a (opening paragraph tag), and where it ends with a (closing paragraph tag).

- The tag emphasizes text. If you surround some text with it and its partner tag, , you get boldface type. The HTML snippet Warning! tells a web browser to display the word "Warning!" in bold type.

- The <a> tag, or anchor tag, creates a *hyperlink* in a web page. When clicked, a hyperlink—or *link*—can lead anywhere on the Web. You tell the browser where the link points by putting a web address inside the <a> tags. For instance, you might type Click here!.

  The browser knows that when your visitor clicks the words "Click here!" it should go to the Missing Manuals website. The href part of the tag is called an *attribute* and the URL (the uniform resource locator or web address) is the *value*. In this example, *http://www.missingmanuals.com* is the *value* of the href attribute.

UP TO SPEED

Validating Web Pages

As mentioned on page xvii, a web page's doctype identifies which type of HTML or XHTML you used to create the web page. The rules differ subtly depending on type: For example, unlike HTML 4.01, XHTML doesn't let you have an unclosed <p> tag, and requires that all tag names and attributes be lowercase (<a> *not* <A>, for example). HTML5 includes new tags and lets you use either HTML or XHTML syntax. Because different rules apply to each variant of HTML, you should always *validate* your web pages.

An HTML validator is a program that makes sure a web page is written correctly. It checks the page's doctype and then analyzes the code in the page to see whether it matches the rules defined by that doctype. For example, the validator flags mistakes like a misspelled tag name or an unclosed tag. The World Wide Web Consortium (W3C), the organization that's responsible for many of the technologies used on the Web, has a free online validator at *http://validator.w3.org*. You can copy your HTML and paste it into a web form, upload a web page, or point the validator to an already existing page on the Web; the validator then analyzes the HTML and reports back whether the page is valid or not. If there are any errors, the validator tells you what the error is and on which line of the HTML file it occurs.

Valid HTML isn't just good form—it also helps to make sure your JavaScript programs work correctly. A lot of JavaScript involves manipulating a web page's HTML: identifying a particular form field, for example, or placing new HTML (like an error message) in a particular spot. In order for JavaScript to access and manipulate a web page, the HTML must be in proper working order. Forgetting to close a tag, using the same ID name more than once, or improperly nesting your HTML tags can make your JavaScript code behave erratically or not at all.

CSS: Adding Style to Web Pages

At the beginning of the Web, HTML was the only language you needed to know. You could build pages with colorful text and graphics and make words jump out using different sizes, fonts, and colors. But today, web designers turn to Cascading Style Sheets to add visual sophistication to their pages. CSS is a formatting language that lets you make text look good, build complex page layouts, and generally add style to your site.

Think of HTML as merely the language you use to structure a page. It helps identify the stuff you want the world to know about. Tags like <h1> and <h2> denote headlines and assign them relative importance: A *heading 1* is more important than a *heading 2*. The <p> tag indicates a basic paragraph of information. Other tags provide further structural clues: for example, a <ul> tag identifies a bulleted list (to make a list of recipe ingredients more intelligible, for example).

CSS, on the other hand, adds design flair to well-organized HTML content, making it more beautiful and easier to read. Essentially, a CSS *style* is just a rule that tells a web browser how to display a particular element on a page. For example, you can create a CSS rule to make all <h1> tags appear 36 pixels tall, in the Verdana font, and in orange. CSS can do more powerful stuff, too, like add borders, change margins, and even control the exact placement of a page element.

When it comes to JavaScript, some of the most valuable changes you make to a page involve CSS. You can use JavaScript to add or remove a CSS style from an HTML tag, or dynamically change CSS properties based on a visitor's input or mouse clicks. You can even animate from the properties of one style to the properties of another (say, animating a background color changing from yellow to red). For example, you can make a page element appear or disappear simply by changing the CSS `display` property. To animate an item across the screen, you can change the CSS position properties dynamically using JavaScript.

Anatomy of a Style

A single style that defines the look of one element is a pretty basic beast. It's essentially a rule that tells a web browser how to format something—turn a headline blue, draw a red border around a photo, or create a 150-pixel-wide sidebar box to hold a list of links. If a style could talk, it would say something like, "Hey, Browser, make *this* look like *that*." A style is, in fact, made up of two elements: the web page element that the browser formats (the *selector*) and the actual formatting instructions (the *declaration block*). For example, a selector can be a headline, a paragraph of text, a photo, and so on. Declaration blocks can turn that text blue, add a red border around a paragraph, position the photo in the center of the page—the possibilities are endless.

NOTE Technical types often follow the lead of the W3C and call CSS styles *rules*. This book uses the terms "style" and "rule" interchangeably.

Of course, CSS styles can't communicate in nice, clear English. They have their own language. For example, to set a standard font color and font size for all paragraphs on a web page, you'd write the following:

```
p { color: red; font-size: 1.5em; }
```

This style simply says, "Make the text in all paragraphs—marked with <p> tags—red and 1.5 ems tall." (An *em* is a unit or measurement that's based on a browser's normal text size.) As Figure I-1 illustrates, even a simple style like this example contains several elements:

- **Selector.** The selector tells a web browser which element or elements on a page to style—like a headline, paragraph, image, or link. In Figure I-1, the selector (p) refers to the <p> tag, which makes web browsers format all <p> tags using the formatting directions in this style. With the wide range of selectors that CSS offers and a little creativity, you can gain fine control of your pages' formatting. (Selectors are an important part of using jQuery, so you'll find a detailed discussion of them starting on page 119.)
- **Declaration block.** The code following the selector includes all the formatting options you want to apply to the selector. The block begins with an opening brace ({) and ends with a closing brace (}).

- **Declaration.** Between the opening and closing braces of a declaration, you add one or more *declarations*, or formatting instructions. Every declaration has two parts, a *property* and a *value*, and ends with a semicolon. A colon separates the property name from its value: `color : red;`.
- **Property.** CSS offers a wide range of formatting options, called *properties*. A property is a word—or a few hyphenated words—indicating a certain style effect. Most properties have straightforward names like `font-size`, `margin-top`, and `background-color`. For example, the `background-color` property sets—you guessed it—a background color.

NOTE If you need to brush up on your CSS, grab a copy of *CSS3: The Missing Manual.*

- **Value**. Finally, you get to express your creative genius by assigning a *value* to a CSS property—by making a background blue, red, purple, or chartreuse, for example. Different CSS properties require specific types of values—a color (like `red`, or `#FF0000`), a length (like `18px`, `2in`, or `5em`), a URL (like *images/background.gif*), or a specific keyword (like `top`, `center`, or `bottom`).

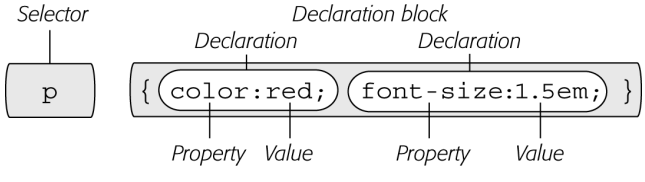

FIGURE I-1
A style (or rule) is made of two main parts: a selector, which tells web browsers what to format, and a declaration block, which lists the formatting instructions that the browsers use to style the selector.

You don't need to write a style on a single line as pictured in Figure I-1. Many styles have multiple formatting properties, so you can make them easier to read by breaking them up into multiple lines. For example, you may want to put the selector and opening brace on the first line, each declaration on its own line, and the closing brace by itself on the last line, like so:

```
p {
    color: red;
    font-size: 1.5em;
}
```

It's also helpful to indent properties, with either a tab or a couple of spaces, to visibly separate the selector from the declarations, making it easy to tell which is which. And finally, putting one space between the colon and the property value is optional, but adds to the readability of the style. In fact, you can put as much white space between the two as you want. For example, `color:red`, `color: red`, and `color : red` all work.

Software for JavaScript Programming

To create web pages made up of HTML, CSS, and JavaScript, you need nothing more than a basic text editor like Notepad (Windows) or TextEdit (Mac). But after typing a few hundred lines of JavaScript code, you may want to try a program better suited to working with web pages. This section lists some common editors—some free and some you can buy.

NOTE There are literally hundreds of tools that can help you create web pages and write JavaScript programs, so the following is by no means a complete list. Think of it as a greatest-hits tour of the most popular programs that JavaScript fans are using today.

Free Programs

There are plenty of free programs out there for editing web pages and style sheets. If you're still using Notepad or TextEdit, give one of these a try. Here's a short list to get you started:

- **Brackets** (Windows, Mac, and Linux, *http://brackets.io*) is an open source code editor from Adobe. It's free (there is a commercial version with more features named Edge Code), has many great features including a great live browser preview, and is even written in JavaScript!
- **Notepad++** (Windows, *http://notepad-plus-plus.org*) is a coder's friend. It highlights the syntax of JavaScript and HTML code, and lets you save macros and assign keyboard shortcuts to them so you can automate the process of inserting the code snippets you use most.
- **HTML-Kit** (Windows, *www.chami.com/html-kit*) is a powerful HTML/XHTML editor that includes lots of useful features, like the ability to preview a web page directly in the program (so you don't have to switch back and forth between browser and editor), shortcuts for adding HTML tags, and a lot more.
- **CoffeeCup Free HTML Editor** (Windows, *www.coffeecup.com/free-editor*) is the free version of the commercial ($49) CoffeeCup HTML editor.
- **TextWrangler** (Mac, *www.barebones.com/products/textwrangler*) is free software that's actually a pared-down version of BBEdit, the sophisticated, well-known text editor for the Mac. TextWrangler doesn't have all of BBEdit's built-in HTML tools, but it does include syntax coloring (highlighting tags and properties in different colors so it's easy to scan a page and identify its parts), FTP support (so you can upload files to a web server), and more.
- **Eclipse** (Windows, Linux, and Mac, *www.eclipse.org*) is a free, popular choice among Java Developers, but includes tools for working with HTML, CSS, and JavaScript. A version specifically for JavaScript developers is also available (*www.eclipse.org/downloads/packages/eclipse-ide-javascript-web-developers/indigor*), as well as Eclipse plug-ins to add autocomplete for jQuery (*http://marketplace.eclipse.org/category/free-tagging/jquery*).

- **Aptana Studio** (Windows, Linux, and Mac, *www.aptana.org*) is a powerful, free coding environment with tools for working with HTML, CSS, JavaScript, PHP, and Ruby on Rails.
- **Vim** and **Emacs** are tried and true text editors from the Unix world. They're included with OS X and Linux, and you can download versions for Windows. They're loved by serious programmers, but have a steep learning curve for most people.

Commercial Software

Commercial website development programs range from inexpensive text editors to complete website construction tools with all the bells and whistles:

- **Atom** (Windows and Mac, *https://atom.io*) is a new kid on the block. It's not yet available for sale, but the beta version is free for now. Atom is developed by the folks at GitHub (a site for sharing and collaboratively working on projects), and offers a large array of features built specifically for the needs of today's developers. It features a modular design, which allows for lots of third-party plug-ins that enhance the program's functionality.
- **SublimeText** (Windows, Mac, and Linux, *https://www.sublimetext.com*) is a darling of many programmers. This text editor ($70) includes many timesaving features for JavaScript programmers, like "auto-paired characters," which automatically plops in the second character of a pair of punctuation marks (for example, the program automatically inserts a closing parenthesis after you type an opening parenthesis).
- **EditPlus** (Windows, *www.editplus.com*) is an inexpensive text editor ($35) that includes syntax coloring, FTP, autocomplete, and other wrist-saving features.
- **BBEdit** (Mac, *www.barebones.com/products/bbedit*). This much-loved Mac text editor ($99.99) has plenty of tools for working with HTML, XHTML, CSS, JavaScript, and more. It includes many useful web building tools and shortcuts.
- **Dreamweaver** (Mac and Windows, *www.adobe.com/products/dreamweaver.html*) is a visual web page editor ($399). It lets you see how your page looks in a web browser. The program also includes a powerful text editor for writing JavaScript programs and excellent CSS creation and management tools. Check out *Dreamweaver CC: The Missing Manual* for the full skinny on how to use this powerful program.

About This Book

Unlike a piece of software such as Microsoft Word or Dreamweaver, JavaScript isn't a single product developed by a single company. There's no support department at JavaScript headquarters writing an easy-to-read manual for the average web developer. While you'll find plenty of information on sites like Mozilla.org (see, for example,

https://developer.mozilla.org/en/JavaScript/Reference or *www.ecmascript.org*), there's no definitive source of information on the JavaScript programming language.

Because there's no manual for JavaScript, people just learning JavaScript often don't know where to begin. And the finer points regarding JavaScript can trip up even seasoned web pros. The purpose of this book, then, is to serve as the manual that should have come with JavaScript. In this book's pages, you'll find step-by-step instructions for using JavaScript to create highly interactive web pages.

