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

HTML5

the missing manual®

The book that should have been in the box®

Second Edition

O'REILLY®

Matthew MacDonald

HTML5 / WEB DEVELOPMENT

Answers found here!

HTML5 is more than a markup language—it's a collection of several independent web standards. Fortunately, this expanded guide covers everything you need in one convenient place. With step-by-step tutorials and real-world examples, *HTML5: The Missing Manual* shows you how to build web apps that include video tools, dynamic graphics, geolocation, offline features, and responsive layouts for mobile devices.

the missing manual®

The book that should have been in the box®

The important stuff you need to know

- **Add audio and video without plugins.** Build playback pages that work in every browser.
- **Create stunning visuals with Canvas.** Draw shapes, pictures, and text; play animations; and run interactive games.
- **Jazz up your pages with CSS3.** Add fancy fonts and eye-catching effects with transitions and animation.
- **Design better web forms.** Collect information from visitors more efficiently with HTML5 form elements.
- **Build it once, run it everywhere.** Use responsive design to make your site look good on desktops, tablets, and smartphones.
- **Include rich desktop features.** Build self-sufficient web apps that work offline and store the data users need.

Matthew MacDonald is a science and technology writer with more than a dozen books to his name. He's known for books about building websites, including *Creating a Website: The Missing Manual* and *WordPress: The Missing Manual*, as well as quirky handbooks like *Your Brain: The Missing Manual* and *Your Body: The Missing Manual*.

US $39.99 CAN $41.99
ISBN: 978-1-449-36326-0

O'REILLY®
missingmanuals.com
twitter: @missingmanuals
facebook.com/MissingManuals

HTML5

2nd Edition

the missing manual®

The book that should have been in the box®

Matthew MacDonald

O'REILLY®

Beijing | Cambridge | Farnham | Köln | Sebastopol | Tokyo

HTML5: The Missing Manual, 2nd Edition

by Matthew MacDonald

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

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

August 2011: First Edition.
December 2013: Second Edition

Revision History for the Second Edition:

2013-12-09 First release

See *http://oreil.ly/html5tmm_2e* for release details.

ISBN-13: 978-1-4493-6326-0

[LSI]

Contents

Part Two: Video, Graphics, and Glitz

Part Three: Building Web Apps

Part Four: Appendixes

The Missing Credits

ABOUT THE AUTHOR

Matthew MacDonald is a science and technology writer with well over a dozen books to his name. Web novices can tiptoe out onto the Internet with him in *Creating a Website: The Missing Manual*. Office geeks can crunch the numbers in *Excel 2013: The Missing Manual*. And human beings of all description can discover just how strange they really are in the quirky handbooks *Your Brain: The Missing Manual* and *Your Body: The Missing Manual*.

ABOUT THE CREATIVE TEAM

Nan Barber (editor) has been working on the Missing Manual series since its inception. She lives in Massachusetts with her husband and various Apple and Android devices. Email: *nanbarber@oreilly.com*.

Kristen Brown (production editor) is a graduate of the publishing program at Emerson College. She lives in the Boston area with her husband and their large collection of books and board games. Email: *kristen@oreilly.com*.

Kara Ebrahim (conversion) lives, works, and plays in Cambridge, MA. She loves graphic design and all things outdoors. Email: *kebrahim@oreilly.com*.

Julie Van Keuren (proofreader) quit her newspaper job in 2006 to move to Montana and live the freelancing dream. She and her husband (who is living the novel-writing dream) have two sons. Email: *little_media@yahoo.com*.

Julie Hawks (indexer) is a teacher and eternal student. She can be found wandering about with a camera in hand. Email: *juliehawks@gmail.com*.

Shelley Powers (technical reviewer) is a former HTML5 working group member and author of several O'Reilly books. Website: *http://burningbird.net*.

Darrell Heath (technical reviewer) is a freelance web/print designer and web developer from Newfoundland and Labrador, Canada, with a background in Information Technology and visual arts. He has authored weekly tutorial content for NAPP, *Layers* magazine, and Planet Photoshop, and in his spare time offers design- and technology-related tips through his blog at *www.heathrowe.com/blog*. Email: *darrell@heathrowe.com*.

ACKNOWLEDGEMENTS

No author could complete a book without a small army of helpful individuals. I'm deeply indebted to the whole Missing Manual team, especially my editor Nan Barber, who never seemed fazed by the shifting sands of HTML5; and expert tech reviewers Shelley Powers and Darrell Heath, who helped spot rogue errors and offered consistently good advice. And, as always, I'm also deeply indebted to numerous others who've toiled behind the scenes indexing pages, drawing figures, and proofreading the final copy.

Finally, for the parts of my life that exist outside this book, I'd like to thank all my family members. They include my parents, Nora and Paul; my extended parents, Razia and Hamid; my wife, Faria; and my daughters, Maya, Brenna, and Aisha. Thanks, everyone!

—Matthew MacDonald

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; cross-references to specific pages (not just chapters); and RepKover, a detached-spine binding that lets the book lie perfectly flat without the assistance of weights or cinder blocks.

Recent and upcoming titles include:

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 2013: The Missing Manual by Matthew MacDonald

FileMaker Pro 12: 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

Google+: The Missing Manual by Kevin Purdy

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, Fifth 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

JavaScript & jQuery: The Missing Manual, Second Edition by David Sawyer McFarland

Kindle Fire HD: The Missing Manual by Peter Meyers

Living Green: The Missing Manual by Nancy Conner

Microsoft Project 2013: The Missing Manual by Bonnie Biafore

Motorola Xoom: The Missing Manual by Preston Gralla

Netbooks: The Missing Manual by J.D. Biersdorfer

NOOK HD: The Missing Manual by Preston Gralla

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

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 2013: The Missing Manual by Bonnie Biafore

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

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

Windows 8.1: The Missing Manual by David Pogue

WordPress: The Missing Manual 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

At first glance, you might assume that HTML5 is the fifth version of the HTML web page–writing language. But the real story is a whole lot messier.

HTML5 is a rebel. It was dreamt up by a loose group of freethinkers who weren't in charge of the official HTML standard. It allows page-writing practices that were banned a decade ago. It spends thousands of words painstakingly telling browser makers how to deal with markup mistakes, rather than rejecting them outright. It finally makes video playback possible without a browser plug-in like Flash. And it introduces an avalanche of JavaScript-fueled features that can give web pages some of the rich, interactive capabilities of traditional desktop software.

Understanding HTML5 is no small feat. One stumbling block is that people use the word *HTML5* to refer to a dozen or more separate standards. (As you'll learn, this problem is the result of HTML5's evolution. It began as a single standard and was later broken into more manageable pieces.) In fact, HTML5 has come to mean "HTML5 and all its related standards" or, even more broadly, "the next generation of web-page-writing technologies." That's the version of HTML5 that you'll explore in this book: everything from the HTML5 core language to a few new features lumped in with HTML5 even though they were *never* a part of the standard.

The second challenge of HTML5 is browser support. Different browsers support HTML5 to different degrees. The most notable laggard is Internet Explorer 8, which supports very little HTML5 and is still found on one out of every 20 web-surfing computers. (At least it was at the time of this writing. Page 30 explains how you can get the latest browser usage statistics.) Fortunately, there are workarounds that can bridge the browser support gaps—some easy, and some ugly. In this book, you'll learn a bit of both on your quest to use HTML5 in your web pages *today*.

Despite the challenges HTML5 presents, there's one fact that no one disputes—*HTML5 is the future*. Huge software companies like Apple, Google, and Microsoft have lent it support, and the W3C (World Wide Web Consortium) has given up its work on XHTML to formalize and endorse it. With this book, you too can join the party and use HTML5 to create cool pages like the one shown in Figure I-1.

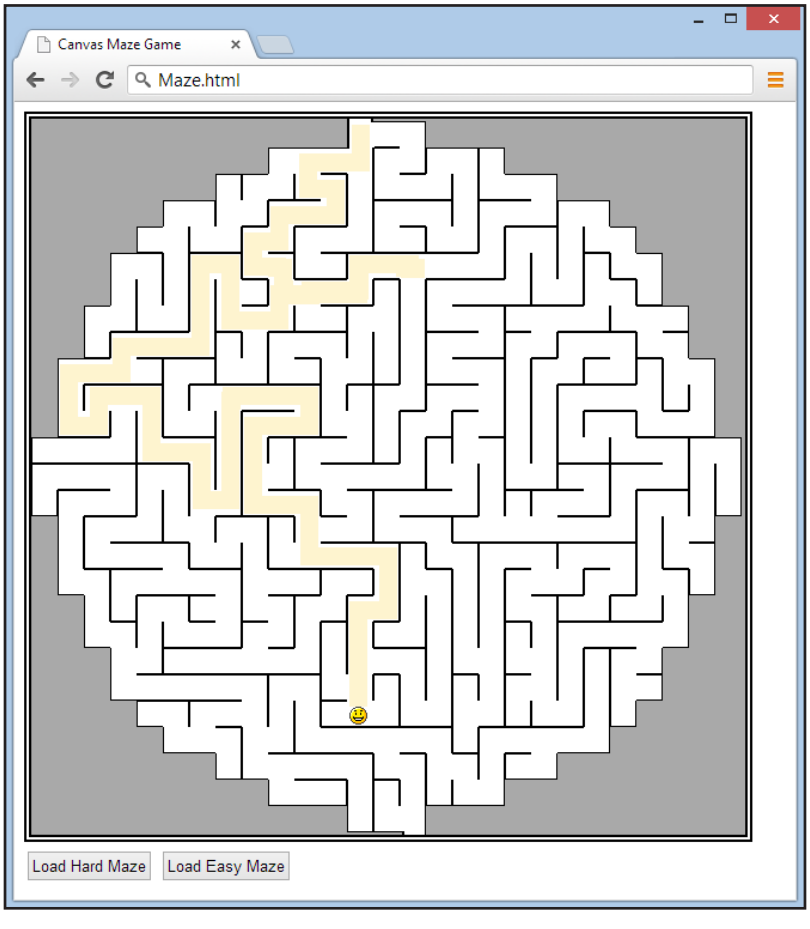


FIGURE I-1

In the dark old days of the Web, you had to build your web page games with a browser plug-in like Flash. But with HTML5's features, including the canvas (shown here), you can use trusty, plug-in-free JavaScript. Here, HTML5 powers a maze game that you'll dissect in Chapter 9.

What You Need to Get Started

This book covers HTML5, the latest and greatest version of the HTML standard. And while you don't need to be a markup master to read it, you *do* need some previous web design experience. Here's the official rundown:

- **Web page writing.** This book assumes you've written at least a few web pages before (or at the very least, you understand how to use HTML elements to structure content into headings, paragraphs, and lists). If you're new to web design, you're better off with a gentler introduction, like my own *Creating a Website: The Missing Manual, Third Edition*. (But don't worry; you won't be trapped in the past, as all the examples in the third edition of *Creating a Website* are valid HTML5 documents.)
- **Style sheet experience.** No modern website is possible without CSS—the Cascading Style Sheet standard—which supplies the layout and formatting for web pages. To follow along in this book, you should know the basics of style sheets: how to create them, what goes inside, and how to attach one to a page. If you're a bit hazy on the subject, you can catch up in Appendix A, "Essential CSS." But if you need more help, or if you just want to sharpen your CSS skills to make truly cool layouts and styles, check out a supplementary book like *CSS3: The Missing Manual* by David Sawyer McFarland.
- **JavaScript experience.** No, you don't need JavaScript to create an HTML5 page. However, you do need JavaScript if you want to use many of HTML5's most powerful features, like drawing on a canvas or talking to a web server. If you have a smattering of programming experience but don't know much about JavaScript, then Appendix B, "JavaScript: The Brains of Your Page" can help you get up to speed. But if the idea of writing code sounds about as comfortable as crawling into bed with an escaped python, then you'll either end up skipping a lot of material in this book, or you'll need to fill in the gaps with a book like *JavaScript & jQuery: The Missing Manual* by David Sawyer McFarland.

Writing HTML5

You can write HTML5 pages using the same software you use to write HTML pages. That can be as simple as a lowly text editor, like Notepad (on Windows) or TextEdit (on Mac). Many current design tools, like Adobe Dreamweaver and Microsoft Visual Studio, have templates that let you quickly create new HTML5 documents. However, the basic structure of an HTML5 page is so simple that you can use any web editor to create one, even if it wasn't specifically designed for HTML5.

NOTE And, of course, it doesn't matter whether you do your surfing and web page creation on a Windows PC or the latest MacBook Pro—HTML5 pays no attention to what operating system you use.

Viewing HTML5

You'll get support for most HTML5 features in the latest version of any modern browser, including the mobile browsers than run on Apple and Android devices. As long as your browser is up to date, HTML5 will perform beautifully—and you'll be able to try out the examples in this book.

Currently, no browser supports *every* last detail of HTML5, in part because HTML5 is really a collection of interrelated standards. Google Chrome generally leads the browser race in HTML5 support, with Firefox and Opera in close pursuit. Safari lags the pack a bit, and Internet Explorer trails still further behind. The real problem lies in the old copies of Internet Explorer that can't be updated because they're running on creaky operating systems like Windows Vista or Windows XP (which is still chugging away on a fifth of the world's desktop computers). Page 26 has a closer look at this problem and some advice on how to deal with it.

When Will HTML5 Be Ready?

The short answer is "now." Even the despised Internet Explorer 6, which is 10 years old and chock-full of website-breaking quirks, can display basic HTML5 documents. That's because the HTML5 standard was intentionally created in a way that embraces and extends traditional HTML.

The more detailed answer is "it depends." As you've already learned, HTML5 is a collection of different standards with different degrees of browser support. So although every web developer can switch over to HTML5 documents today (and many big sites, like Google, YouTube, and Wikipedia, already have), it may be some time before it's safe to use all of HTML5's fancy new features—at least without adding some sort of fallback mechanism for less-enlightened browsers.

NOTE Before encouraging you to use a new HTML5 feature, this book clearly indicates that feature's current level of browser support. Of course, browser versions change relatively quickly, so you'll want to perform your own up-to-date research before you embrace any feature that might cause problems. The website *http://caniuse.com* lets you look up specific features and tells you exactly which browser versions support them. (You'll learn more about this useful tool on page 27.)

As a standards-minded developer, you also might be interested in knowing how far the various standards are in their journey toward official status. This is complicated by the fact that the people who dreamt up HTML5 have a slightly subversive philosophy, and they often point out that what browsers support is more important than what the official standard says. In other words, you can go ahead and use everything that you want right now, if you can get it to work. But web developers, big companies, governments, and other organizations often take their cues about whether a language is ready to use by looking at the status of its standard.

At this writing, the HTML5 language is in the *candidate recommendation* stage, which means the standard is largely settled but browser makers are still polishing up their HTML5 implementations. The next and final stage is for the standard to become a full *recommendation*, and HTML5 is expected to hit that landmark in late 2014. In the meantime, the W3C has already published a *working draft* of the next version of the standard, which it calls HTML 5.1. (For more help making sense of all the different versions, see the box on the next page.)

FREQUENTLY ASKED QUESTION

The Difference Between HTML5 and HTML 5.1

Is there another new version of HTML? And what's with the inconsistent spacing?

As you'll learn in Chapter 1, HTML5 has gone through two sets of hands. This process has left a few quirks behind, including a slightly schizophrenic versioning system.

The people who originally created HTML5—the members of WHATWG, which you'll meet on page 5—aren't much interested in version numbers. They consider HTML5 to be a living language. They encourage web developers to pay attention to browser support, rather than worry about exact version numbers.

However, the WHATWG passed HTML5 to the official web standard-keepers—the W3C—so they could finalize it. The W3C is a more careful, methodical organization. The folks there wanted a way to separate their initial publication of the HTML5 standard from the slightly tweaked and cleaned up successors that were sure to follow. Thus, the W3C decided to name the first release of the HTML5 standard HTML 5.0 (note the space). The second release will be HTML 5.1, followed by a third release called HTML 5.2. Confusingly enough, all these versions are still considered to be HTML5.

Incidentally, the later iterations of the HTML5 standard aren't likely to add major changes. Instead, new features will turn up in separate, complementary specifications. This way, small groups of people can quickly develop new, useful HTML5 features without needing to wait for an entirely new revision of the language.

About the Outline

This book crams a comprehensive HTML5 tutorial into 13 chapters. Here's what you'll find:

Part One: Meet the New Language

- Chapter 1 explains how HTML turned into HTML5. You'll meet your first HTML5 document, see how the language has changed, and take a look at browser support.
- Chapter 2 tackles HTML5's *semantic elements*—a group of elements that can inject meaning into your markup. Used properly, this extra information can help browsers, screen readers, web design tools, and search engines work smarter.
- Chapter 3 goes deeper into the world of semantics with add-on standards like *microdata*. And while it may seem a bit theoretical, there's a fat prize for the web developers who understand it best: better, more detailed listings in search engines like Google.
- Chapter 4 explores HTML5's changes to the web form elements—the text boxes, lists, checkboxes, and other widgets that you use to collect information from your visitors. HTML5 adds a few frills and some basic tools for catching data-entry errors.

Part Two: Video, Graphics, and Glitz

- Chapter 5 hits one of HTML5's most exciting features: its support for audio and video playback. You'll learn how to survive Web Video Codec Wars to create playback pages that work in every browser, and you'll even see how to create your own customized player.
- Chapter 6 introduces the latest version of the CSS3 standard, which complements HTML5 nicely. You'll learn how to jazz up your text with fancy fonts and add eye-catching effects with transitions and animation.
- Chapter 7 explores CSS3 media queries. You'll learn how to use them to create responsive designs—website layouts that seamlessly adapt themselves to different mobile devices.
- Chapter 8 introduces the two-dimensional drawing surface called the *canvas*. You'll learn how to paint it with shapes, pictures, and text, and even build a basic drawing program (with a healthy dose of JavaScript code).
- Chapter 9 pumps up your canvas skills. You'll learn about shadows and fancy patterns, along with more ambitious canvas techniques like clickable, interactive shapes and animation.

Part Three: Building Web Apps

- Chapter 10 covers the web storage feature that lets you store small bits of information on the visitor's computer. You'll also learn about ways to process a user-selected file in your web page JavaScript code, rather than on the web server.
- Chapter 11 explores the HTML5 caching feature that can let a browser keep running a web page, even if it loses the web connection.
- Chapter 12 dips into the challenging world of web server communication. You'll start with the time-honored `XMLHttpRequest` object, which lets your JavaScript code contact the web server and ask for information. Then you'll move on to two newer features: server-side events and the more ambitious web sockets.
- Chapter 13 covers three miscellaneous features that address challenges in modern web applications. First, you'll see how geolocation can pin down a visitor's position. Next, you'll use web workers to run time-consuming tasks in the background. Finally, you'll learn about the browser history feature, which lets you sync up the web page URL to the current state of the page.

There are also two appendixes that can help you catch up with the fundamentals you need to master HTML5. Appendix A, "Essential CSS," gives a stripped-down summary of CSS; Appendix B, "JavaScript: The Brains of Your Page" gives a concise overview of JavaScript.

About the Online Resources

As the owner of a Missing Manual, you've got more than just a book to read. Online, you'll find example files as well as tips, articles, and maybe even a video or two. 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.

The Missing CD

This book doesn't have a CD pasted inside the back cover, but you're not missing out on anything. Go to *http://missingmanuals.com/cds/html5tmm2e* to download the web page examples discussed and demonstrated in this book. And so you don't wear down your fingers typing long web addresses, the Missing CD page offers a list of clickable links to the websites mentioned in each chapter.

TIP If you're looking for a specific example, here's a quick way to find it: Look at the corresponding figure in this book. The file name is usually visible at the end of the text in the web browser's address box. For example, if you see the file path *c:\HTML5\Chapter01\SuperSimpleHTML5.html* (Figure 1-1), you'll know that the corresponding example file is *SuperSimpleHTML5.html*.

The Try-Out Site

There's another way to use the examples: on the example site at *www.prosetech.com/html5*. There you'll find live versions of every example from this book, which you can run in your browser. This convenience just might save you a few headaches, because HTML5 includes several features that require the involvement of a real web server. (If you're running web pages from the hard drive on your personal computer, these features may develop mysterious quirks or stop working altogether.) By using the live site, you can see how an example is supposed to work before you download the page and start experimenting on your own.