Likewise, you'll find good documentation on jQuery at *http://api.jquery.com.* But it's written by programmers for programmers, and so the explanations are mostly brief and technical. And while jQuery is generally more straightforward than regular JavaScript programming, this book will teach you fundamental jQuery principles and techniques so you can start off on the right path when enhancing your websites with jQuery.

JavaScript & jQuery: The Missing Manual is designed to accommodate readers who have some experience building web pages. You'll need to feel comfortable with HTML and CSS to get the most from this book, because JavaScript often works closely with HTML and CSS to achieve its magic. The primary discussions are written for advanced-beginner or intermediate computer users. But if you're new to building web pages, special boxes called Up to Speed provide the introductory information you need to understand the topic at hand. If you're an advanced web page jockey, on the other hand, keep your eye out for similar shaded boxes called Power Users' Clinic. They offer more technical tips, tricks, and shortcuts for the experienced computer fan.

NOTE This book periodically recommends *other* books, covering topics that are too specialized or tangential for a manual about using JavaScript. Sometimes the recommended titles are from Missing Manual series publisher O'Reilly Media—but not always. If there's a great book out there that's not part of the O'Reilly family, we'll let you know about it.

This Book's Approach to JavaScript

JavaScript is a real programming language: It doesn't work like HTML or CSS, and it has its own set of (often complicated) rules. It's not always easy for web designers to switch gears and start thinking like computer programmers, and there's no *one* book that can teach you everything there is to know about JavaScript.

The goal of *JavaScript & jQuery: The Missing Manual* isn't to turn you into the next great programmer (though it might start you on your way). This book is meant to familiarize web designers with the ins and outs of JavaScript and then move on to jQuery so that you can add really useful interactivity to a website as quickly and easily as possible.

In this book, you'll learn the basics of JavaScript and programming; but just the basics won't make for very exciting web pages. It's not possible in 500 pages to teach you everything about JavaScript that you need to know to build sophisticated,

interactive web pages. Instead, much of this book will cover the wildly popular jQuery JavaScript library, which, as you'll soon learn, will liberate you from all of the minute, time-consuming details of creating JavaScript programs that run well across different browsers.

You'll learn the basics of JavaScript, and then jump immediately to advanced web page interactivity with a little help—OK, a *lot* of help—from jQuery. Think of it this way: You could build a house by cutting down and milling your own lumber, constructing your own windows, doors, and doorframes, manufacturing your own tile, and so on. That do-it-yourself approach is common to a lot of JavaScript books. But who has that kind of time? This book's approach is more like building a house by taking advantage of already-built pieces and putting them together using basic skills. The end result will be a beautiful and functional house built in a fraction of the time it would take you to learn every step of the process.

About the Outline

JavaScript & jQuery: The Missing Manual is divided into five parts, each containing several chapters:

- **Part One** starts at the very beginning. You'll learn the basic building blocks of JavaScript as well as get some helpful tips on computer programming in general. This section teaches you how to add a script to a web page, store and manipulate information, and add smarts to a program so it can respond to different situations. You'll also learn how to communicate with the browser window, store and read cookies, respond to various events like mouse clicks and form submissions, and modify the HTML of a web page.
- **Part Two** introduces jQuery—the Web's most popular JavaScript library. Here you'll learn the basics of this amazing programming tool that will make you a more productive and capable JavaScript programmer. You'll learn how to select and manipulate page elements, add interaction by making page elements respond to your visitors, and add flashy visual effects and animations.
- **Part Three** covers jQuery's sister project, jQuery UI. jQuery UI is a JavaScript library of helpful "widgets" and effects. It makes adding common user interface elements like tabbed panels, dialog boxes, accordions, drop-down menus really easy. jQuery UI can help you build a unified-looking and stylish user interface for your next big web application.
- **Part Four** looks at some advanced uses of jQuery and JavaScript. In particular, Chapter 13 covers the technology that single-handedly made JavaScript one of the most glamorous web languages to learn. In this chapter, you'll learn how to use JavaScript to communicate with a web server so your pages can receive information and update themselves based on information provided by a web server—without having to load a new web page. Chapter 14 guides you step by step in creating a to-do list application using jQuery and jQuery UI.
- **Part Five** takes you past the basics, covering more complex concepts. You'll learn more about how to use jQuery effectively, as well as delve into advanced

JavaScript concepts. This part of the book also helps you when nothing seems to be working: when your perfectly crafted JavaScript program just doesn't seem to do what you want (or worse, it doesn't work at all!). You'll learn the most common errors new programmers make as well as techniques for discovering and fixing bugs in your programs.

At the end of the book, an appendix provides a detailed list of references to aid you in your further exploration of the JavaScript programming language.

The Very Basics

To use this book, and indeed to use a computer, you need to know a few basics. This book assumes that you're familiar with a few terms and concepts:

- **Clicking**. This book gives you three kinds of instructions that require you to use your computer's mouse or trackpad. To *click* means to point the arrow cursor at something on the screen and then—without moving the cursor at all—to press and release the clicker button on the mouse (or laptop trackpad). To *right-click* means to do the same thing with the right mouse button. To *double-click*, of course, means to click twice in rapid succession, again without moving the cursor at all. And to *drag* means to move the cursor *while* pressing the button.

TIP If you're on a Mac and don't have a right mouse button, you can accomplish the same thing by pressing the Control key as you click with the one mouse button.

When you're told to ⌘-*click* something on the Mac, or *Ctrl-click* something on a PC, you click while pressing the ⌘ or Ctrl key (both of which are near the space bar).

- **Menus**. The *menus* are the words at the top of your screen or window: File, Edit, and so on. Click one to make a list of commands appear, as though they're written on a window shade you've just pulled down.
- **Keyboard shortcuts**. If you're typing along in a burst of creative energy, it's sometimes disruptive to have to take your hand off the keyboard, grab the mouse, and then use a menu (for example, to use the Bold command). That's why many experienced computer mavens prefer to trigger menu commands by pressing certain combinations on the keyboard. For example, in the Firefox web browser, you can press Ctrl-+ (Windows) or ⌘-+ (Mac) to make text on a web page get larger (and more readable). When you read an instruction like "press ⌘-B," start by pressing the ⌘-key; while it's down, type the letter B, and then release both keys.
- **Operating system basics**. This book assumes that you know how to open a program, surf the Web, and download files. You should know how to use the Start menu (Windows) and the Dock or Apple menu (Macintosh), as well as the Control Panel (Windows), or System Preferences (Mac OS X).

If you've mastered this much information, you have all the technical background you need to enjoy *JavaScript & jQuery: The Missing Manual*.

About→These→Arrows

Throughout this book, and throughout the Missing Manual series, you'll find sentences like this one: "Open the System→Library→Fonts." That's shorthand for a much longer instruction that directs you to open three nested folders in sequence, like this: "On your hard drive, you'll find a folder called System. Open that. Inside the System folder window is a folder called Library; double-click it to open it. Inside *that* folder is yet another one called Fonts. Double-click to open it, too."

Similarly, this kind of arrow shorthand helps to simplify the business of choosing commands in menus, as shown in Figure I-2.

FIGURE I-2

In this book, arrow notations help simplify menu instructions. For example, View→Text Size→Increase is a more compact way of saying, "From the View menu, choose Text Size; from the submenu that then appears, choose Increase."

About the Online Resources

This book is designed to get your work onto the Web faster and more professionally; it's only natural, then, that much of the value of this book also lies on the Web. Online, you'll find example files so you can get some hands-on experience. You can also communicate with the Missing Manual team and tell us what you love (or hate) about the book. Head over to *www.missingmanuals.com*, or go directly to one of the following sections.

Living Examples

As you read the book's chapters, you'll encounter a number of *living examples*—step-by-step tutorials that you can build yourself, using raw materials (like graphics and

half-completed web pages) that you can download from either *https://github.com/sawmac/js3e* or from this book's Missing CD page at *www.missingmanuals.com/cds*. You might not gain very much from simply reading these step-by-step lessons while relaxing in your porch hammock, but if you take the time to work through them at the computer, you'll discover that these tutorials give you unprecedented insight into the way professional designers build web pages.

You'll also find, in this book's lessons, the URLs of the finished pages, so that you can compare your work with the final result. In other words, you won't just see pictures of JavaScript code in the pages of the book; you'll find the actual, working web pages on the Internet.

Registration

If you register this book at *http://oreilly.com*, you'll be eligible for special offers—like discounts on future editions of *JavaScript & jQuery: The Missing Manual*. Registering takes only a few clicks. To get started, type *www.oreilly.com/register* into your browser to hop directly to the Registration page.

Feedback

Got questions? Need more information? Fancy yourself a book reviewer? On our Feedback page, you can get expert answers to questions that come to you while reading, share your thoughts on this Missing Manual, and find groups for folks who share your interest in JavaScript and jQuery. To have your say, go to *www.missingmanuals.com/feedback.*

Errata

In an effort to keep this book as up to date and accurate as possible, each time we print more copies, we'll make any confirmed corrections you've suggested. We also note such changes on the book's website, so you can mark important corrections into your own copy of the book, if you like. Go to *http://tinyurl.com/jsjq3-mm* to report an error and view existing corrections.

Safari® Books Online

Safari® Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cellphone and mobile devices. Access new titles before they're available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

PART

1

Getting Started with JavaScript

CHAPTER

1

Writing Your First JavaScript Program

By itself, HTML doesn't have any smarts: It can't do math, it can't figure out if someone has correctly filled out a form, and it can't make decisions based on how a web visitor interacts with it. Basically, HTML lets people read text, look at pictures, watch videos, and click links to move to other web pages with more text, pictures, and videos. In order to add intelligence to your web pages so they can respond to your site's visitors, you need JavaScript.

JavaScript lets a web page react intelligently. With it, you can create smart web forms that let visitors know when they've forgotten to include necessary information. You can make elements appear, disappear, or move around a web page (see Figure 1-1). You can even update the contents of a web page with information retrieved from a web server—without having to load a new web page. In short, JavaScript lets you make your websites more engaging, effective, and useful.

NOTE Actually, HTML5 *does* add some smarts to HTML—including basic form validation. But because not all browsers support these nifty additions (and because you can do a whole lot more with forms and JavaScript), you still need JavaScript to build the best, most user-friendly and interactive forms. You can learn more about HTML5 and web forms in Ben Henick's *HTML5 Forms* (O'Reilly) and Gaurav Gupta's *Mastering HTML5 Forms* (Packt Publishing).

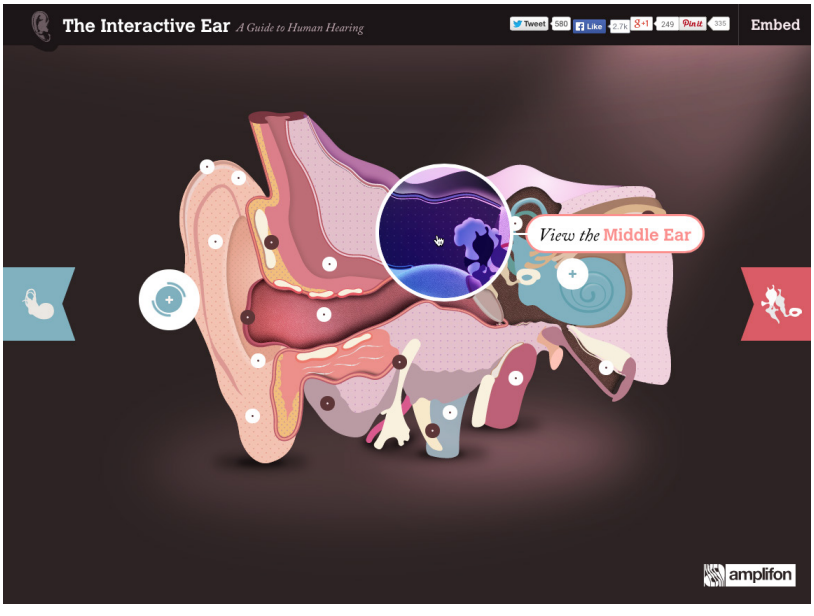

FIGURE 1-1

The Interactive Ear (http://www.amplifon.co.uk/interactive-ear/), an interactive guide to human hearing, lets visitors learn about and explore the different parts of the human ear. New information appears in response to mouse movements and clicks. With JavaScript, you can create your own interactive effects.