NOTE Don't worry—when you come across an HTML5 feature that needs web server hosting, this book will warn you.

Registration

If you register this book at oreilly.com (*www.oreilly.com*), you'll be eligible for special offers—like discounts on future editions of *HTML5: The Missing Manual*. Registering takes only a few clicks. Type *http://tinyurl.com/registerbook* 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 of folks who share your interest in creating their own sites.

To have your say, go to *www.missingmanuals.com/feedback*.

Errata

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 suggest. 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/html52e-mm* to report an error and view existing corrections.

Safari® Books Online

Safari® Books Online is an on-demand digital library that lets you search over 7,500 technology books and videos.

With a subscription, you can read any page and watch any video from our library. Access new titles before they're available in print. 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.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

PART 1

Modern Markup

CHAPTER

1

Introducing HTML5

If HTML were a movie, HTML5 would be its surprise twist. HTML wasn't meant to survive into the 21st century. The official web standards organization, the W3C (short for World Wide Web Consortium), left HTML for dead way back in 1998. The W3C pinned its future plans on a specification called XHTML, which it intended to be HTML's cleaned-up, modernized successor. But XHTML stumbled, and a group of disenfranchised rebels resuscitated HTML, laying the groundwork for the features that you'll explore in this book.

In this chapter, you'll get the scoop on why HTML died and how it came back to life. You'll learn about HTML5's philosophy and features, and you'll consider the thorny issue of browser support. You'll also get your first look at an authentic HTML5 document.

The Story of HTML5

The basic idea behind HTML—that you use *elements* to structure your content—hasn't changed since the Web's earliest days. In fact, even the oldest web pages still work perfectly in the most modern web browsers.

Being old and successful also carries some risks—namely, that everyone wants to replace you. In 1998, the W3C stopped working on HTML and attempted to improve it with an XML-powered successor called XHTML 1.0.

XHTML 1.0: Getting Strict

XHTML has most of the same syntax conventions as HTML, but it enforces stricter rules. Much of the sloppy markup that traditional HTML permitted just isn't acceptable in XHTML.

For example, suppose you want to italicize the last word in a heading, like so:

```
<h1>The Life of a <i>Duck</i></h1>
```

And you accidentally swap the final two tags:

```
<h1>The Life of a <i>Duck</h1></i>
```

When a browser encounters this slightly messed-up markup, it can figure out what you really want. It italicizes the last word without even a polite complaint. However, the mismatched tags break XHTML's official rules. If you plug your page into an XHTML validator (or use a web design tool like Dreamweaver), you'll get a warning that points out your mistake. From a web design point of view, XHTML's strictness is helpful in that it lets you catch minor mistakes that might cause inconsistent results on different browsers (or might cause bigger problems when you edit and enhance the page).

At first, XHTML was a success story. Professional web developers, frustrated with browser quirks and the anything-goes state of web design, flocked to XHTML. Along the way, they were forced to adopt better habits and give up a few of HTML's half-baked formatting features. However, many of XHTML's imagined benefits—like interoperability with XML tools, easier page processing for automated programs, portability to mobile platforms, and extensibility of the XHTML language itself—never came to pass.

Still, XHTML became the standard for most serious web designers. And while everyone seemed pretty happy, there was one dirty secret: Although browsers understood XHTML markup, they didn't enforce the strict error-checking that the standard required. That means a page could break the rules of XHTML, and the browsers wouldn't blink twice. In fact, there was nothing to stop a web developer from throwing together a mess of sloppy markup and old-fashioned HTML content and calling it an XHTML page. There wasn't a single browser on the planet that would complain. And *that* made the people in charge of the XHTML standard deeply uncomfortable.

XHTML 2: The Unexpected Failure

XHTML 2 was supposed to provide a solution to this sloppiness. It was set to tighten up the error-handling rules, forcing browsers to reject invalid XHTML 2 pages. XHTML 2 also threw out many of the quirks and conventions that originated with HTML. For example, the system of numbered headings (`<h1>`, `<h2>`, `<h3>`, and so on) was superseded by a new `<h>` element, whose significance depended on its position in a web page. Similarly, the `<a>` element was eclipsed by a feature that let web developers transform any element into a link, and the `<img>` element lost its `alt` attribute in favor of a new way to supply alternate content.

These changes were typical of XHTML 2. In theory, they made for cleaner, more logical markup. In practice, the changes forced web designers to alter the way they wrote web pages (to say nothing of updating the web pages they already had), and added no new features to make all that work worthwhile. XHTML 2 even dumped a few well-worn elements that some web designers still loved, like `<b>` for bold text, `<i>` for italics, and `<iframe>` for embedding one web page inside another.

But perhaps the worst problem was the glacial pace of change. Development on XHTML 2 dragged on for five years, and developer enthusiasm slowly leaked away.

HTML5: Back from the Dead

At about the same time—starting in 2004—a group of people started looking at the future of the Web from a different angle. Instead of trying to sort out what was wrong (or just "philosophically impure") in HTML, they focused on what was missing, in terms of the things web developers wanted to get done.

After all, HTML began its life as a tool for displaying documents. With the addition of JavaScript, it had morphed into a system for developing web applications, like search engines, ecommerce stores, mapping tools, email clients, and a whole lot more. And while a crafty web application can do a lot of impressive things, it isn't easy to create one. Most web apps rely on a soup of handwritten JavaScript, one or more popular JavaScript toolkits, and a code module that runs on the web server. It's a challenge to get all these pieces to interact consistently on different browsers. Even when you get it to work, you need to mind the duct tape and staples that hold everything together.

The people creating browsers were particularly concerned about this situation. So a group of forward-thinking individuals from Opera Software (the creators of the Opera browser) and the Mozilla Foundation (the creators of Firefox) lobbied to get XHTML to introduce more developer-oriented features. When they failed, Opera, Mozilla, and Apple formed the loosely knit WHATWG (Web Hypertext Application Technology Working Group) to think of new solutions.

The WHATWG wasn't out to replace HTML, but to *extend* it in a seamless, backward-compatible way. The earliest version of its work had two add-on specifications called Web Applications 1.0 and Web Forms 2.0. Eventually, these standards evolved into HTML5.

NOTE The number *5* in the HTML5 specification name is supposed to indicate that the standard picks up where HTML left off (that's HTML version 4.01, which predates XHTML). Of course, this isn't really accurate, because HTML5 supports everything that's happened to web pages in the decade since HTML 4.01 was released, including strict XHTML-style syntax (if you choose to use it) and a slew of JavaScript innovations. However, the name still makes a clear point: HTML5 may support the *conventions* of XHTML, but it enforces the *rules* of HTML.

By 2007, the WHATWG camp had captured the attention of web developers everywhere. After some painful reflection, the W3C decided to disband the group that was working on XHTML 2 and work on formalizing the HTML5 standard instead. At

this point, the original HTML5 was broken into more manageable pieces, and many of the features that had originally been called HTML5 became separate standards (for more, see the box on this page).

TIP You can read the official W3C version of the HTML5 standard at *www.w3.org/TR/html5*.

UP TO SPEED

What Does HTML5 Include?

HTML5 is really a web of interrelated standards. This approach is both good and bad. It's good because the browsers can quickly implement mature features while others continue to evolve. It's bad because it forces web page writers to worry about checking whether a browser supports each feature they want to use. You'll learn some painful and not-so-painful techniques for doing so in this book.

Here are the major feature categories that fall under the umbrella of HTML5:

- **Core HTML5.** This part of HTML5 makes up the official W3C version of the specification. It includes the new semantic elements (Chapter 2 and Chapter 3), new and enhanced web form widgets (Chapter 4), audio and video support (Chapter 5), and the canvas for drawing with JavaScript (Chapter 8 and Chapter 9).
- **Features that were once HTML5.** These features sprang from the original HTML5 specification as prepared by the WHATWG. Most of these are specifications for features that require JavaScript and support rich web applications. The most significant include local data storage (Chapter 10), offline applications (Chapter 11), and messaging (Chapter 12), but you'll learn about several more in this book.
- **Features that are sometimes called HTML5.** These are next-generation features that are often lumped together with HTML5, even though they weren't ever a part of the HTML5 standard. This category includes CSS3 (Chapter 6 and Chapter 7) and geolocation (Chapter 13).

Even the W3C is blurring the boundaries between the "real" HTML5 (what's actually in the standard) and the "marketing" version (which includes everything that's part of HTML5 and many complementary specifications). For example, the official W3C logo website (*www.w3.org/html/logo*) encourages you to generate HTML5 logos that promote CSS3 and SVG—two standards that were under development well before HTML5 appeared.

HTML: The Living Language

The switch from the W3C to the WHATWG and back to the W3C again has led to a rather unusual arrangement. Technically, the W3C is in charge of determining what is and isn't official HTML5. But at the same time, the WHATWG continues its work dreaming up future HTML features. Only now, they no longer refer to their work as HTML5. They simply call it HTML, explaining that HTML will continue as a *living language*.

Because HTML is a living language, an HTML page will never become obsolete and stop working. HTML pages will never use a version number (even in the doctype), and web developers will never need to "upgrade" their markup from one version to another to get it to work on new browsers. By the same token, new features may be added to HTML at any time.

When web developers hear about this plan, their first reaction is usually unmitigated horror. After all, who wants to deal with a world of wildly variable standards support, where developers need to pick and choose the features they use based on the likelihood that these features will be supported? However, on reflection, most web developers come to a grudging realization: For better or for worse, this is exactly the way browsers have worked since the dawn of the Web.

As explained earlier, today's browsers are happy with any mishmash of supported features. You can take a state-of-the-art XHTML page and add something as scandalously backward as the `<marquee>` element (an obsolete feature for creating scrolling text), and no browser will complain. Similarly, browsers have well-known holes in their support for even the oldest standards. For example, browser makers started implementing CSS3 before CSS2 support was finished, and many CSS2 features were later dropped. The only difference is that now HTML5 makes the "living language" status official. Still, it's no small irony that just as HTML is embarking on a new, innovative chapter, it has finally returned full circle to its roots.