Introducing Programming

For a lot of people, the term "computer programming" conjures up visions of super-intelligent nerds hunched over keyboards, typing nearly unintelligible gibberish for hours on end. And, honestly, some programming is like that. Programming can seem like complex magic that's well beyond the average mortal. But many programming concepts aren't difficult to grasp, and as programming languages go, JavaScript is a good first language for someone new to programming.

Still, JavaScript is more complex than either HTML or CSS, and programming often is a foreign world to web designers; so one goal of this book is to help you think more like a programmer. Throughout this book, you'll learn fundamental programming concepts that apply whether you're writing JavaScript, ActionScript, or even writing a desktop program using C++. More importantly, you'll learn how to approach a programming task so you'll know exactly what you want to do before you start adding JavaScript to a web page.

Many web designers are immediately struck by the strange symbols and words used in JavaScript. An average JavaScript program is sprinkled with symbols ({ } [ ] ; , () !=) and full of unfamiliar words (`var`, `null`, `else if`). In many ways, learning

a programming language is a lot like learning another language. You need to learn new words, new punctuation, and understand how to put them together so you can communicate successfully.

Every programming language has its own set of keywords and characters, and its own set of rules for putting those words and characters together—the language's *syntax*. You'll need to memorize the words and rules of the JavaScript language (or at least keep this book handy as a reference). When learning to speak a new language, you quickly realize that placing an accent on the wrong syllable can make a word unintelligible. Likewise, a simple typo or even a missing punctuation mark can prevent a JavaScript program from working, or trigger an error in a web browser. You'll make plenty of mistakes as you start to learn to program—that's just the nature of programming.

At first, you'll probably find JavaScript programming frustrating—you'll spend a lot of your time tracking down errors you made when typing the script. Also, you might find some of the concepts related to programming a bit hard to follow at first. But don't worry: If you've tried to learn JavaScript in the past and gave up because you thought it was too hard, this book will help you get past the hurdles that often trip up folks new to programming. (And if you do have programming experience, this book will teach you JavaScript's idiosyncrasies and the unique concepts involved in programming for web browsers.)

In addition, this book isn't just about JavaScript—it's also about jQuery, the world's most popular JavaScript library. jQuery makes complex JavaScript programming easier…*much* easier. So with a little bit of JavaScript knowledge and the help of jQuery, you'll be creating sophisticated, interactive websites in no time.

FREQUENTLY ASKED QUESTION

Compiled vs. Scripting Languages

JavaScript is called a scripting language. *I've heard this term used for other languages like PHP and ColdFusion as well. What's a scripting language?*

Most of the programs running on your computer are written using languages that are *compiled*. Compiling is the process of creating a file that will run on a computer by translating the code a programmer writes into instructions that a computer can understand. Once a program is compiled, you can run it on your computer, and because a compiled program has been converted directly to instructions a computer understands, it will run faster than a program written with a scripting language. Unfortunately, compiling a program is a time-consuming process: You have to write the program, compile it, and then test it. If the program doesn't work, you have to go through the whole process again.

A scripting language, on the other hand, is only compiled when an *interpreter* (another program that can convert the script into something a computer can understand) reads it. In the case of JavaScript, the interpreter is built into the web browser. So when your web browser reads a web page with a JavaScript program in it, the web browser translates the JavaScript into something the computer understands. As a result, a scripting language operates more slowly than a compiled language, because every time it runs, the program must be translated for the computer. Scripting languages are great for web developers: Scripts are generally much smaller and less complex than desktop programs, so the lack of speed isn't as important. In addition, because they don't require compiling, creating and testing programs that use a scripting language is a much faster process.

What's a Computer Program?

When you add JavaScript to a web page, you're writing a computer program. Granted, most JavaScript programs are much simpler than the programs you use to read email, retouch photographs, and build web pages. But even though JavaScript programs (also called *scripts*) are simpler and shorter, they share many of the same properties of more complicated programs.

In a nutshell, any computer program is a series of steps that are completed in a designated order. Say you want to display a welcome message using the web-page visitor's name: "Welcome, Bob!" There are several things you'd need to do to accomplish this task:

1. **Ask the visitor's name.**
2. **Get the visitor's response.**
3. **Print (that is, display) the message on the web page.**

While you may never want to print a welcome message on a web page, this example demonstrates the fundamental process of programming: Determine what you want to do, then break that task down into individual steps. Every time you want to create a JavaScript program, you must go through the process of determining the steps needed to achieve your goal. Once you know the steps, you'll translate your ideas into programming *code*—the words and characters that make the web browser behave how you want it to.

How to Add JavaScript to a Page

Web browsers are built to understand HTML and CSS and convert those languages into a visual display on the screen. The part of the web browser that understands HTML and CSS is called the *layout* or *rendering* engine. But most browsers also have something called a *JavaScript interpreter*. That's the part of the browser that understands JavaScript and can execute the steps of a JavaScript program. The web browser is usually expecting HTML, so you must specifically tell the browser when JavaScript is coming by using the `<script>` tag.

The `<script>` tag is regular HTML. It acts like a switch that in effect says "Hey, web browser, here comes some JavaScript code; you don't know what to do with it, so hand it off to the JavaScript interpreter." When the web browser encounters the closing `</script>` tag, it knows it's reached the end of the JavaScript program and can get back to its normal duties.

Much of the time, you'll add the `<script>` tag in the web page's `<head>` section, like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
html4/strict.dtd">
<html>
```

```
<head>
<title>My Web Page</title>
<script type="text/javascript">
</script>
</head>
```

The <script> tag's type attribute indicates the format and the type of script that follows. In this case, type="text/javascript" means the script is regular text (just like HTML) and that it's written in JavaScript.

If you're using HTML5, life is even simpler. You can skip the type attribute entirely:

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>My Web Page</title>
<script>
</script>
</head>
```

In fact, web browsers let you leave out the type attribute in HTML 4.01 and XHTML 1.0 files as well—the script will run the same; however, your page won't validate correctly without the type attribute (see the box on page xix for more on validation). This book uses HTML5 for the doctype, but the JavaScript code will be the same and work the same for HTML 4.01, and XHTML 1.

You then add your JavaScript code between the opening and closing <script> tags:

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>My Web Page</title>
<script>
  alert('hello world!');
</script>
</head>
```

You'll find out what this JavaScript does in a moment. For now, turn your attention to the opening and closing <script> tags. To add a script to your page, start by inserting these tags. In many cases, you'll put the <script> tags in the page's <head> in order to keep your JavaScript code neatly organized in one area of the web page.

However, it's perfectly valid to put <script> tags anywhere inside the page's HTML. In fact, as you'll see later in this chapter, there's a JavaScript command that lets you write information directly into a web page. Using that command, you place the <script> tags in the location on the page (somewhere inside the body) where you want the script to write its message. In fact, it's common to put <script> tags just

below the closing </body> tag—this approach makes sure the page is loaded and the visitor sees it before running any JavaScript.

UP TO SPEED

The Client Side vs. the Server Side

JavaScript was originally created as a *client-side* language. Client-side JavaScript is delivered to web browsers by a web server. The people visiting your site download your web page and its JavaScript, and then their web browser—the client—processes the JavaScript and makes the magic happen.

An alternative type of web programming language is called a *server-side* language, which you'll find in pages built around PHP, .NET, ASP, ColdFusion, Ruby on Rails, and other web server technologies. Server-side programming languages, as the name suggests, run on a web server. They can exhibit a lot of intelligence by accessing databases, processing credit cards, and sending email around the globe. The problem with server-side languages is that they require the web browser to send requests to the web server, forcing visitors to wait until a new page arrives with new information.

Client-side languages, on the other hand, can react immediately and change what a visitor sees in his web browser without the need to download a new page. Content can appear or disappear, move around the screen, or automatically update based on how a visitor interacts with the page. This responsiveness lets you create websites that feel more like desktop programs than static web pages. But JavaScript isn't the only client-side technology in town. You can also use plug-ins to add programming smarts to a web page. Java applets are one example. These are small programs, written in the Java programming language, that run in a web browser. They also tend to start up slowly and have been known to crash the browser.

Flash is another plug-in based technology that offers sophisticated animation, video, sound, and lots of interactive potential. In fact, it's sometimes hard to tell if an interactive web page is using JavaScript or Flash. For example, Google Maps could also be created in Flash (in fact, Yahoo! Maps was at one time a Flash application, until Yahoo! re-created it using JavaScript). A quick way to tell the difference: Right-click on the part of the page that you think might be Flash (the map itself, in this case); if it is, you'll see a pop-up menu that includes "About the Flash Player."

Ajax, which you'll learn about in Part Four of this book, brings the client side and server side together. Ajax is a method for using JavaScript to talk to a server, retrieve information from the server, and update the web page without the need to load a new web page. Google Maps uses this technique to let you move around a map without forcing you to load a new web page.

These days, JavaScript is finding a lot of use outside of the web browser. Node.js is a server-side version of JavaScript that can connect to databases, access the web server's filesystem, and perform many other tasks on a web server. This book doesn't discuss that aspect of JavaScript programming, but for a great video introduction to Node.js, check out *www.youtube.com/watch?v=hKQr2DGJjUQ/*.

In addition, some relatively new databases even use JavaScript as the language for creating, retrieving, and updating database records. MongoDB and CouchDB are two popular examples. You may hear the term full-stack JavaScript, which means using JavaScript as the language for the client-side browser, the web server, and database control. One language to rule them all!

External JavaScript Files

Using the `<script>` tag as discussed in the previous section lets you add JavaScript to a single web page. But many times you'll create scripts that you want to share with all of the pages on your site. For example, you might add a panel of additional navigation options that slides onto the page in response to a visitor's mouse movements (see Figure 1-2). You'll want that same fancy slide-in panel on every page

of your site, but copying and pasting the same JavaScript code into each page is a really bad idea for several reasons.

First, it's a lot of work copying and pasting the same code over and over again, especially if you have a site with hundreds of pages. Second, if you ever decide to change or enhance the JavaScript code, you'll need to locate every page using that JavaScript and update the code. Finally, because all of the code for the JavaScript program would be located in every web page, each page will be that much larger and slower to download.

A better approach is to use an external JavaScript file. If you've used external CSS files for your web pages, this technique should feel familiar. An external JavaScript file is a text file containing JavaScript code and ending with the file extension *.js*—*navigation.js*, for example. The file is linked to a web page using the `<script>` tag. For example, to add this JavaScript file to your home page, you might write the following:

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>My Web Page</title>
<script src="navigation.js"></script>
</head>
```

The `src` attribute of the `<script>` tag works just like the `src` attribute of an `<img>` tag, or an `<a>` tag's `href` attribute. In other words, it points to a file either in your website or on another website (see the box on page 11).

NOTE When adding the `src` attribute to link to an external JavaScript file, don't add any JavaScript code between the opening and closing `<script>` tags. If you want to link to an external JavaScript file and add custom JavaScript code to a page, use a second set of `<script>` tags. For example:

```
<script src="navigation.js"></script>
<script>
  alert('Hello world!');
</script>
```

You can (and often will) attach multiple external JavaScript files to a single web page. For example, you might have created one external JavaScript file that controls a drop-down navigation panel, and another that lets you add a nifty slideshow to a page of photos. On your photo gallery page, you'd want to have both JavaScript programs, so you'd attach both files.

In addition, you can attach external JavaScript files and add a JavaScript program to the same page like this:

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>My Web Page</title>
<script src="navigation.js"></script>
<script src="slideshow.js"></script>
<script>
  alert('hello world!');
</script>
</head>
```

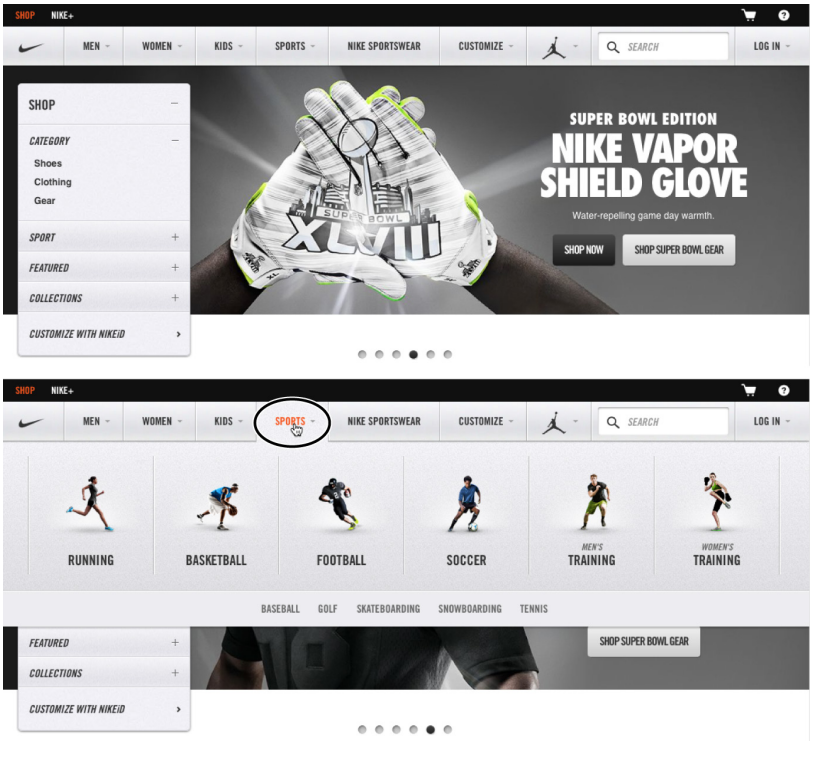

FIGURE 1-2

Nike.com's website uses JavaScript extensively to create a compelling showcase for their products. The home page (top) includes a row of navigation buttons along the top—Men, Women, Kids, and so on—that, when moused over, reveal a panel of additional navigation options. For example, mousing over the Sports button (circled in bottom image) reveals a panel listing different sports that Nike makes products for.

Just remember that you must use one set of opening and closing `<script>` tags for each external JavaScript file. You'll create an external JavaScript file in the tutorial that starts on page 15.

You can keep external JavaScript files anywhere inside your website's root folder (or any subfolder inside the root). Many web developers create a special directory for external JavaScript files in the site's root folder: common names are *js* (meaning JavaScript) or *libs* (meaning libraries).

UP TO SPEED

URL Types

When attaching an external JavaScript file to a web page, you need to specify a *URL* for the `src` attribute of the `<script>` tag. A URL or *Uniform Resource Locator* is a path to a file—like another web page, a graphic, or a JavaScript file—located on the Web. There are three types of paths: *absolute path, root-relative path*, and *document-relative path*. All three indicate where a web browser can find a particular file.

An *absolute path* is like a postal address–it contains all the information needed for a web browser located anywhere in the world to find the file. An absolute path includes *http://*, the hostname, and the folder and name of the file. For example: *http://www.uptospeedguides.com/scripts/site.js*.

A *root-relative* path indicates where a file is located relative to a site's top-level folder–the site's root folder. A root-relative path doesn't include *http://* or the domain name. It begins with a / (slash) indicating the site's root folder–the folder the home page is in. For example, */scripts/site.js* indicates that the file *site.js* is located inside a folder named *scripts*, which is itself located in the site's top-level folder. An easy way to create a root-relative path is to take an absolute path and strip off the *http://* and the host name. For example, *http://www.uptospeedguides.com/index.html* written as a root-relative URL is */index.html*.

A *document-relative* path specifies the path from the web page to the JavaScript file. If you have multiple levels of folders on your website, you'll need to use different paths to point to the same JavaScript file. For example, suppose you have a JavaScript file named *site.js* located in a folder named *scripts* in your website's main directory. The document-relative path to that file will look one way for the home page–*scripts/site.js*–but for a page located inside a folder named *about*, the path to the same file would be different; *../scripts/site.js*–the *../* means climb up *out* of the *about* folder, while the */scripts/site.js* means go to the *scripts* folder and get the file *site.js*.

Here are some tips on which URL type to use:

- If you're pointing to a file that's not on the same server as the web page, you *must* use an absolute path. It's the only type that can point to another website.
- Root-relative paths are good for JavaScript files stored on your own site. Because they always start at the root folder, the URL for a JavaScript file will be the same for every page on your website, even when web pages are located in folders and subfolders on your site. However, root-relative paths don't work unless you're viewing your web pages through a web server—either your web server out on the Internet, or a web server you've set up on your own computer for testing purposes. In other words, if you're just opening a web page off your computer using the browser's File→Open command, the web browser won't be able to locate, load, or run JavaScript files that are attached using a root-relative path.
- Document-relative paths are the best when you're designing on your own computer without the aid of a web server. You can create an external JavaScript file, attach it to a web page, and then check the JavaScript in a web browser simply by opening the web page off your hard drive. Document-relative paths work fine when moved to your actual, living, breathing website on the Internet, but you'll have to rewrite the URLs to the JavaScript file if you move the web page to another location on the server. This book uses document-relative paths, which will let you follow along and test the tutorials on your own computer without a web server.

NOTE Sometimes the order in which you attach external JavaScript files matters. As you'll see later in this book, sometimes scripts you write depend upon code that comes from an external file. That's often the case when using JavaScript libraries (JavaScript code that simplifies complex programming tasks). You'll see an example of a JavaScript library in action in the tutorial on page 16.

Your First JavaScript Program

The best way to learn JavaScript programming is by actually programming. Throughout this book, you'll find hands-on tutorials that take you step by step through the process of creating JavaScript programs. To get started, you'll need a text editor (see page xxii for recommendations), a web browser, and the exercise files located at *https://github.com/sawmac/js3e* (see the following Note for complete instructions).

NOTE The tutorials in this chapter require the example files from this book's website, *www.missingmanuals.com/cds/jsjq3emm*. (The tutorial files are stored as a single Zip file.)

In Windows, download the Zip file and double-click it to open the archive. Click the Extract All Files option, and then follow the instructions of the Extraction Wizard to unzip the files and place them on your computer. If you have trouble opening the Zip file, the free 7-Zip utility can help: *www.7-zip.org*.

On a Mac, simply double-click the file to decompress it. After you've downloaded and decompressed the files, you should have a folder named *MM_JAVASCRIPT3E* on your computer, containing all of the tutorial files for this book.

To get your feet wet and provide a gentle introduction to JavaScript, your first program will be very simple:

1. **In your favorite text editor, open the file *hello.html*.**

   This file is located in the *chapter01* folder in the *MM_JAVASCRIPT3E* folder you downloaded as described in the note above. It's a very simple HTML page, with an external cascading style sheet to add a little visual excitement.

2. **Click in the empty line just *before* the closing </head> tag and type:**

   ```
   <script>
   ```

   This code is actually HTML, not JavaScript. It informs the web browser that the stuff following this tag is JavaScript.

3. **Press the Return key to create a new blank line, and type:**

   ```
   alert('hello world');
   ```

   You've just typed your first line of JavaScript code. The JavaScript `alert()` function is a command that pops open an Alert box and displays the message that appears inside the parentheses—in this case, *hello world*. Don't worry about all of the punctuation (the parentheses, quotes, and semicolon) just yet. You'll learn what they do in the next chapter.

4. **Press the Return key once more, and type `</script>`. The code should now look like this:**

```
<link href="../_css/site.css" rel="stylesheet">
<script>
  alert('hello world');
</script>
</head>
```

In this example, the stuff you just typed is shown in boldface. The two HTML tags are already in the file; make sure you type the code exactly where shown.

5. **Launch a web browser and open the *hello.html* file to preview it.**

A JavaScript Alert box appears (see Figure 1-3). Notice that the page is blank when the alert appears. (If you don't see the Alert box, you probably mistyped the code listed in the previous steps. Double-check your typing and read the following Tip.)

TIP When you first start programming, you'll be shocked at how often your JavaScript programs don't seem to work...at all. For new programmers, the most common cause of nonfunctioning programs is simple typing mistakes. Always double-check to make sure you spelled commands (like `alert` in the first script) correctly. Also, notice that punctuation frequently comes in pairs (the opening and closing parentheses, and single-quote marks from your first script, for example). Make sure you include both opening and closing punctuation marks when they're required.

6. **Click the Alert box's OK button to close it.**

When the Alert box disappears, the web page appears in the browser window.

Although this first program isn't earth-shatteringly complex (or even that interesting), it does demonstrate an important concept: A web browser will run a JavaScript program the moment it reads in the JavaScript code. In this example, the `alert()` command appeared *before* the web browser displayed the web page, because the JavaScript code appeared *before* the HTML in the <body> tag. This concept comes into play when you start writing programs that manipulate the HTML of the web page—as you'll learn in Chapter 3.

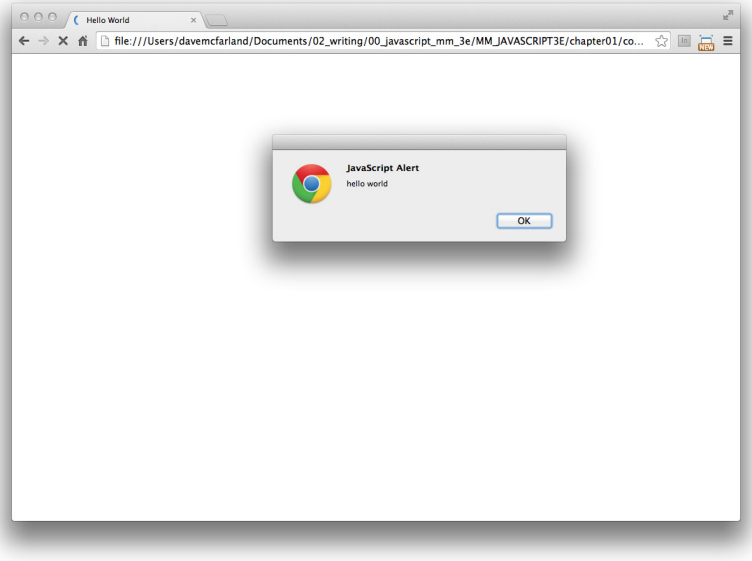

FIGURE 1-3
The JavaScript Alert box is a quick way to grab someone's attention. It's one of the simplest JavaScript commands to learn and use.

NOTE Some versions of Internet Explorer (IE) don't like to run JavaScript programs in web pages that you open directly off your hard drive, for fear that the code might do something harmful. So when you try to preview the tutorial files for this book in Internet Explorer, you might see a message saying that IE has blocked the script. Click "Allow blocked content."

This annoying behavior only applies to web pages you preview from your computer, not to files you put up on a web server. To avoid hitting the "Allow blocked content" button over and over, preview pages in a different web browser, like Chrome or Firefox.

Writing Text on a Web Page

The script in the previous section popped up a dialog box in the middle of your monitor. What if you want to print a message directly onto a web page using JavaScript? There are many ways to do so, and you'll learn some sophisticated techniques later in this book. However, you can achieve this simple goal with a built-in JavaScript command, and that's what you'll do in your second script:

1. **In your text editor, open the file *hello2.html*.**

   While `<script>` tags usually appear in a web page's `<head>`, you can put them and JavaScript programs directly in the page's body.

2. **Directly below `<h1>Writing to the document window</h1>`, type the following code:**

   ```
   <script>
   document.write('<p>Hello world!</p>');
   </script>
   ```

   Like the `alert()` function, `document.write()` is a JavaScript command that literally writes out whatever you place between the opening and closing parentheses. In this case, the HTML `<p>Hello world!</p>` is added to the page: a paragraph tag and two words.

3. **Save the page and open it in a web browser.**

   The page opens and the words "Hello world!" appear below the headline (see Figure 1-4).

NOTE The tutorial files you downloaded also include the completed version of each tutorial. If you can't seem to get your JavaScript working, compare your work with the file that begins with *complete_* in the same folder as the tutorial file. For example, the file *complete_hello2.html* contains a working version of the script you added to file *hello2.html*.

The two scripts you just created may leave you feeling a little underwhelmed with JavaScript...or this book. Don't worry—this is only the beginning. It's important to start out with a full understanding of the basics. You'll be doing some very useful and complicated things using JavaScript in just a few chapters. In fact, in the remainder of this chapter you'll get a taste of some of the advanced features you'll be able to add to your web pages after you've worked your way through the first two parts of this book.

Attaching an External JavaScript File

As discussed on page 8, you'll usually put JavaScript code in a separate file if you want to use the same scripts on more than one web page. You then instruct your web pages to load that file and use the JavaScript inside it. External JavaScript files also come in handy when you're using someone else's JavaScript code. In particular, there are collections of JavaScript code called *libraries*, which provide useful JavaScript programming. Usually, these libraries make it easy to do something that's normally quite difficult. You'll learn more about JavaScript libraries on page 105, and, in particular, the JavaScript library this book (and much of the Web) uses—jQuery.

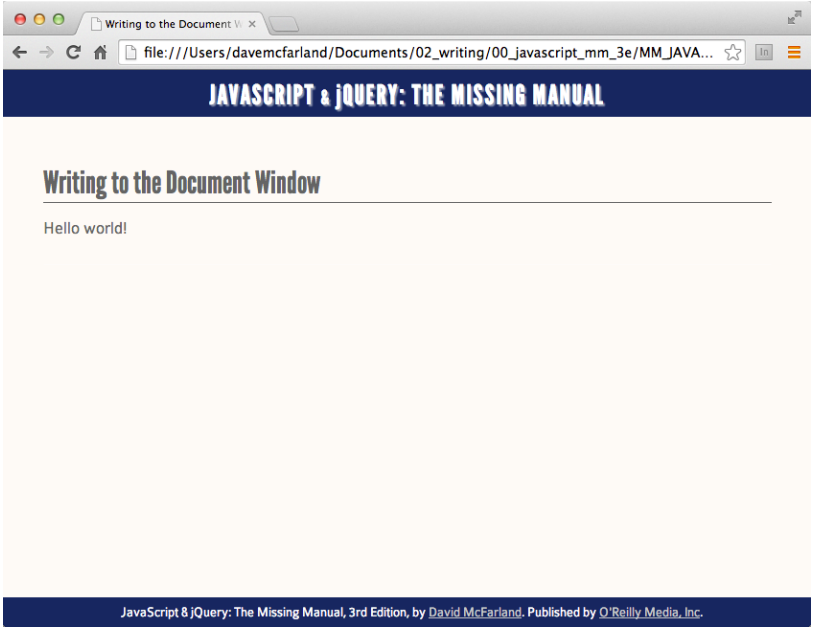

FIGURE 1-4
Wow. This script may not be something to "document.write" home about—ha, ha, JavaScript humor—but it does demonstrate that you can use JavaScript to add content to a web page, a trick that comes in handy when you want to display messages (like "Welcome back to the site, Dave") after a web page has downloaded.

But for now, you'll get experience attaching an external JavaScript file to a page, and writing a short program that does something cool:

1. **In your text editor, open the file *fadeIn.html*.**

   This page contains just some simple HTML—a few `<div>` tags, a headline, and a couple of paragraphs. You'll be adding a simple visual effect to the page, which causes all of the content to slowly fade into view.

2. **Click in the blank line between the `<link>` and closing `</head>` tags near the top of the page, and type:**

   ```
   <script src="../_js/jquery.min.js"></script>
   ```

   This code links a file named *jquery.min.js*, which is contained in a folder named *_js*, to this web page. When a web browser loads this web page, it also downloads the *jquery.min.js* JavaScript file and runs the code inside it.

   Next, you'll add your own JavaScript programming to this page.

NOTE The min part means that the file is *minimized*—a process that removes unneeded whitespace and condenses the code to make the file smaller so that it downloads faster.

3. **Press Return to create a new blank line, and then type:**