TIP To see the current, evolving draft of HTML that includes the stuff called HTML5 and a small but ever-evolving set of new, unsupported features, go to *http://whatwg.org/html*.

Three Key Principles of HTML5

By this point, you're probably eager to get going with a real HTML5 page. But first, it's worth climbing into the minds of the people who built HTML5. Once you understand the philosophy behind the language, the quirks, complexities, and occasional headaches that you'll encounter in this book will make a whole lot more sense.

1. Don't Break the Web

"Don't break the Web" means that a standard shouldn't introduce changes that make other people's web pages stop working. Fortunately, this kind of wreckage rarely happens.

"Don't break the Web" *also* means that a standard shouldn't casually change the rules, and in the process make perfectly good current-day web pages to be obsolete (even if they still happen to work). For example, XHTML 2 broke the Web because it demanded an immediate, dramatic shift in the way web pages were written. Yes, old pages would still work—thanks to the backward compatibility that's built into browsers. But if you wanted to prepare for the future and keep your website up to date, you'd be forced to waste countless hours correcting the "mistakes" that XHTML 2 had banned.

HTML5 has a different viewpoint. Everything that was valid before HTML5 remains valid in HTML5. In fact, everything that was valid in HTML 4.01 also remains valid in HTML5.

NOTE Unlike previous standards, HTML5 doesn't just tell browser makers what to support—it also documents and formalizes the way they *already work*. Because the HTML5 standard documents reality, rather than just setting out a bunch of ideal rules, it may become the best-supported web standard ever.

UP TO SPEED

How HTML5 Handles Obsolete Elements

Because HTML5 supports all of HTML, it supports many features that are considered obsolete. These include formatting elements like `<font>`, despised special-effect elements like `<blink>` and `<marquee>`, and the awkward system of HTML frames.

This open-mindedness is a point of confusion for many HTML5 apprentices. On the one hand, HTML5 should by all rights ban these outdated elements, which haven't appeared in an official specification for years (if ever). On the other hand, modern browsers still quietly support these elements, and HTML5 is supposed to reflect how web browsers really work. So what's a standard to do?

To solve this problem, the HTML5 specification has two separate parts. The first part—which is what you'll consider in this book—targets web developers. Developers need to avoid the bad habits and discarded elements of the past. You can make sure you're following this part of the HTML5 standard by using an HTML5 validator.

The second, much longer part of the HTML5 specification targets browser makers. Browsers need to support everything that's ever existed in HTML, for backward compatibility. Ideally, the HTML5 standard should have enough information that someone could build a browser from scratch and make it completely compatible with the modern browsers of today, whether it was processing new or old markup. This part of the standard tells browsers how to deal with obsolete elements that are officially discouraged but still supported.

Incidentally, the HTML5 specification also formalizes how browsers should deal with a variety of errors (for example, missing or mismatched tags). This point is important, because it ensures that a flawed page will work the same on different browsers, even when it comes to subtle issues like the way a page is modeled in the DOM (that's the Document Object Model, the tree of in-memory objects that represents the page and is made available to JavaScript code). To create this long, tedious part of the standard, the creators of HTML5 performed exhaustive tests on modern browsers to figure out their undocumented error-handling behavior. Then, they wrote it down.

2. Pave the Cowpaths

A cowpath is the rough, heavily trodden track that gets people from one point to another. A cowpath exists because it's being used. It might not be the best possible way to move around, but at some point it was the most practical working solution.

HTML5 standardizes these unofficial (but widely used) techniques. It may not be as neat as laying down a nicely paved expressway with a brand-new approach, but it has a better chance of succeeding. That's because switching over to new techniques may be beyond the ability or interest of the average website designer. And worse, new techniques may not work for visitors who are using older browsers. XHTML 2 tried to drive people off the cowpaths, and it failed miserably.

NOTE Paving the cowpaths has an obvious benefit: It uses established techniques that already have some level of browser support. If you give web developers a choice between a beautifully designed new feature that works on 70 percent of the web browsers out there and a messy hack that works everywhere, they'll choose the messy hack and the bigger audience every time.

The "pave the cowpaths" approach also requires some compromises. Sometimes it means embracing a widely supported but poorly designed feature. One example is HTML5's drag-and-drop ability (page 337), which is based entirely on the behavior Microsoft created for IE 5. Although this drag-and-drop feature is now supported in all browsers, it's universally loathed for being clumsy and overly complicated. This magnanimousness has led some web designers to complain that "HTML5 not only encourages bad behavior, it defines it."

3. Be Practical

This principle is simple: Changes should have a practical purpose. And the more demanding the change, the bigger the payoff needs to be. Web developers may prefer nicely designed, consistent, quirk-free standards, but that isn't a good enough reason to change a language that's already been used to create several billion pages. Of course, it's still up to someone to decide whose concerns are the most important. A good clue is to look at what web pages are already doing—or trying to do.

For example, the world's third most popular website (at the time of this writing) is YouTube. But because HTML had no real video features before HTML5, YouTube has had to rely on the Flash browser plug-in. This solution works surprisingly well because the Flash plug-in is present on virtually all web-connected computers. However, there are occasional exceptions, like locked-down corporate computers that don't allow Flash, or mobile devices that don't support it (like the iPhone, iPad, and Kindle). And no matter how many computers have Flash, there's a good case for extending the HTML standard so it directly supports one of the most fundamental ways people use web pages today—to watch video.

There's a similar motivation behind HTML5's drive to add more interactive features—drag-and-drop support, editable HTML content, two-dimensional drawing on a canvas, and so on. You don't need to look far to find web pages that use all of these features right now, some with plug-ins like Adobe Flash and Microsoft Silverlight, and others with JavaScript libraries or (more laboriously) with pages of custom-written JavaScript code. So why not add official support to the HTML standard and make sure these features work consistently on all browsers? That's what HTML5 sets out to do.

NOTE Browser plug-ins like Flash won't go away overnight. Despite its many innovations, it still takes far more work to build complex, graphical applications in HTML5. But HTML5's ultimate vision is clear: to allow websites to offer video, rich interactivity, and piles of frills without requiring a plug-in.

Your First Look at HTML5 Markup

Here's one of the simplest HTML5 documents you can create:

```
<!DOCTYPE html>
<title>A Tiny HTML Document</title>
<p>Let's rock the browser, HTML5 style.</p>
```

It starts with the HTML5 doctype (a special code that's explained on page 11), followed by a title, and then followed by some content. In this case, the content is a single paragraph of text.

You already know what this looks like in a browser, but if you need reassuring, check out Figure 1-1.

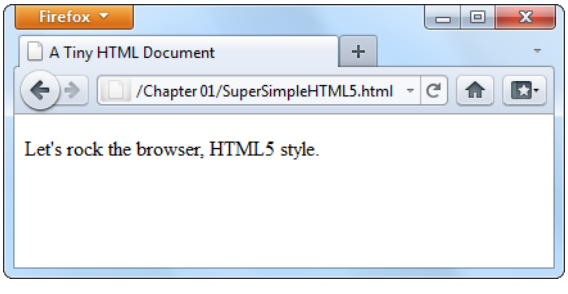


FIGURE 1-1
This super-simple HTML5 document holds a single line of text.

You can pare down this document a bit more. For example, the HTML5 standard doesn't really require the final </p> tag, since browsers know to close all open elements at the end of the document (and the HTML5 standard makes this behavior official). However, shortcuts like these create confusing markup and can lead to unexpected mistakes.

The HTML5 standard also lets you omit the <title> element if the title information is provided in another way. For example, if you're sending an HTML document in an email message, you could put the title in the title of the email message and put the rest of the markup—the doctype and the content—into the body of the message. But this is obviously a specialized scenario.

More commonly, you'll want to flesh out this bare-bones HTML5 document. Most web developers agree that using the traditional <head> and <body> sections can prevent confusion, by cleanly separating the information about your page (the head) and its actual content (the body). This structure is particularly useful when you start adding scripts, style sheets, and meta elements.

```
<!DOCTYPE html>
<head>
  <title>A Tiny HTML Document</title>
</head>
<body>
  <p>Let's rock the browser, HTML5 style.</p>
</body>
```

As always, the indenting (at the beginning of lines three and six) is purely optional. This example uses it to make the structure of the page easier to see at first glance.

Finally, you can choose to wrap the entire document (not including the doctype) in the traditional <html> element. Here's what that looks like:

```
<!DOCTYPE html>
<html>
<head>
  <title>A Tiny HTML Document</title>
</head>
<body>
  <p>Let's rock the browser, HTML5 style.</p>
</body>
</html>
```

Up until HTML5, every version of the official HTML specification had demanded that you use the <html> element, despite the fact that it has no effect on browsers. However, HTML5 makes this detail completely optional.

NOTE The use of the <html>, <head>, and <body> elements is simply a matter of style. You can leave them out and your page will work perfectly well, even on old browsers that don't know a thing about HTML5. In fact, the browser will automatically assume these details. So if you use JavaScript to peek at the DOM (the set of programming objects that represents your page), you'll find objects for the <html>, <head>, and <body> elements, even if you didn't add them yourself.

Currently, this example is somewhere between the simplest possible HTML5 document and the fleshed-out starting point of a practical HTML5 web page. In the following sections, you'll fill in the rest of what you need and dig a little deeper into the markup.

The HTML5 Doctype

The first line of every HTML5 document is a special code called the *doctype*. The doctype clearly indicates the standard that was used to write the document markup that follows. Here's how a page announces that it adheres to the HTML5 standard:

```
<!DOCTYPE html>
```

The first thing you'll notice about the HTML5 doctype is its striking simplicity. Compare it, for example, to the ungainly doctype that web developers need when using XHTML 1.0 strict:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Even professional web developers were forced to copy and paste the XHTML doctype from one document to another. But the HTML5 doctype is short and snappy, so you won't have much trouble typing it by hand.

The HTML5 doctype is also notable for the fact that it doesn't include the official specification version (that's the *5* in HTML5). Instead, the doctype simply indicates that the page is HTML, which is in keeping with the new vision of HTML5 as a living language (page 6). When new features are added to the HTML language, they're automatically available in your page, without requiring you to edit the doctype.

All of this raises a good question—if HTML5 is a living language, why does your web page require any doctype at all?

The answer is that the doctype remains for historical reasons. Without a doctype, most browsers (including Internet Explorer and Firefox) will lapse into *quirks mode*. In this mode, they'll attempt to render pages according to the slightly buggy rules that they used in older versions. The problem is that one browser's quirks mode differs from the next, so pages designed for one browser are likely to get inconsistently sized fonts, scrambled layouts, and other glitches on another browser.

When you add a doctype, the browser recognizes that you want to use the stricter *standards mode*, which ensures that the web page is displayed with consistent formatting and layout on every modern browser. The browser doesn't even care *which* doctype you use (with just a few exceptions). Instead, it simply checks that you have *some* doctype. The HTML5 doctype is simply the shortest valid doctype, so it always triggers standards mode.

TIP The HTML5 doctype triggers standards mode on all browsers that have a standards mode, including browsers that don't know anything about HTML5. For that reason, you can use the HTML5 doctype now, in all your pages, even if you need to hold off on some of HTML5's less-supported features.

Although the doctype is primarily intended to tell web browsers what to do, other agents can also check it. This includes HTML5 validators, search engines, design tools, and other human beings when they're trying to figure out what flavor of markup you've chosen for your page.

Character Encoding

The *character encoding* is the standard that tells a computer how to convert your text into a sequence of bytes when it's stored in a file—and how to convert it back again when the file is opened. For historical reasons, there are many different character encodings in the world. Today, virtually all English websites use an encoding

called UTF-8, which is compact, fast, and supports all the non-English characters you'll ever need.

Often, the web server that hosts your pages is configured to tell browsers that it's serving out pages with a certain kind of encoding. However, because you can't be sure that your web server will take this step (unless you own the server), and because browsers can run into an obscure security issue when they attempt to guess a page's encoding, you should always add encoding information to your markup.

HTML5 makes that easy to do. All you need to do is add the `<meta>` element shown below at the very beginning of your `<head>` section (or right after the doctype, if you don't define the `<head>` element):

```
<head>
  <meta charset="utf-8">
  <title>A Tiny HTML Document</title>
</head>
```

Design tools like Dreamweaver add this detail automatically when you create a new page. They also make sure that your files are being saved with UTF encoding. However, if you're using an ordinary text editor, you may need to take an extra step to make sure your files are being saved correctly. For example, when editing an HTML file in Notepad (on Windows), in the Save As dialog box, you must choose UTF-8 from the Encoding list (at bottom). In TextEdit (on Mac), in the Save As dialog box, you need to first choose Format→Make Plain Text to make sure the program saves your page as an ordinary text file, and then choose "Unicode (UTF-8)" from the Plain Text Encoding pop-up menu.

The Language

It's considered good style to indicate your web page's *natural language*. This information is occasionally useful to other people—for example, search engines can use it to filter search results so they include only pages that match the searcher's language.

To specify the language of some content, you use the `lang` attribute on any element, along with the appropriate language code. That's *en* for plain English, but you can find more exotic language codes at *http://tinyurl.com/l-codes*.

The easiest way to add language information to your web page is to use the `<html>` element with the `lang` attribute:

```
<html lang="en">
```

This detail can also help screen readers if a page has text from multiple languages. In this situation, you use the `lang` attribute to indicate the language of different sections of your document; for example, by applying it to different `<div>` elements that wrap different content. Screen readers can then determine which sections to read aloud.

Adding a Style Sheet

Virtually every web page in a properly designed, professional website uses CSS style sheets. You specify the style sheets you want to use by adding <link> elements to the <head> section of an HTML5 document, like this:

```
<head>
  <meta charset="utf-8">
  <title>A Tiny HTML Document</title>
  <link href="styles.css" rel="stylesheet">
</head>
```

This method is more or less the same way you attach style sheets to a traditional HTML document, but slightly simpler.

NOTE Because CSS is the only style sheet language around, there's no need to add the type="text/css" attribute that web pages used to require.

Adding JavaScript

JavaScript started its life as a way to add frivolous glitter and glamour to web pages. Today, JavaScript is less about user interface frills and more about novel web applications, including super-advanced email clients, word processors, and mapping engines that run right in the browser.

You add JavaScript to an HTML5 page in much the same way that you add it to a traditional HTML page. Here's an example that references an external file with JavaScript code:

```
<head>
  <meta charset="utf-8">
  <title>A Tiny HTML Document</title>
  <script src="scripts.js"></script>
</head>
```

There's no need to include the language="JavaScript" attribute. The browser assumes you want JavaScript unless you specify otherwise—and because JavaScript is the only HTML scripting language with broad support, you never will. However, you *do* still need to remember the closing </script> tag, even when referring to an external JavaScript file. If you leave it out or attempt to shorten your markup using the empty element syntax, your page won't work.

If you spend a lot of time testing your JavaScript-powered pages in Internet Explorer, you may also want to add a special comment called the *mark of the Web* to your <head> section, right after the character encoding. It looks like this:

```
<head>
  <meta charset="utf-8">
  <!-- saved from url=(0014)about:internet -->
  <title>A Tiny HTML Document</title>
  <script src="scripts.js"></script>
</head>
```

This comment tells Internet Explorer to treat the page as though it has been downloaded from a remote website. Otherwise, IE switches into a special locked-down mode, pops up a security warning in a message bar, and won't run any JavaScript code until you explicitly click "Allow blocked content."

All other browsers ignore the "mark of the Web" comment and use the same security settings for remote websites and local files.

The Final Product

If you've followed these steps, you'll have an HTML5 document that looks something like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>A Tiny HTML Document</title>
  <link href="styles.css" rel="stylesheet">
  <script src="scripts.js"></script>
</head>

<body>
  <p>Let's rock the browser, HTML5 style.</p>
</body>
</html>
```

Although it's no longer the shortest possible HTML5 document, it's a reasonable starting point for any web page you want to build. And while this example seems wildly dull, don't worry—in the next chapter, you'll step up to a real-life page that's full of carefully laid-out content, and all wrapped up in CSS.

NOTE All the HTML5 syntax you've learned about in this section—the new doctype, the meta element for character encoding, the language information, and the style sheet and JavaScript references, work in browsers both new and old. That's because they rely on defaults and built-in error-correcting practices that all browsers use.

A Closer Look at HTML5 Syntax

As you've already learned, HTML5 loosens some of the rules. That's because the creators of HTML5 wanted the language to more closely reflect web browser reality—in other words, they wanted to narrow the gap between "web pages that work" and "web pages that are considered valid, according to the standard." In the next section, you'll take a closer look at how the rules have changed.

NOTE There are still plenty of obsolete practices that browsers support but that the HTML5 standard strictly discourages. For help catching these in your own web pages, you'll need an HTML5 validator (page 17).

The Loosened Rules

In your first walk through an HTML5 document, you discovered that HTML5 makes the <html>, <head>, and <body> elements optional (although they can still be pretty useful). But HTML5's relaxed attitude doesn't stop there.

HTML5 ignores capitalization, letting you write markup like the following:

```
<P>Capital and lowercase letters <EM>don't matter</eM> in tag names.</p>.
```

HTML5 also lets you omit the closing slash from a *void element*—that's an element with no nested content, like an <img> (image), a
 (line break), or an <hr> (horizontal line). Here are three equivalent ways to add a line break:

```
I cannot<br />
move backward<br>
or forward.<br/>
I am caught
```

HTML5 also changes the rules for attributes. Attribute values don't need quotation marks anymore, as long as the value doesn't include a restricted character (typically >, =, or a space). Here's an example of an <img> element that takes advantage of this ability:

```
<img alt="Horsehead Nebula" src=Horsehead01.jpg>
```

Attributes with no values are also allowed. So while XHTML required the somewhat redundant syntax to put a checkbox in the checked state...

```
<input type="checkbox" checked="checked" />
```

...you can now revive the shorter HTML 4.01 tradition of including the attribute name on its own.