```
<script>
```

HTML tags usually travel in pairs—an opening and closing tag. To make sure you don't forget to close a tag, it helps to close the tag immediately after typing the opening tag, and then fill in the stuff that goes between the tags.

4. **Press Return twice to create two blank lines, and then type:**

```
</script>
```

This ends the block of JavaScript code. Now you'll add some programming.

5. **Click the empty line between the opening and closing script tags and type:**

```
$(document).ready(function() {
```

You're probably wondering what the heck that is. You'll find out all the details of this code on page 112, but in a nutshell, this line takes advantage of the programming that's inside the *jquery.min.js* file to make sure that the browser executes the next line of code at the right time.

6. **Hit return to create a new line, and then type:**

```
$('header').hide().slideDown(3000);
```

This line does something magical: It makes the "JavaScript & jQuery The Missing Manual" header first disappear and then slowly slide down onto the page over the course of 3 seconds (or 3,000 milliseconds). How does it do that? Well, that's part of the magic of jQuery, which makes complex effects possible with just a single line of code.

7. **Hit Return one last time, and then type:**

```
});
```

This code closes up the JavaScript code, much as a closing `</script>` tag indicates the end of a JavaScript program. Don't worry too much about all those weird punctuation marks—you'll learn how they work in detail later in the book. The main thing you need to make sure of is to type the code exactly as it's listed here. One typo, and the program may not work.

The final code you added to the page should look like the bolded text in the following:

```
<link href="../_css/site.css" rel="stylesheet">
<script src="../_js/jquery.min.js"></script>
<script>
$(document).ready(function() {
  $('header').hide().slideDown(3000);
});
</script>
</head>
```

TIP To make your programming easier to read, it's a good idea to indent code. Much as you indent HTML tags to show which tags are nested inside of other tags, you can indent lines of code that are inside another block of code. For example, the line of code you added in step 6 is nested inside the code for steps 5 and 7, so hitting Tab or pressing the spacebar a couple of times before typing the code for step 6 can make your code easier to understand (as pictured in the final code listed at the end of step 7).

8. **Save the HTML file, and open it in a web browser.**

   You should see the headline—Sliding Down—plus a paragraph and the footer at the bottom of the browser window, followed by the boxes containing "JavaScript & jQuery: The Missing Manual" slowly slide down into place. Change the number 3000 to different values (like 250 and 10000) to see how that changes the way the page works.

NOTE If you try to preview this page in Internet Explorer and it doesn't seem to do anything, you'll need to click the "Enable blocked content" box that appears at the bottom of the page (see the Note on page 14).

As you can see, it doesn't take a whole lot of JavaScript to do some amazing things to your web pages. Thanks to jQuery, you'll be able to create sophisticated, interactive websites even if you're not a programming wizard. However, you'll find it helps to know the basics of JavaScript and programming. Chapters 2 and 3 will cover the basics of JavaScript to get you comfortable with the fundamental concepts and syntax that make up the language.

Tracking Down Errors

The most frustrating moment in JavaScript programming comes when you try to view your JavaScript-powered page in a web browser...and nothing happens. It's one of the most common experiences for programmers. Even experienced programmers often don't get it right the first time they write a program, so figuring out what went wrong is just part of the game.

Most web browsers are set up to silently ignore JavaScript errors, so you usually won't even see a "Hey, this program doesn't work!" dialog box. (Generally, that's a good thing, as you don't want a JavaScript error to interrupt the experience of viewing your web pages.)

So how do you figure out what's gone wrong? There are many ways to track errors in a JavaScript program. You'll learn some advanced *debugging* techniques in Chapter 17, but the most basic method is to consult the web browser. Most web browsers keep track of JavaScript errors and record them in a separate window called an *error console*. When you load a web page that contains an error, you can then view the console to get helpful information about the error, like which line of the web page it occurred in and a description of the error.

Often, you can find the answer to the problem in the error console, fix the JavaScript, and then the page will work. The console helps you weed out the basic typos you make when you first start programming, like forgetting closing punctuation, or mistyping the name of a JavaScript command. You can use the error console in your favorite browser, but because scripts sometimes work in one browser and not another, this section shows you how to turn on the JavaScript console in all major browsers, so you can track down problems in each.

The Chrome JavaScript Console

Google's Chrome browser is beloved by many a web developer. Its DevTools feature gives you many ways to troubleshoot HTML, CSS, and JavaScript problems. Also, its JavaScript console is a great place to begin tracking down errors in your code. It not only describes the errors it finds, it also identifies the line in your code where each error occurred.

To open the JavaScript console, click the Customize menu button (circled in Figure 1-5) and choose Tools→JavaScript Console. Or use the keyboard shortcut Ctrl+Shift+J (Windows) or ⌘-Option-J (Mac).

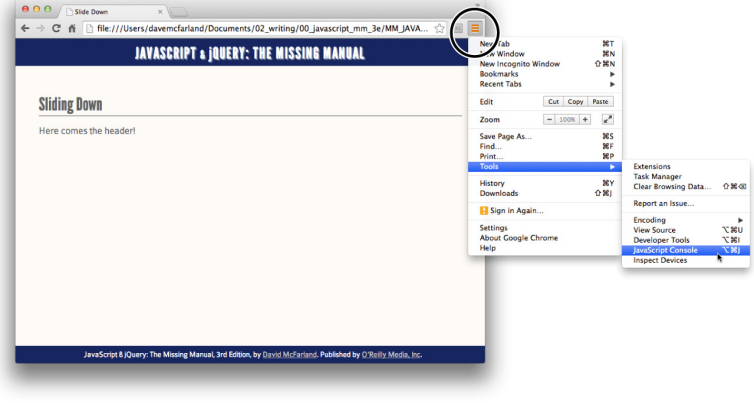

FIGURE 1-5

Click the Customize menu (circled) to access the JavaScript console as well as other helpful tools. Choosing the Developer Tools option is another way to get to the console, as the JavaScript console is part of a larger set of Chrome tools called the Developer Tools (DevTools for short). You'll learn more about those tools in Chapter 17.

After you open the console, you can examine any errors that appear in the current page. For example, in Figure 1-6, the console identifies the error as an "Uncaught SyntaxError: Unexpected token ILLEGAL." OK, it may not be immediately obvious what that means, but as you encounter (and fix) more errors you'll get used to these terse descriptions. Basically, a syntax error points to some kind of typographical error—an error with the syntax or language of the program. The "Unexpected token ILLEGAL" part just means that the browser has encountered an illegal character, or (and here's the tricky part) that there's a missing character. In this case, looking closely at the code you can see there's an opening single quote mark before "slow" but no final quote mark.

The console also identifies the name of the file the error is in (*complete_slide.html*, in this case) and the line number the error occurs (line 10). Click the filename, and Chrome opens the file above the console and briefly highlights the line (see Figure 1-5).

TIP Because the error console displays the line number where the error occurred, you may want to use a text editor that can show line numbers. That way, you can easily jump from the error console to your text editor and identify the line of code you need to fix.

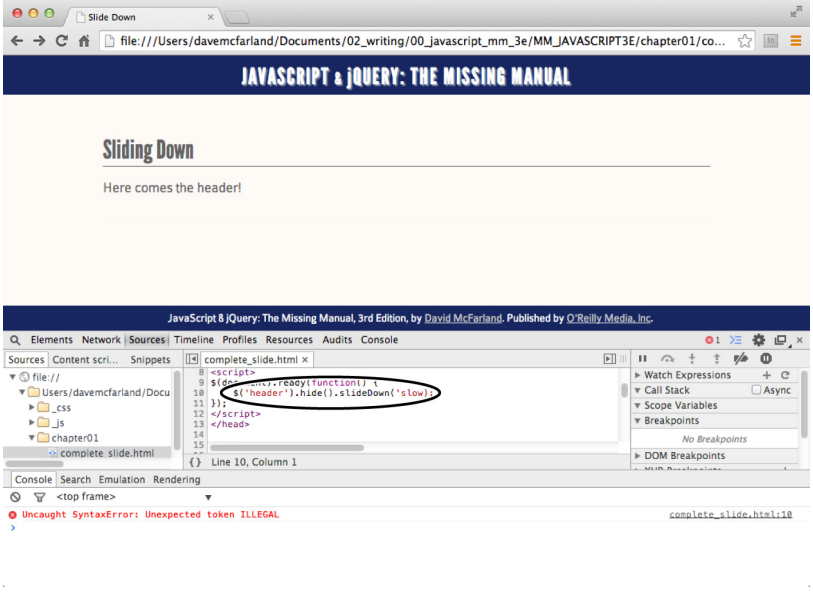

FIGURE 1-6
Chrome's JavaScript console identifies errors in your programs. Click the filename listed to the right of the error, and Chrome briefly highlights the page with the error (circled).

Unfortunately, there's a long list of things that can go wrong in a script, from simple typos to complex errors in logic. When you're just starting out with JavaScript programming, many of your errors will be the simple typographic sort. For example, you might forget a semicolon, quote mark, or parenthesis, or misspell a JavaScript command. You're especially prone to typos when following examples from a book (like this one). Here are a few common mistakes you might make and the (not-so obvious) error messages you may encounter:

- **Missing punctuation.** As mentioned earlier, JavaScript programming often involves lots of symbol pairs like opening and closing parentheses and brackets. For example, if you type `alert('hello';`—leaving off the closing parenthesis—you'll probably get the: "Unexpected token ;" message, meaning that Chrome was expecting something other than the character it's showing. In this case, it encountered the semicolon instead of the closing parenthesis.

- **Missing quote marks**. A *string* is a series of characters enclosed by quote marks (you'll learn about these in greater detail on page 27). For example, *'hello'* is a string in the code `alert('hello');`. It's easy to forget either the opening or closing quote mark. It's also easy to mix up those quote marks; for instance, by pairing a single-quote with a double quote like this: `alert('hello");`. In either case, you'll probably see an "Uncaught SyntaxError: Unexpected token ILLEGAL" error.
- **Misspelling commands**. If you misspell a JavaScript command—`aler('hello');`—you'll get an error saying that the misspelled command isn't defined: for example, "Uncaught ReferenceError: aler is not defined ," if you misspell the `alert` command. You'll also encounter problems when you misspell jQuery functions (like the `.hide()` and `.slideDown()` functions in the previous tutorial). In this case, you'll get a different error. For example, if you mistyped "hide" as "hid" in step 6 on page 17, Chrome will give you this error: "Uncaught TypeError: Object [object Object] has no method 'hid'".
- **Syntax error**. Occasionally, Chrome has no idea what you were trying to do and provides this generic error message. A syntax error represents some mistake in your code. It may not be a typo, but you may have put together one or more statements of JavaScript in a way that isn't allowed. In this case, you need to look closely at the line where the error was found and try to figure out what mistake you made. Unfortunately, these types of errors often require experience with and understanding of the JavaScript language to fix.

As you can see from the preceding list, many errors you'll make simply involve forgetting to type one of a pair of punctuation marks—like quote marks or parentheses. Fortunately, these are easy to fix, and as you get more experience programming, you'll eventually stop making them almost completely (no programmer is perfect).

The Internet Explorer Console

Internet Explorer provides a sophisticated set of developer tools for not only viewing JavaScript errors, but also analyzing CSS, HTML, and transfers of information over the network. When open, the developer tool window appears in the bottom half of the browser window. Press the F12 key to open the developer tools, and press it again to close them. You'll find JavaScript errors listed under the Console tab (circled in Figure 1-7).

NOTE If you first open a web page and then open the Internet Explorer console, you won't see any errors (even if there are some). You need to reload the page to see any errors. Once the console is open, you'll see errors on the pages you visit as they load.

IE's Console displays error messages similar to those described earlier for Chrome. However, sometimes they're very different. For example, IE's "Unterminated string constant" is an "Unexpected token ILLEGAL" error in Chrome. Like Chrome, Internet Explorer identifies the line of code in the HTML file where the error occurred, which you can click to see the actual code where the error occurs.

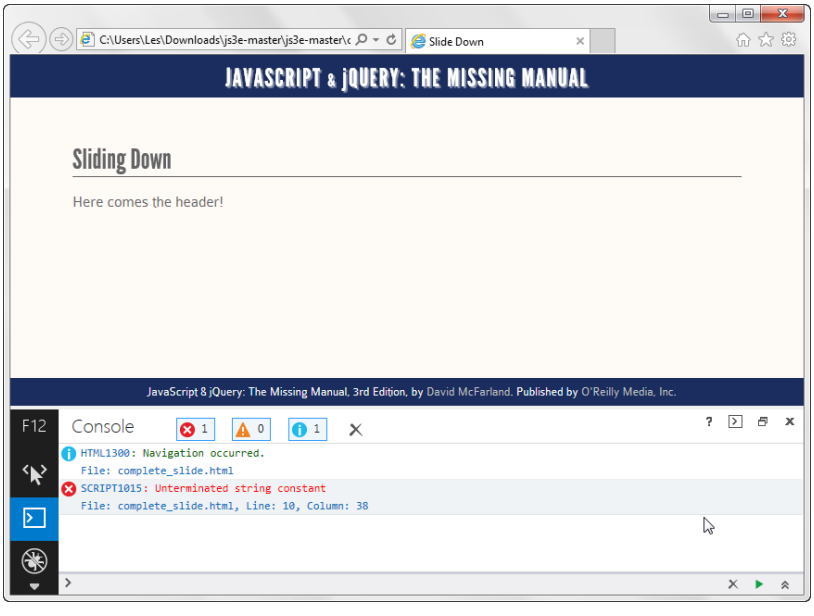

FIGURE 1-7
The Internet Explorer developer tools provide access to JavaScript errors that occur on a page, as well as a whole lot of other information.

The Firefox JavaScript Web Console

Mozilla's Firefox browser also gives you a console to view JavaScript errors. To open the JavaScript console, on Windows click the Firefox tab in the top left of the browser window and choose Web Developer→Web Console. On a Mac, select Tools→Web Developer→Web Console. Or use the keyboard shortcuts Ctrl+Shift+I (Windows) or ⌘-Option-K (Mac).

Once the console opens, you'll see any JavaScript errors on the page. Unfortunately, Firefox's Web Console is more like a fire hose of data than a simple JavaScript error reporter (Figure 1-8). That's because it provides information on all sorts of things: files downloaded, CSS and HTML errors, and more.

NOTE The Firebug plug-in (*http://getfirebug.com*) greatly expands on Firefox's Error Console. In fact, it provided the model for the developer tools in Internet Explorer, Chrome, and Safari (discussed next).

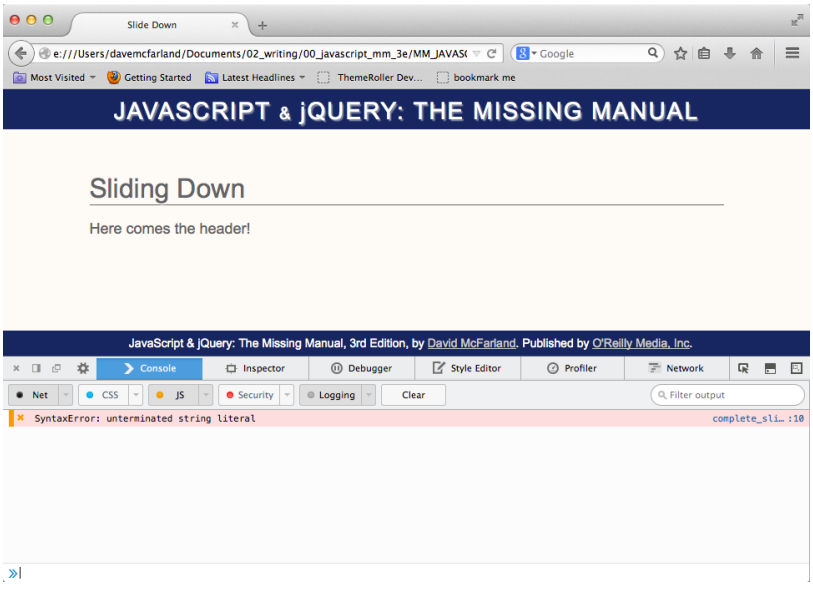

FIGURE 1-8

If you don't want to see all of the messages in Firefox's Web Console, just click the button for the type of message you wish to hide. For example, click the CSS button to hide CSS error messages, the Security button to hide security warnings, and so on. You'll know if the button is disabled because it looks lighter gray, like the CSS and Security buttons here. A button is enabled when it's darker and looks like it has been pressed "in," like the Net, JS (short for JavaScript), and Logging buttons here.

The Safari Error Console

Safari's error console is available from the Develop menu: Develop→Show Error Console (or, if you're on a Mac, use the Option-⌘-C keyboard shortcut). However, the Develop menu isn't normally turned on when Safari is installed, so there are a couple of steps to get to the JavaScript console.

To turn on the Develop menu, you need to first access the Preferences window. Choose Safari→Preferences. Once the Preferences window opens, click the Advanced button. Turn on the "Show Develop menu in menu bar" box and close the Preferences window.

When you restart Safari, the Develop menu will appear between the Bookmarks and Window menus in the menu bar at the top of the screen. Select Develop→Show Error Console to open the console (see Figure 1-9).

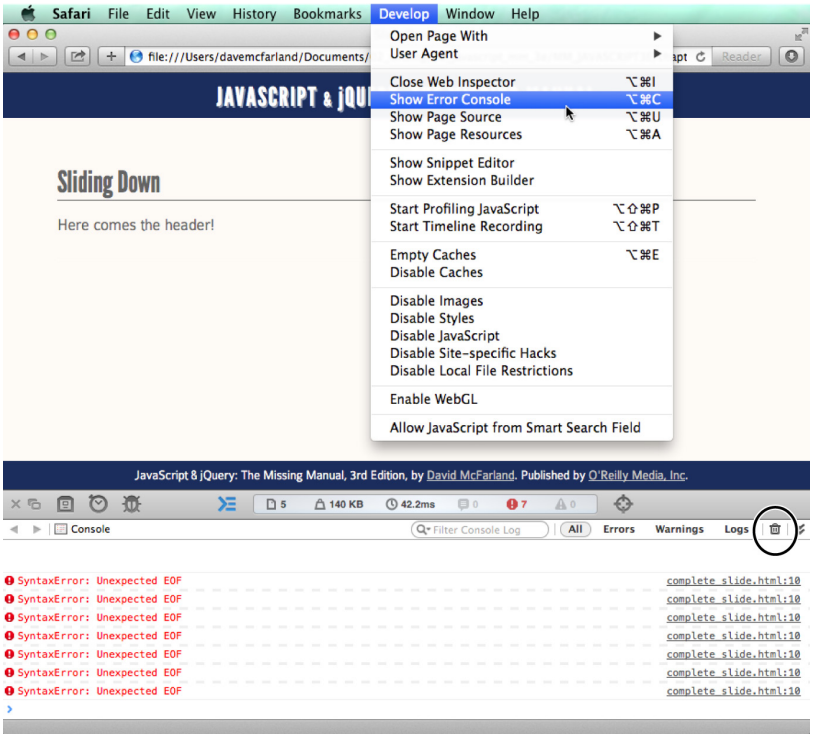

FIGURE 1-9

The Safari Error Console displays the name of the JavaScript error, the filename (and location), and the line on which Safari encountered the error. Each tab or browser window has its own error console, so if you've already opened the console for one tab, you need to choose Develop⟶Show Error Console again if you wish to see an error for another tab or window. In addition, if you reload a page, Safari doesn't clear any prior errors on that page, so you can end up with a long list of old, fixed errors as you work on a page and reload it. The answer: click the Trash icon (circled) to remove the list of old errors, and then reload the page.

NOTE If you're on Windows, you may have an old version of the Safari browser. Apple has stopped updating Safari for Windows, so the Safari information shown here may not apply to you.

CHAPTER

2

The Grammar of JavaScript

Learning a programming language is a lot like learning any new language: you need to learn new words and punctuation, as well as master a new set of rules. And just as you need to learn the grammar of French to speak French, you must become familiar with the grammar of JavaScript to program JavaScript. This chapter covers the concepts that all JavaScript programs rely on.

If you've had any experience with JavaScript programming, many of these concepts may be old hat, so you might just skim this chapter. But if you're new to JavaScript, or you're still not sure about the fundamentals, this chapter introduces you to basic (but crucial) topics.

Statements

A JavaScript *statement* is a basic programming unit, usually representing a single step in a JavaScript program. Think of a statement as a sentence: Just as you string sentences together to create a paragraph (or a chapter, or a book), you combine statements to create a JavaScript program. In the previous chapter, you saw several examples of statements. For example:

```
alert('Hello world!');
```

This single statement opens an alert window with the message "Hello World!" In many cases, a statement is a single line of code. Each statement ends with a semicolon—it's like a period at the end of a sentence. The semicolon makes it clear that the step is over and that the JavaScript interpreter should move on to the next action. When you're writing a JavaScript program, you generally type a statement, enter a

semicolon, press Return to create a new line, type another statement, followed by a semicolon, and so on until the program is complete.

NOTE As you see more advanced examples of JavaScript (like later in this book), you'll realize that semicolons don't always go at the end of every *line*. A semicolon sometimes won't appear until after many lines of code. Nonetheless, those multiple lines still form a single *statement*—just think of them as one, really long statement with lots of different punctuation (kind of like this sentence).

Officially, putting a semicolon even at the end of every *statement* is optional, and some programmers leave it out to make their code shorter. Don't be one of them. Leaving off the semicolon makes reading your code more difficult and, in some cases, causes JavaScript errors. If you want to make your JavaScript code more compact so that it downloads more quickly, see page 585.

Built-In Functions

JavaScript (and web browsers) lets you use various commands to make things happen in your programs and on your web pages. These commands, called *functions*, are like verbs in a sentence. They get things done. For example, the `alert()` function you encountered earlier makes the web browser open a dialog box and display a message.

Some functions, like `alert()` or `document.write()`, which you encountered on page 15, are specific to web browsers. In other words, they only work with web pages, so you won't find them when programming in other environments that use JavaScript (like Node.js, Adobe Photoshop, or Flash's JavaScript-based ActionScript).

Other functions are universal to JavaScript and work anywhere JavaScript works. For example, `isNaN()` is a function that checks to see whether a particular value is a number—this function comes in handy when you want to see if a visitor has correctly supplied a number for a question that requires a numerical answer (for example, "How many widgets would you like?"). You'll learn more about how to use `isNaN()`on page 564.

You'll learn many JavaScript functions throughout this book. One quick way to identify a function is by the parentheses. For example, you can tell `isNaN()` is a function because of the parentheses following `isNaN`.

In addition, JavaScript lets you create your own functions, so you can make your scripts do things beyond what the standard JavaScript commands offer. You'll learn about functions in Chapter 3, starting on page 85.

NOTE Sometimes you'll hear people refer to a JavaScript function as a *method*.

Types of Data

You deal with different types of information every day. Your name, the price of food, the address of your doctor's office, and the date of your next birthday are all information that is important to you. You make decisions about what to do based on this information. Computer programs are no different. They also rely on information to get things done. For example, to calculate the total for a shopping cart, the program needs to know the price and quantity of each item ordered. To customize a web page with a visitor's name ("Welcome Back, *Kotter*"), the program needs to know the name.

Programming languages usually categorize information into different types, and treat each type in a different way. In JavaScript, the three most basic types of data are `number`, `string`, and `Boolean`.

Numbers

Numbers are used for counting and calculating; you can keep track of the number of days until summer vacation, or calculate the cost of buying two tickets to a movie. Numbers are very important in JavaScript programming: You can use numbers to keep track of how many times a visitor has visited a web page, to specify the exact pixel position of an item on a web page, or to determine how many products a visitor wants to order.

In JavaScript, a number is represented by a numeric character; *5*, for example, is the number five. You can also use fractional numbers with decimals, like 5.25 or 10.3333333. JavaScript even lets you use negative numbers, like –130 or -459.67.

Because numbers are frequently used for calculations, your programs will often include mathematical operations. You'll learn about *operators* on page 33, but just to provide an example of using JavaScript with numbers, say you wanted to print the total value of 5 plus 15 on a web page; you could do that with this line of code:

```
document.write(5 + 15);
```

This snippet of JavaScript adds the two numbers together and prints the total (20) onto a web page. There are many different ways to work with numbers, and you'll learn more about them starting on page 33.

Strings

To display a name, a sentence, or any series of letters, you use strings. A *string* is just a series of characters (letters and other symbols) enclosed inside of quote marks. For example, `'Welcome, Hal'`, and `"You are here"` are both examples of strings. You used a string in Chapter 1 with the alert command—`alert('Hello World!')`.

A string's opening quote mark tells the JavaScript interpreter that what follows is a string—just a series of symbols. The interpreter accepts the symbols literally, rather than trying to interpret the string as anything special to JavaScript like a command. When the interpreter encounters the final quote mark, it understands that it has reached the end of the string and continues onto the next part of the program.

You can use either double quote marks (`"hello world"`) or single quote marks (`'hello world'`) to enclose the string, but you must make sure to use the *same type* of quote mark at the beginning and end of the string (for example, `"this is not right'` isn't a valid string because it begins with a double-quote mark and ends with a single-quote).

So, to pop-up an alert box with the message *Warning, warning!* you could write:

```
alert('Warning, warning!');
```

or

```
alert("Warning, warning!");
```

You'll use strings frequently in your programming—when adding alert messages, when dealing with user input on web forms, and when manipulating the contents of a web page. They're so important that you'll learn a lot more about using strings starting on page 36.

FREQUENTLY ASKED QUESTION

Putting Quotes into Strings

When I try to create a string with a quote mark in it, my program doesn't work. Why?

In JavaScript, quote marks indicate the beginning and end of a string, even when you don't want them to. When the JavaScript interpreter encounters the first quote mark, it says to itself, "Ahh, here comes a string." When it reaches a matching quote mark, it figures it has come to the end of the string. That's why you can't create a string like this: `"He said, "Hello.""`. In this case, the first quote mark (before the word "He") marks the start of the string, but as soon as the JavaScript interpreter encounters the second quote mark (before the word "Hello"), it figures that the string is over, so you then end up with the string "*He said*," then the *Hello.* part, and then another, empty string created by the two quote marks at the end. This code creates a JavaScript error, and your program won't work.

There are a couple of ways to get around this conundrum. The easiest method is to use single quotes to enclose a string that has one or more double quotes inside it. For example, `'He said, "Hello."'` is a valid string—the single quotes create the string, and the double quotes inside are a *part* of the string. Likewise, you can use double quotes to enclose a string that has a single quote inside it: `"This isn't fair"`, for example.

Another method is to tell the JavaScript interpreter to just treat the quote mark inside the string literally—that is, treat the quote mark as part of the string, not the end of the string. You do this using an *escape character*. If you precede the quote mark with a backward slash (\), JavaScript treats the quote as part of the string. So, you could rewrite the example like this: `"He said, \"Hello.\""`. In some cases, an escape character is the only choice. For example: `'He said, "This isn\'t fair."'`. Because the string is enclosed by single quotes, the lone single quote in the word "isn't" has to have a backward slash before it: *isn\'t*.

You can even escape quote marks when you don't necessarily have to—as a way to make it clear that the quote mark should be taken literally. For example, `'He said, \"Hello.\"'`. Even though you don't need to escape the double quotes (because single quotes surround the entire string), some programmers do it anyway so it's clear to them that the quote mark is just a quote mark.

Booleans

Whereas numbers and strings offer almost limitless variations, the Boolean data type is simple. It is either one of two values: *true* or *false*. You'll encounter Boolean data types when you create JavaScript programs that respond intelligently to user input and actions. For example, if you want to make sure a visitor supplied an email address before submitting a form, you can add logic to your page by asking the simple question: "Did the user type a valid email address?" The answer to this question is a Boolean value: Either the email address is valid (true) or it's not (false). Depending on the answer to the question, the page could respond in different ways. For example, if the email address is valid (true), then submit the form; if it is not valid (false), then display an error message and prevent the form from being submitted.

In fact, Boolean values are so important that JavaScript includes two special keywords to represent those values:

```
true
```

and

```
false
```

You'll learn how Boolean values come into play when adding logic to your programs in the box on page 66.

Variables

You can type a number, string, or Boolean value directly into your JavaScript program, but these data types work only when you already have the information you need. For example, you can make the string "Hi, Bob" appear in an alert box like this:

```
alert('Hi, Bob');
```

But that statement only makes sense if everyone who visits the page is named Bob. If you want to present a personalized message for different visitors, the name needs to be different depending on who is viewing the page: "Hi, Mary," "Hi, Joseph,'' "Hi, Ezra," and so on. Fortunately, all programming languages provide a tool called a *variable* to deal with just this kind of situation.

A variable is a way to store information so you can later use and manipulate it. For example, imagine a JavaScript-based pinball game where the goal is to get the highest score. When a player first starts the game, her score will be 0, but as she knocks the pinball into targets, the score will get bigger. In this case, the score is a variable because it starts at 0 but changes as the game progresses—in other words, a variable holds information that can *change* with the circumstances. See Figure 2-1 for an example of another game that uses variables.

Think of a variable as a kind of basket: You can put an item into a basket, look inside the basket, dump out the contents of a basket, or even replace what's inside the

basket with something else. However, even though you might change what's inside the basket, the basket itself remains the same.

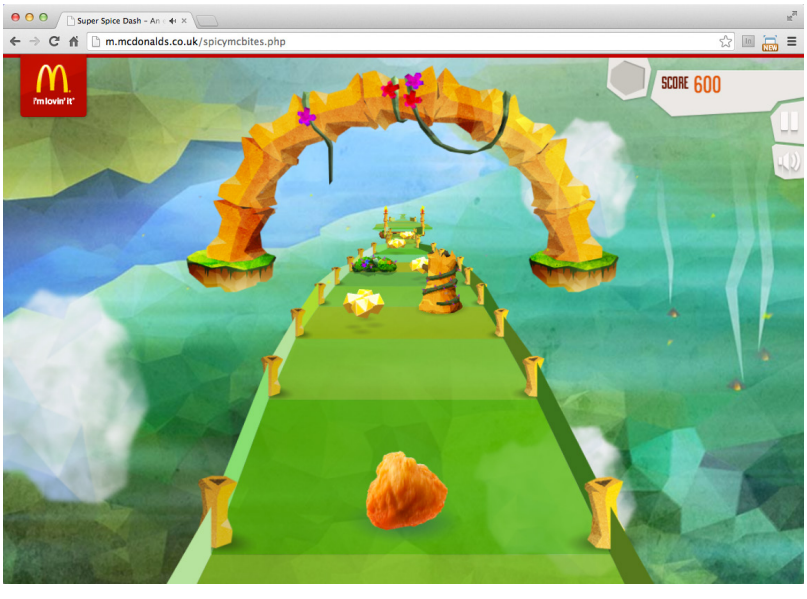

FIGURE 2-1

Super Spice Dash (http://mcbites.sh75.net) uses JavaScript to create a Super Monkey Ball-like game to promote McDonalds Spicy Chicken McBites. You maneuver a Chicken McBite (no joke) through a course, collecting golden crumbs and chilies while avoiding obstacles and jumping over gaps in the road. As you play, your score is tracked in the upper-right corner. This game uses a variable to keep track of the score, as its value changes as the game goes on.

Creating a Variable

In JavaScript, to create a variable named `score`, you would type:

```
var score;
```

The first part, `var`, is a JavaScript keyword that creates, or, in programming-speak, *declares* the variable. The second part of the statement, `score`, is the variable's name.

What name you use is up to you, but there are a few rules you must follow when naming variables:

- **Variable names must begin with a letter, $, or _.** In other words, you can't begin a variable name with a number or punctuation: so `1thing`, and `&thing` won't work, but `score`, `$score`, and `_score` are fine.
- **Variable names can only contain letters, numbers, $, and _.** You can't use spaces or any other special characters anywhere in the variable name: `fish&chips` and `fish and chips` aren't legal, but `fish_n_chips` and `plan9` are.
- **Variable names are case-sensitive.** The JavaScript interpreter sees uppercase and lowercase letters as distinct, so a variable named `SCORE` is different from a variable named `score`, which is also different from variables named `sCoRE` and `Score`.

- **Avoid keywords.** Some words in JavaScript are specific to the language itself: `var`, for example, is used to create a variable, so you can't name a variable `var`. In addition, some words, like `alert`, `document`, and `window`, are considered special properties of the web browser. You'll end up with a JavaScript error if you try to use those words as variable names. You can find a list of some reserved words in Table 2-1. Not all of these reserved words will cause problems in all browsers, but it's best to steer clear of these names when naming variables.

TABLE 2-1 *Some words are reserved for use by JavaScript and the web browser. Avoid using them as variable names.*

JAVASCRIPT KEYWORDS	RESERVED FOR FUTURE USE	RESERVED FOR BROWSER
break	abstract	alert
case	boolean	blur
catch	byte	closed
continue	char	document
debugger	class	focus
default	const	frames
delete	double	history
do	enum	innerHeight
else	export	innerWidth
false	extends	length
finally	final	location
for	float	navigator
function	goto	open
if	implements	outerHeight
in	import	outerWidth
instanceof	int	parent
new	interface	screen
null	let	screenX
return	long	screenY
switch	native	statusbar
this	package	window
throw	private	
true	protected	
try	public	
typeof	short	
var	static	

JAVASCRIPT KEYWORDS	RESERVED FOR FUTURE USE	RESERVED FOR BROWSER
void	super	
while	synchronized	
with	throws	
	transient	
	volatile	
	yield	

In addition to these rules, aim to make your variable names clear and meaningful. Naming variables according to what type of data you'll be storing in them makes it much easier to look at your programming code and immediately understand what's going on. For example, `score` is a great name for a variable used to track a player's game score. The variable name `s` would also work, but the single letter "s" doesn't give you any idea about what's stored in the variable.

Likewise, make your variable names easy to read. When you use more than one word in a variable name, either use an underscore between words or capitalize the first letter of each word after the first. For example, `imagepath` isn't as easy to read and understand as `image_path` or `imagePath`.

Using Variables

Once you declare a variable, you can store any type of data you'd like in it. To do so, you use the = sign. For example, to store the number 0 in a variable named `score`, you could type this code:

```
var score;
score = 0;
```

The first line of this code creates the variable; the second line stores the number 0 in that variable. The equals sign is called an *assignment operator*, because it's used to assign a value to a variable. You can also create a variable and store a value in it with a single JavaScript statement like this:

```
var score = 0;
```

You can store strings, numbers, and Boolean values in a variable:

```
var firstName = 'Peter';
var lastName = 'Parker';
var age = 22;
var isSuperHero = true;
```

NOTE To save typing, you can declare multiple variables with a single `var` keyword, like this:

```
var x, y, z;
```

You can even declare and store values into multiple variables in one JavaScript statement:

```
var isSuperHero=true, isAfraidOfHeights=false;
```

Once you've stored a value in a variable, you can access that value simply by using the variable's name. For example, to open an alert dialog box and display the value stored in the variable `score`, you'd type this:

```
alert(score);
```

Notice that you don't use quotes with a variable—that's just for strings; so the code `alert('score')` will display the word "score" and not the value stored in the variable `score`. Now you can see why strings have to be enclosed in quote marks: The JavaScript interpreter treats words without quotes as either special JavaScript objects (like the `alert()` command) or as variable names.

NOTE You should only use the `var` keyword once for each variable—when you first create the variable. After that, you're free to assign new values to the variable without using `var`.

Working with Data Types and Variables

Storing a particular piece of information like a number or string in a variable is usually just a first step in a program. Most programs also manipulate data to get new results. For example, you can add a number to a score to increase it, multiply the number of items ordered by the cost of the item to get a sub total, or personalize a generic message by adding a name to the end: "Good to see you again, Igor." JavaScript provides various *operators* to modify data. An operator is a symbol or word that can change one or more values into something else. For example, you use the + sign—the addition operator—to add numbers together. There are different types of operators for the different data types.

Basic Math

JavaScript supports basic mathematical operations such as addition, division, subtraction, and so on. Table 2-2 shows the most basic math operators and how to use them.

TABLE 2-2 *Basic math with JavaScript*

OPERATOR	WHAT IT DOES	HOW TO USE IT
+	Adds two numbers	`5 + 25`
-	Subtracts one number from another	`25 - 5`
*	Multiplies two numbers	`5 * 10`
/	Divides one number by another	`15/5`

FREQUENTLY ASKED QUESTION

Spaces, Tabs, and Carriage Returns in JavaScript

JavaScript seems so sensitive about typos. How do I know when I'm supposed to use space characters, and when I'm not allowed to?

In general, JavaScript is pretty relaxed about spaces, carriage returns, and tabs. You can often leave out spaces or even add extra spaces and carriage returns without a problem. JavaScript interpreters ignore extra spaces, so you're free to insert extra spaces, tabs, and carriage returns to format your code. For example, you don't need a space on either side of an assignment operator, but you can add them if you find it easier to read. Both of the following lines of code work:

```
var formName='signup';
var formRegistration = 'newsletter' ;
```

In fact, you can insert as many spaces as you'd like, and even insert carriage returns within a statement. So both of the following statements also work:

```
var formName     =          'signup';
var formRegistration
                =
          'newsletter';
```

Of course, just because you can insert extra space, doesn't mean you should. The last two examples are actually harder to read and understand because of the extra space. So the general rule of thumb is to add extra space if it makes your code easier to understand. For example, extra carriage returns help make code easier to read when declaring and setting the value of multiple variables at once. The following code is a single line:

```
var score=0, highScore=0, player='';
```

However, some programmers find it easier to read if each variable is on its own line:

```
var score=0,
    highScore=0,
    player='';
```

Whether you find this spacing easier to read is up to you; the JavaScript interpreter just ignores those line breaks. You'll see examples of how space can make code easier to read with JavaScript object literals (page 136) and with arrays (page 44).

There are a couple of important exceptions to these rules. For example, you can't insert a carriage return inside a string; in other words, you can't split a string over two lines in your code like this:

```
var name = 'Bob
           Smith';
```

Inserting a carriage return (pressing the Enter or Return key) like this produces a JavaScript error and your program won't run.

In addition, you must put a space between keywords: `varscore=0`, for example, is not the same as `var score=0`. The latter example creates a new variable named "score," while the former stores the value 0 in a variable named "varscore." The JavaScript interpreter needs the space between `var` and `score` to identify the `var` keyword: `var score=0`. However, a space isn't necessary between keywords and symbols like the assignment operator (=) or the semicolon that ends a statement.

You may be used to using an x for multiplication (4 x 5, for example), but in JavaScript, you use the * (asterisk) to multiply two numbers.

You can also use variables in mathematical operations. Because a variable is only a container for some other value like a number or string, using a variable is the same as using the contents of that variable. Here's an example:

```
var price = 10;
var itemsOrdered = 15;
var totalCost = price * itemsOrdered;
```

The first two lines of code create two variables (`price` and `itemsOrdered`) and store a number in each. The third line of code creates another variable (`totalCost`) and stores the results of multiplying the value stored in the `price` variable (10) and the value stored in the `itemsOrdered` variable. In this case, the total (150) is stored in the variable `totalCost`.

This sample code also demonstrates the usefulness of variables. Suppose you write a program as part of a shopping cart system for an e-commerce website. Throughout the program, you need to use the price of a particular product to make various calculations. You could code the actual price throughout the program (for example, say the product cost $10, so you'd type 10 in each place in the program that price is used). However, if the price ever changes, you'd have to locate and change each line of code that uses the price. By using a variable, on the other hand, you can set the price of the product somewhere near the beginning of the program. Then, if the price ever changes, you only need to modify the one line of code that defines the product's price to update the price throughout the program:

```
var price = 20;
var itemsOrdered = 15;
var totalCost = price * itemsOrdered;
```

There are lots of other ways to work with numbers (you'll learn a bunch starting on page 562), but you'll find that you most frequently use the basic math operators listed in Table 2-2.

The Order of Operations

If you perform several mathematical operations at once—for example, if you add up several numbers and then multiply them all by 10—you need to keep in mind the order in which the JavaScript interpreter performs its calculations. Some operators take precedence over other operators, so they're calculated first. This fact can cause some unwanted results if you're not careful. Take this example:

```
4 + 5 * 10
```

You might think this is simply calculated from left to right: 4 + 5 is 9 and 9 * 10 is 90. It's not. The multiplication actually goes first, so this equation works out to 5 * 10 is 50, plus 4 is 54. Multiplication (the * symbol) and division (the / symbol) take precedence over addition (+) and subtraction (–).

To make sure that the math works out the way you want it, use parentheses to group operations. For example, you could rewrite the equation above like this:

```
(4 + 5) * 10
```

Any math that's performed inside parentheses happens first, so in this case the 4 is added to 5 first and the result, 9, is then multiplied by 10. If you want the multiplication to occur first, it would be clearer to write that code like this:

```
4 + (5*10);
```

Combining Strings

Combining two or more strings to make a single string is a common programming task. For example, if a web page has a form that collects a person's first name in one form field and his last name in a different field, you need to combine the two fields to get his complete name. What's more, if you want to display a message letting the user know his form information was submitted, you need to combine the generic message with the person's name: "John Smith, thanks for your order."

Combining strings is called *concatenation*, and you accomplish it with the + operator. Yes, that's the same + operator you use to add number values, but with strings it behaves a little differently. Here's an example:

```
var firstName = 'John';
var lastName = 'Smith';
var fullName = firstName + lastName;
```

In the last line of the code above, the contents of the variable `firstName` are combined (or concatenated) with the contents of the variable `lastName`. The two are literally joined together and the result is placed in the variable `fullName`. In this example, the resulting string is "JohnSmith"—there isn't a space between the two names, as concatenating just fuses the strings together. In this case (and many others), you need to add an empty space between strings that you intend to combine:

```
var firstName = 'John';
var lastName = 'Smith';
var fullName = firstName + ' ' + lastName;
```

The `' '` in the last line of this code is a single quote, followed by a space, followed by a final single quote. This code is simply a string that contains an empty space. When placed between the two variables in this example, it creates the string "John Smith". This last example also demonstrates that you can combine more than two strings at a time; in this case, three strings.

NOTE Remember that a variable is just a container that can hold any type of data, like a string or number. So when you combine two variables with strings (`firstName + lastName`), it's the same as joining two strings like this: `'John' + 'Smith'`.

Combining Numbers and Strings

Most of the mathematical operators only make sense for numbers. For example, it doesn't make any sense to multiply 2 and the string "eggs". If you try this example, you'll end up with a special JavaScript value `NaN`, which stands for "not a number." However, there are times when you may want to combine a string with a number. For example, say you want to present a message on a web page that specifies how many times a visitor has been to your website. The number of times she's visited is a *number*, but the message is a *string*. In this case, you use the + operator to do two things: convert the number to a string and concatenate it with the other string. Here's an example:

```
var numOfVisits = 101;
var message = 'You have visited this site ' + numOfVisits + ' times.';
```

In this case, *message* contains the string "You have visited this site 101 times." The JavaScript interpreter recognizes that there is a string involved, so it realizes it won't be doing any math (no addition). Instead, it treats the + as the concatenation operator, and at the same time realizes that the number should be converted to a string as well.

This example may seem like a good way to print words and numbers in the same message. In this case, it's obvious that the number is part of a string of letters that makes up a complete sentence, and whenever you use the + operator with a string value and a number, the JavaScript interpreter converts the number to a string.

That feature, known as *automatic type conversion*, can cause problems, however. For example, if a visitor answers a question on a form ("How many pairs of shoes would you like?") by typing a number (2, for example), that input is treated like a string—`'2'`. So you can run into a situation like this:

```
var numOfShoes = '2';
var numOfSocks = 4;
var totalItems = numOfShoes + numOfSocks;
```

You'd expect the value stored in `totalItems` to be 6 (2 shoes + 4 pairs of socks). Instead, because the value in `numOfShoes` is a string, the JavaScript interpreter converts the value in the variable `numOfSocks` to a string as well, and you end up with the string `'24'` in the `totalItems` variable. There are a couple of ways to prevent this error.

First, you add + to the beginning of the string that contains a number like this:

```
var numOfShoes = '2';
var numOfSocks = 4;
var totalItems = +numOfShoes + numOfSocks;
```

Adding a + sign before a variable (making sure there's no space between the two) tells the JavaScript interpreter to try to convert the string to a number value—if the string only contains numbers, like `'2'`, you'll end up with the string converted to a number. In this example, you end up with `6` (`2 + 4`). Another technique is to use the `Number()` command like this:

```
var numOfShoes = '2';
var numOfSocks = 4;
var totalItems = Number(numOfShoes) + numOfSocks;
```

`Number()` converts a string to a number if possible. (If the string is just letters and not numbers, you get the `NaN` value to indicate that you can't turn letters into a number.)

In general, you'll most often encounter numbers as strings when getting input from a visitor; for example, when retrieving a value a visitor entered into a form field. So, if you need to do any addition using input collected from a form or other source of visitor input, make sure you run it through the `Number()` command first.

NOTE This problem only occurs when adding a number to a string that contains a number. If you try to multiply the `numOfShoes` variable with a variable containing a number—`shoePrice`, for example—the JavaScript interpreter will convert the string in `numOfShoes` to a number and then multiply it by the `shoePrice` variable.

Changing the Values in Variables

Variables are useful because they can hold values that change as the program runs—a score that changes as a game is played, for example. So how do you change a variable's value? If you just want to replace what's contained inside a variable, assign a new value to the variable. For example:

```
var score = 0;
score = 100;
```

However, you'll frequently want to keep the value that's in the variable and just add something to it or alter it in some way. For example, with a game score, you never just set a new score—you add or subtract from the current score. To add to the value of a variable, you use the variable's name as part of the operation, like this:

```
var score = 0;
score = score + 100;
```

That last line of code may appear confusing at first, but it uses a very common technique. Here's how it works: All of the action happens to the right of the = sign first; that is, the *score + 100* part. Translated, it means "take what's currently stored in `score` (0) and then add 100 to it." The result of that operation is *then* stored back into the variable `score`. The final outcome of these two lines of code is that the variable `score` now has the value of 100.

The same logic applies to other mathematical operations like subtraction, division, and multiplication:

```
score = score - 10;
score = score * 10;
score = score / 10;
```

In fact, performing math on the value in a variable and then storing the result back into the variable is so common that there are shortcuts for doing so with the main mathematical operations, as pictured in Table 2-3.

TABLE 2-3 *Shortcuts for performing math on a variable*

OPERATOR	WHAT IT DOES	HOW TO USE IT	THE SAME AS
+=	Adds value on the right side of equals sign to the variable on the left.	`score += 10`	`score = score + 10`
-=	Subtracts value on the right side of the equals sign from the variable on the left.	`score -= 10`	`score = score - 10`
*=	Multiplies the variable on the left side of the equals sign and the value on the right side of the equals sign.	`score *= 10`	`score = score * 10`
/=	Divides the value in the variable by the value on the right side of the equals sign.	`score /= 10`	`score = score / 10`
++	Placed directly after a variable name, ++ adds 1 to the variable.	`score++`	`score = score + 1`
--	Placed directly after a variable name, -- subtracts 1 from the variable.	`score--`	`score = score - 1`

The same rules apply when concatenating a string to a variable. For example, say you have a variable with a string in it and want to add another couple of strings onto that variable:

```
var name = 'Franklin';
var message = 'Hello';
message = message + ' ' + name;
```

As with numbers, there's a shortcut operator for concatenating a string to a variable. The += operator adds the string value to the right of the = sign to the end of the variable's string. So you could rewrite the last line of the above code like this:

```
message += ' ' + name;
```

You'll see the += operator frequently when working with strings, and throughout this book.