```
<input type="checkbox" checked>
```

What's particularly disturbing to some people isn't the fact that HTML5 allows these things. It's the fact that inconsistent developers can casually switch back and forth between the stricter and the looser styles, even using both in the same document. In reality, though, XHTML permitted the same kind of inconsistency. In both cases,

good style is the responsibility of the web designer, and the browser tolerates whatever you can throw at it.

Here's a quick summary of what constitutes good HTML5 style—and what conventions the examples in this book follow, even if they don't have to:

- **Including the optional <html>, <body>, and <head> elements.** The <html> element is a handy place to define the page's natural language (page 13); and the <body> and <head> elements help to keep page content separate from the other page details.
- **Using lowercase tags (like <p> instead of <P>).** They're not necessary, but they're far more common, easier to type (because you don't need the Shift key), and not nearly as shouty.
- **Using quotation marks around attribute values.** The quotation marks are there for a reason—to protect you from mistakes that are all too easy to make. Without quotation marks, one invalid character can break your whole page.

On the other hand, there are some old conventions that this book ignores (and you can, too). The examples in this book don't close empty elements, because most developers don't bother to add the extra slash (/) when they switch to HTML5. Similarly, there's no reason to favor the long attribute form when the attribute name and the attribute value are the same.

HTML5 Validation

HTML5's new, relaxed style may suit you fine. Or, the very thought that there could be inconsistent, error-ridden markup hiding behind a perfectly happy browser may be enough to keep you up at night. If you fall into the latter camp, you'll be happy to know that a validation tool can hunt down markup that doesn't conform to the recommended standards of HTML5, even if it doesn't faze a browser.

Here are some potential problems that a validator can catch:

- Missing mandatory elements (for example, the <title> element)
- A start tag without a matching end tag
- Incorrectly nested tags
- Tags with missing attributes (for example, an <img> element without the src attribute)
- Elements or content in the wrong place (for example, text that's placed directly in the <head> section)

Web design tools like Dreamweaver often have their own validators. But if you don't want the cost or complexity of a professional web editor, you can get the same information from an online validation tool. Here's how to use the popular validator provided by the W3C standards organization:

1. **In your web browser, go to *http://validator.w3.org* (Figure 1-2).**

   The W3C validator gives you three choices, represented by three separate tabs: "Validate by URI" (for a page that's already online), "Validate by File Upload" (for a page that's stored in a file on your computer), and "Validate by Direct Input" (for a bunch of markup you type in yourself).

2. **Click the tab you want, and supply your HTML content.**

   - **Validate by URI** lets you validate an existing web page. You just need to type the page's URL in the Address box (for example, *http://www.MySloppySite.com/FlawedPage.html*).
   - **Validate by File Upload** lets you upload any file from your computer. First, click the Browse button (in Chrome, click Choose File). In the Open dialog box, select your HTML file and then click Open.
   - **Validate by Direct Input** lets you validate any markup—you just need to type it into a large box. The easiest way to use this option is to copy the markup from your text editor and paste it into the box on the W3C validation page.

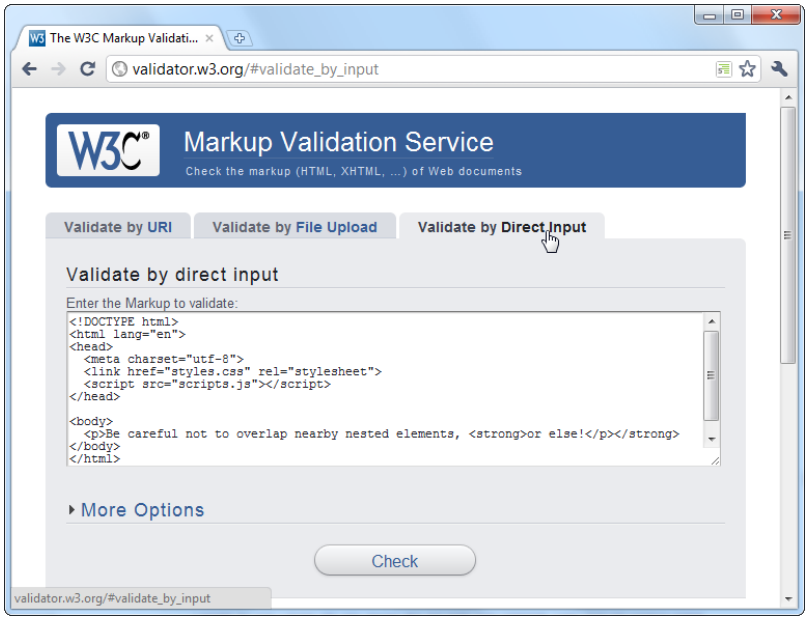


FIGURE 1-2

The website http://validator.w3.org *gives you three options for validating HTML. You can fill in the address of another web page, you can upload a file of your own, or you can type the markup in directly (shown here).*

Before continuing, you can click More Options to change some settings, but you probably won't. It's best to let the validator automatically detect the document type—that way, the validator will use the doctype specified in your web page. Similarly, use automatic detection for the character set unless you have

an HTML page that's written in another language and the validator has trouble determining the correct character set.

3. **Click the Check button.**

   This click sends your HTML page to the W3C validator. After a brief delay, the report appears. You'll see whether your document passed the validation check and, if it failed, what errors the validator detected (see Figure 1-3).

NOTE Even in a perfectly valid HTML document, you may get a few harmless warnings, including that the character encoding was determined automatically and that the HTML5 validation service is considered to be an experimental, not-fully-finished feature.

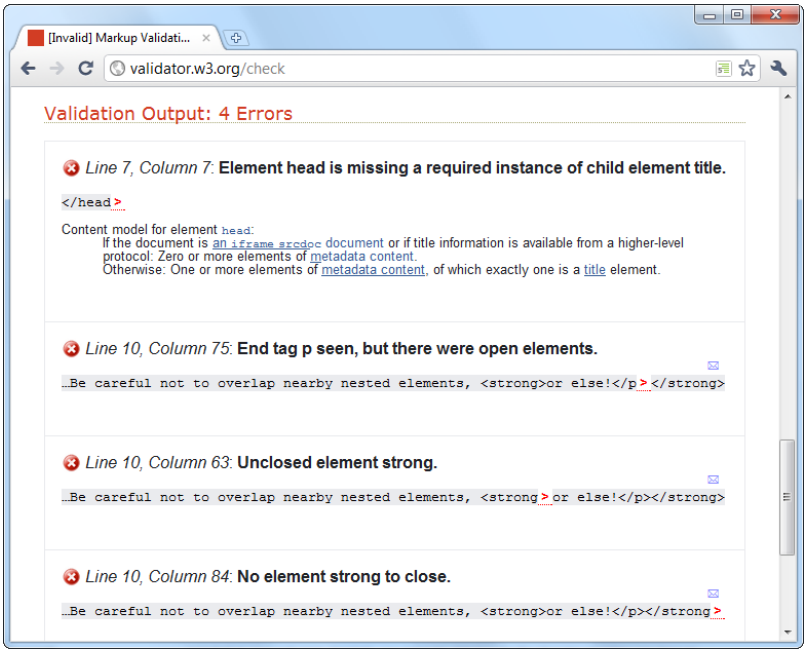


FIGURE 1-3
The validator has discovered four errors in this document that stem from two mistakes. First, the page is missing the mandatory <title> element. Second, it closes the element before closing the element that's nested inside. (To solve this problem, you would replace with .) Incidentally, this document is still close enough to being correct that all browsers will display it properly.

The Return of XHTML

As you've already learned, HTML5 spells the end for the previous king of the Web—XHTML. However, reality isn't quite that simple, and XHTML fans don't need to give up all the things they loved about the past generation of markup languages.

First, remember that XHTML syntax lives on. The rules that XHTML enforced either remain as guidelines (for example, nesting elements correctly) or are still supported as optional conventions (for example, including the trailing slash on empty elements).

But what if you want to *enforce* the XHTML syntax rules? Maybe you're worried that you (or the people you work with) will inadvertently slip into the looser conventions of ordinary HTML. To stop that from happening, you need to use XHTML5—a less common standard that is essentially HTML5 with the XML-based restrictions slapped on top.

To turn an HTML5 document into an XHTML5 document, you need to explicitly add the XHTML namespace to the `<html>` element, close every element, make sure you use lowercase tags, and so on. Here's an example of a web page that takes all these steps:

```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta charset="utf-8"/>
  <title>A Tiny HTML Document</title>
  <link href="styles.css" rel="stylesheet"/>
  <script src="scripts.js"></script>
</head>

<body>
  <p>Let's rock the browser, XHTML5 style.</p>
</body>
</html>
```

Now you can use an XHTML5 validator to get stricter error checking that enforces the old-style XHTML rules. The standard W3C validator won't do it, but the validator at *http://validator.w3.org/nu* will, provided you click the Options button and choose XHTML5 from the Preset list. You also need to choose the "Be lax about content-type" option, unless you're using the direct input approach and pasting your markup into a text box.

By following these steps, you can create and validate an XHTML document. However, *browsers* will still process your page as an HTML5 document—one that just happens to have an XML inferiority complex. They won't attempt to apply any extra rules.

If you want to go XHTML5 all the way, you need to configure your web server to serve your page with the MIME type `application/xhtml+xml` or `application/xml`, instead of the standard `text/html`. (See page 152 for the lowdown on MIME types.) But before you call your web hosting company, be warned that this change will prevent your page from being displayed by any version of Internet Explorer before IE 9. For that reason, true XHTML5 is an immediate deal-breaker in the browser.

Incidentally, browsers that do support XHTML5 deal with it differently from ordinary HTML5. They attempt to process the page as an XML document, and if that process fails (because you've left a mistake behind), the browser gives up on the rest of the document.

Bottom line? For the vast majority of web developers, from ordinary people to serious pros, XHTML5 isn't worth the hassle. The only exceptions are developers who have a

specific XML-related goal in mind; for example, developers who want to manipulate the content in their pages with XML-related standards like XQuery and XPath.

TIP If you're curious, you can trick your browser into switching into XHTML mode. Just rename your file so that it ends with .xhtml or .xht. Then open it from your hard drive. Most browsers (including Firefox, Chrome, and IE 9 or later) will act as though you downloaded the page from a web server with an XML MIME type. If there's a minor error in the page, the browser window will show a partially processed page (IE), an XML error message (Firefox), or a combination of the two (Chrome).

HTML5's Element Family

So far, this chapter has focused on the changes to HTML5's syntax. But more important are the additions, subtractions, and changes to the *elements* that HTML supports. In the following sections, you'll get an overview of how they've changed.

Added Elements

In the following chapters, you'll spend most of your time learning about new elements—ingredients that haven't existed in web pages up until now. Table 1-1 has a preview of what's in store (and where you can read more about it).

TABLE 1-1 *New HTML5 elements*

CATEGORY	ELEMENTS	DISCUSSED IN...
Semantic elements for structuring a page	`<article>`, `<aside>`, `<figcaption>`, `<figure>`, `<footer>`, `<header>`, `<nav>`, `<section>`, `<details>`, `<summary>`	Chapter 2
Semantic elements for text	`<mark>`, `<time>`, `<wbr>` (previously supported, but now an official part of the language)	Chapter 3
Web forms and interactivity	`<input>` (not new, but has many new subtypes) `<datalist>`, `<keygen>`, `<meter>`, `<progress>`, `<command>`, `<menu>`, `<output>`	Chapter 4
Audio, video, and plug-ins	`<audio>`, `<video>`, `<source>`, `<embed>` (previously supported, but now an official part of the language)	Chapter 5
Canvas	`<canvas>`	Chapter 8
Non-English language support	`<bdo>`, `<rp>`, `<rt>`, `<ruby>`	HTML5 specification at *http://dev.w3.org/html5/markup*

Removed Elements

Although HTML5 adds new elements, it also boots a few out of the official family. These elements will keep working in browsers, but any decent HTML5 validator will smoke them out of their hiding places and complain loudly.

Most obviously, HTML5 keeps the philosophy (first cooked up with XHTML) that *presentational elements* are not welcome in the language. Presentational elements are elements that are simply there to add formatting to web pages, and even the greenest web designer knows that's a job for style sheets. Rejects include elements that professional developers haven't use in years (like <big>, , <font>, <tt>, and <strike>). HTML's presentational attributes died the same death, so there's no reason to rehash them all here.

Additionally, HTML5 kicks more sand on the grave where web developers buried the HTML frames feature. When it was first created, HTML frames seemed like a great way to show multiple web pages in a single browser window. But now, frames are better known as an accessibility nightmare because they cause problems with search engines, assistive software, and mobile devices. Interestingly, the <iframe> element—which lets developers put one page inside another—squeaks through. That's because web applications use the <iframe> for a range of integration tasks, like incorporating YouTube windows, ads, and Google search boxes in a web page.

A few more elements were kicked out because they were redundant or the cause of common mistakes, including <acronym> (use <abbr> instead) and <applet> (because <object> is preferred). But the vast majority of the element family lives on in HTML5.

NOTE For those keeping count, HTML5 includes a family of just over 100 elements. Out of these, almost 30 are new and about 10 are significantly changed. You can browse the list of elements (and review which ones are new or changed) at *http://dev.w3.org/html5/markup*.

Adapted Elements

HTML5 has another odd trick: Sometimes it adapts an old feature to a new purpose. For example, consider the <small> element, which fell out of favor as a clumsy way to shrink the font size of a block of text—a task more properly done with style sheets. But unlike the discarded <big> element, HTML5 keeps the <small> element, with a change. Now, the <small> element represents "small print"—for example, the legalese that no one wants you to read at the bottom of a contract:

```
<small>The creators of this site will not be held liable for any injuries that
may result from unsupervised unicycle racing.</small>
```

Text inside the <small> element is still displayed as it always was, using a smaller font size, unless you override that setting with a style sheet.

NOTE Opinions on this <small> technique differ. On the one hand, it's great for backward compatibility, because old browsers already support the <small> element, and so they'll continue to support it in an HTML5 page. On the other hand, it introduces a potentially confusing change of meaning for old pages. They may be using the <small> element for presentational purposes, without wanting to suggest "small print."

Another changed element is <hr> (short for horizontal rule), which draws a separating line between sections. In HTML5, <hr> represents a thematic break—for example, a transition to another topic. The default formatting stays, but now a new meaning applies.

Similarly, <s> (for struck text), isn't just about crossing out words anymore—it now represents text that is no longer accurate or relevant, and has been "struck" from the document. Both of these changes are subtler than the <small> element's shift in meaning, because they capture ways that the <hr> and <s> elements are commonly used in traditional HTML.

BOLD AND ITALIC FORMATTING

The most important adapted elements are the ones for bold and italic formatting. Two of HTML's most commonly used elements—that's <b> for bold and <i> for italics—were partially replaced when the first version of XHTML introduced the look-alike and elements. The idea was to stop looking at things from a formatting point of view (bold and italics), and instead substitute elements that had a real logical meaning (strong importance or stressed emphasis). The idea made a fair bit of sense, but the <b> and <i> tags lived on as shorter and more familiar alternatives to the XHTML fix.

HTML5 takes another crack at solving the problem. Rather than trying to force developers away from <b> and <i>, it assigns new meaning to both elements. The idea is to allow all four elements to coexist in a respectable HTML5 document. The result is the somewhat confusing set of guidelines listed here:

- Use **** for text that has *strong importance*. This is text that needs to stand out from its surroundings.
- Use **<b>** for text that should be presented in bold but doesn't have greater importance than the rest of your text. This could include keywords, product names, and anything else that would be bold in print.
- Use **** for text that has *emphatic stress*—in other words, text that would have a different inflection if read out loud.
- Use **<i>** for text that should be presented in italics but doesn't have extra emphasis. This could include foreign words, technical terms, and anything else that you'd set in italics in print.

And here's a snippet of markup that uses all four of these elements in the appropriate way:

```
<strong>Breaking news!</strong> There's a sale on <i>leche quemada</i> candy
at the <b>El Azul</b> restaurant. Don't delay, because when the last candy
is gone, it's <em>gone</em>.
```

In the browser, the text looks like this:

> **Breaking news!** There's a sale on *leche quemada* candy at the **El Azul** restaurant. Don't delay, because when the last candy is gone, it's *gone*.

Some web developers will follow HTML's well-intentioned rules, while others just stick with the most familiar elements for bold and italic formatting.

Tweaked Elements

HTML5 also shifts the rules of a few elements. Usually, these changes are minor details that only HTML wonks will notice, but occasionally they have deeper effects. One example is the rarely used <address> element, which is not suitable (despite the name) for postal addresses. Instead, the <address> element has the narrow purpose of providing contact information for the creator of the HTML document, usually as an email address or website link:

```
Our website is managed by:
<address>
<a href="mailto:jsolo@mysite.com">John Solo</a>,
<a href="mailto:lcheng@mysite.com">Lisa Cheng</a>, and
<a href="mailto:rpavane@mysite.com">Ryan Pavane</a>.
</address>
```

The <cite> element has also changed. It can still be used to cite some work (for example, a story, article, or television show), like this:

```
<p>Charles Dickens wrote <cite>A Tale of Two Cities</cite>.</p>
```

However, it's not acceptable to use <cite> to mark up a person's name. This restriction has turned out to be surprisingly controversial, because this usage was allowed before. Several guru-level web developers are on record urging people to disregard the new <cite> rule, which is a bit odd, because you can spend a lifetime editing web pages without ever stumbling across the <cite> element in real life.

A more significant tweak affects the <a> element for creating links. Past versions of HTML have allowed the <a> element to hold clickable text or a clickable image. In HTML5, the <a> element allows anything and everything, which means it's perfectly acceptable to stuff entire paragraphs in there, along with lists, images, and so on. (If you do, you'll see that all the text inside becomes blue and underlined, and all the images inside sport blue borders.) Web browsers have supported this behavior for years, but it's only HTML5 that makes it an official, albeit not terribly useful, part of the HTML standard.

There are also some tweaks that don't work yet—in any browser. For example, the <ol> element (for ordered lists) now gets a reversed attribute, which you can set to count backward (either toward 1, or toward whatever starting value you set with the start attribute), but currently there are only two browsers that recognize this setting—Chrome and Safari.

You'll learn about a few more tweaks as you make your way through this book.

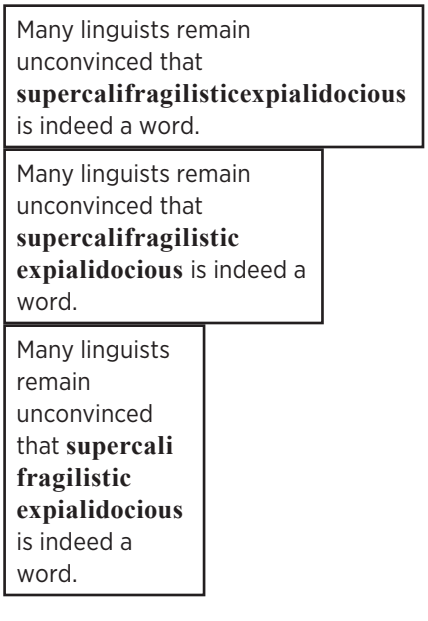
Standardized Elements

HTML5 also adds supports for a few elements that were supported but weren't officially welcome in the HTML or XHTML language. One of the best-known examples is <embed>, which is used all over the Web as an all-purpose way to shoehorn a plug-in into a page.

A more exotic example is <wbr>, which indicates an optional word break—in other words, a place where the browser can split a line if the word is too long to fit in its container:

```
<p>Many linguists remain unconvinced that
<b>supercali<wbr>fragilistic<wbr>expialidocious</b> is indeed a word.</p>
```

The <wbr> element is useful when you have long names (sometimes seen in programming terminology) in small places, like table cells or tiny boxes. Even if the browser supports <wbr>, it will break the word only if it doesn't fit in the available space. In the previous example, that means the browser may render the word in one of the following ways:



The <wbr> element has a natural similarity to the <nobr> element, which prevents text from wrapping no matter how narrow the available space. However, HTML5 considers <nobr> obsolete and advises all self-respecting web developers to avoid using it. Instead, you can get the same effect by adding the white-space property to your style sheet and setting it to nowrap.

Using HTML5 Today

Before you commit to HTML5, you need to know how well it works with the browsers your visitors are likely to use. After all, the last thing any web developer wants is a shiny new page that collapses into a muddle of scrambled markup and script errors when it meets a vintage browser.

In a moment, you'll learn how to research specific HTML5 features to find out which browsers support them, and examine browser usage statistics to find out what portion of your audience meets the bar. But before digging into the fine details, here's a broad overview of the current state of HTML5 support:

- If your visitors use the popular Google Chrome or Mozilla Firefox, they'll be fine. Not only have both browsers supported the bulk of HTML5 for several years, but they're also designed to update themselves automatically. That means you're unlikely to find an old version of Chrome or Firefox in the wild.
- If your visitors use Safari or Opera, you're probably still on safe ground. Once again, these browsers have had good HTML5 support for several years, and old versions are rarely seen.
- If your visitors use tablet computers or smartphones, you may face some limitations with certain features, as you'll learn throughout this book. However, the mobile browsers on all of today's web-enabled gadgets were created with HTML5 in mind. That means your pages are in for maybe a few hiccups, not a horror show.
- If your visitors use an older version of Internet Explorer—that is, any version before IE 10—most HTML5 features *won't* work. Here's where the headaches come in. Old versions of Windows are still common, and they typically include old versions of Internet Explorer. Even worse, many old versions of Windows don't let their users upgrade to a modern, HTML5-capable version of IE. Windows Vista, for example, is limited to IE 9. The mind-bogglingly old (but still popular) Windows XP is stuck with IE 8.

  No, it's not Microsoft's diabolical plan to break the Web—it's just that newer versions of IE were designed with newer computer hardware in mind. This new software simply won't work on old machines. But people with old versions of Windows can use an alternative browser like Firefox, although they may not know how to install it or may not be allowed to make such changes to a company computer.

NOTE Although really old versions of Internet Explorer—like IE 6 and IE 7—have finally disappeared from the scene, the problematic IE 8 and IE 9 still account for over 10 percent of all Web traffic (at the time of this writing). And because it's never OK to force one in ten website visitors to suffer, you'll need to think about workarounds for most HTML5 features—at least for the immediate future.

UP TO SPEED

Dealing with Old Browsers

For the next few years, some of your visitors' browsers won't support all the HTML5 features you want to use. That's a fact of life. But it doesn't need to prevent you from using these features, if you're willing to put in a bit more work. There are two basic strategies you can use:

- **Degrade gracefully.** Sometimes, when a feature doesn't work, it's not a showstopper. For example, HTML5's new `<video>` element has a fallback mechanism that lets you supply something else to older browsers, like a video player that uses the Flash plug-in. (Supplying an error message is somewhat rude, and definitely not an example of degrading gracefully.) Your page can also degrade gracefully by ignoring nonessential frills, like some of the web form features (like placeholder text) and some of the formatting properties from CSS3 (like rounded corners and drop shadows). Or, you can write your own JavaScript code that checks whether the current browser supports a feature you want to use (using a tool like Modernizr). If the browser fails the test, your code can show different content or use a less glamorous approach.
- **Use a JavaScript workaround.** Many of HTML5's new features are inspired by the stuff web developers are already doing the hard way. Thus, it should come as no surprise that you can duplicate many of HTML5's features using a good JavaScript library (or, in the worst-case scenario, by writing a whackload of your own custom JavaScript). Creating JavaScript workarounds can be a lot of work, but there are hundreds of good (and not-so-good) workarounds available free on the Web, which you can drop into your pages when needed. The more elaborate ones are called polyfills (page 35).

How to Find the Browser Requirements for Any HTML5 Feature

The people who have the final word on how much HTML5 you use are the browser vendors. If they don't support a feature, there's not much point in attempting to use it, no matter what the standard says. Today, there are four or five major browsers (not including the mobile variants that run on web-connected devices like smartphones and tablets). A single web developer has no chance of testing each prospective feature on every browser—not to mention evaluating support in older versions that are still widely used.

Fortunately, there's an ingenious website named "Can I use" that can help you out. It details the HTML5 support found in *every* mainstream browser. Best of all, it lets you focus on exactly the features you need. Here's how it works:

1. **Point your browser to *http://caniuse.com*.**

   The main page has a bunch of links grouped into categories, like CSS, HTML5, and so on.

2. **Choose the feature you want to study.**

   The quickest way to find a feature is to type its name into the Search box near the top of the page.

   Or, you can browse to the feature by clicking one of the links on the front page. The HTML5 group has a set of links that are considered part of the core HTML5 standard; the JS API group has links for JavaScript-powered features that began as part of HTML5 but have since been split off; the CSS group has links for the styling features that are part of CSS3; and so on.

TIP If you want, you can view the support tables for every feature in a group, all at once. Click the group title (like HTML5 or JS API), which is itself a link.

3. **Examine your results (Figure 1-4).**

   Each feature table shows a grid of different browser versions. The tables indicate support with the color of the cell, which can be red (no support), bright green (full support), olive green (partial support), or gray (undetermined, usually because this version of the browser is still under development and the feature hasn't been added yet).

4. **Optionally, choose different browsers to put under the microscope.**

   Ordinarily, the support table includes the most recent versions of the most popular browsers. However, you can tweak the table so it includes support information for other browsers that may be important to you—say, the aging IE 7 or a specialized mobile browser like Firefox for Android.

   To choose which browsers appear in the tables, start by clicking the "Show options" link above the table. A list of browsers appears, and you can choose the browsers you want by adding a checkmark next to their names. You can also tweak the "Versions shown" slider, which acts as a kind of popularity threshold—lower it to include older browser versions that are used less frequently.

   Alternatively, click the "Show all versions" link in the top-left corner of the table to see *all* the browser compatibility information that "Can I use" has in its database. But be warned that you'll get an immense table that stretches back to the dark days of Firefox 2 and IE 5.5.

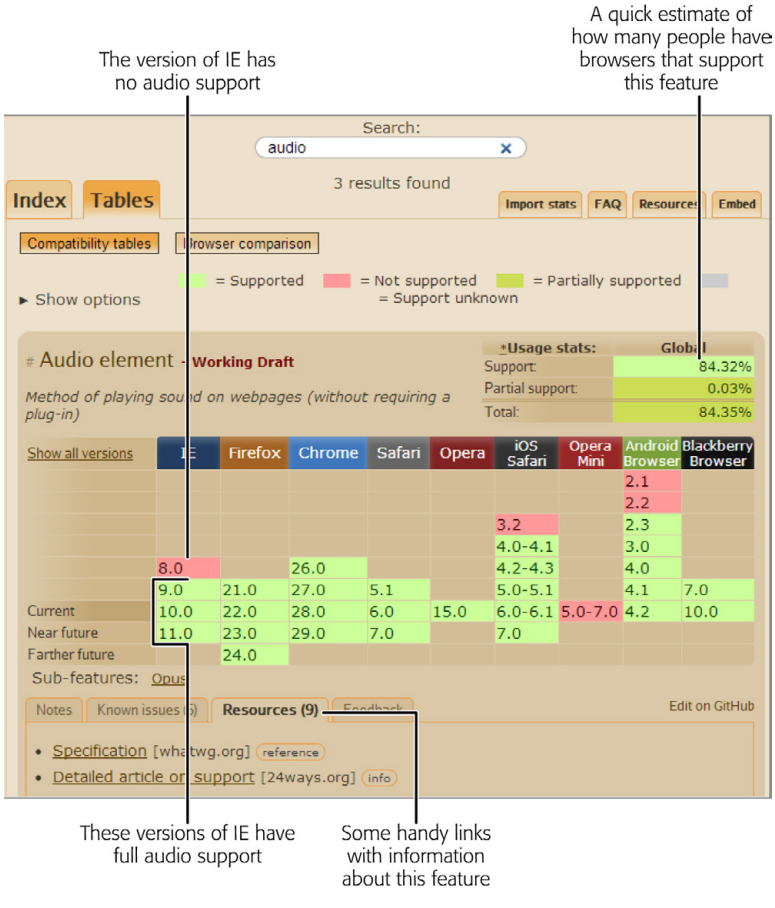


FIGURE 1-4

This search for audio *finds two tables. First is the table that describes browser support for the* `<audio>` *element (shown here).*

Green-shaded squares indicate browser versions that have full audio support, while red-shaded squares indicate browser versions that do not.

How to Find Out Which Browsers Are on the Web

How do you know *which* browser versions you need to worry about? Browser adoption statistics can tell you what portion of your audience has a browser that supports the features you plan to use. One good place to get an overall snapshot of all the browsers on the Web is GlobalStats, a popular tracking site. Here's how to use it:

1. **Browse to *http://gs.statcounter.com*.**

   On the GlobalStats site, you'll see a line graph showing the most popular browsers during the previous year. However, this chart doesn't include version information, so it doesn't tell you how many people are surfing with problematic versions of Internet Explorer (versions before IE 10). To get this information, you need to adjust another setting.

2. **Look for the Stat setting (under the chart) and choose "Browser Version (Partially Combined)."**

   This choice lets you consider not just which browsers are being used, but which *versions* of each browser. The partial combining tells GlobalStats to group together browsers that are rapidly updated, like Chrome and Firefox (Figure 1-5), so your chart isn't cluttered with dozens of extra lines.

3. **Optionally, change the geographic region in the Region box.**

   The standard setting is Worldwide, which shows browser statistics culled from across the globe. However, you can home in on a specific country (like Bolivia) or continent (like North America).

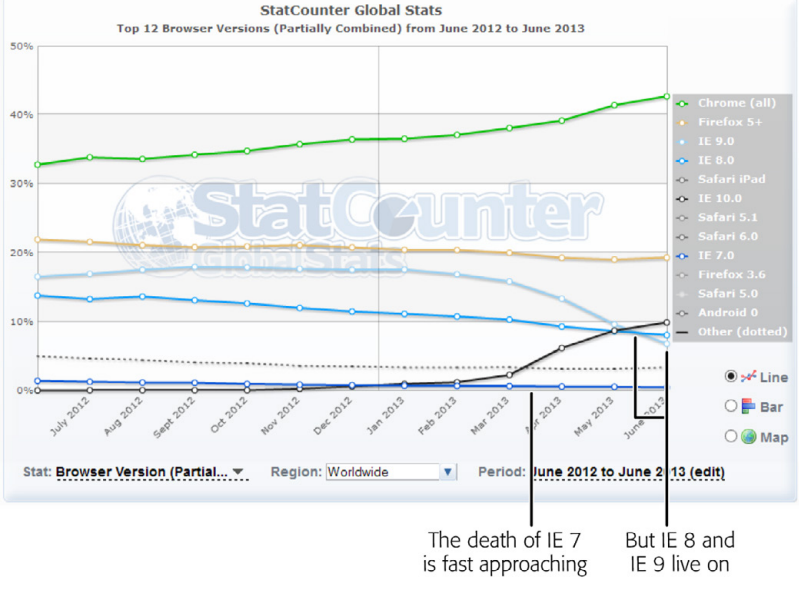


FIGURE 1-5

This chart shows that although Chrome's popularity is soaring, troublesome browser versions like IE 8 and IE 9 still cling to life.

4. **Optionally, click the text next to the Period setting to pick a different date range.**

   You'll usually see the browser usage trends for an entire year, but you can choose to focus on a smaller range, like the past three months.

5. **Optionally, change the chart type using the option buttons that are just to the right of the chart box.**

   Choose the Line option to see a line chart that shows the trend in browser adoption over time. Choose Bar to see a bar chart that shows a snapshot of the current situation. Or, choose Map to see a color-coded map that shows the countries where different browsers reign supreme.

GlobalStats compiles its statistics daily using tracking code that's present on millions of websites. And while that's a large number of pages and a huge amount of data, it's still just a small fraction of the total Web, which means you can't necessarily assume that your website visitors will use the same browsers.

Furthermore, browser-share results change depending on the web surfer's country and the type of website. For example, in Germany, Firefox is the top browser with over 40 percent of web surfers. And on the TechCrunch website (a popular news site for computer nerds), old versions of Internet Explorer are a rarity. So if you want to design a website that works for your peeps, it's worth reviewing the web statistics generated by your own pages. (And if you aren't already using a web tracking service for your site, check out the top-tier and completely free Google Analytics at *www.google.com/analytics*.)

Feature Detection with Modernizr

Feature detection is one strategy for dealing with features that aren't supported by all the browsers that hit up your site. The typical pattern is this: Your page loads and runs a snippet of JavaScript code to check whether a specific feature is available. You can then warn the user (the weakest option), fall back to a slightly less impressive version of your page (better), or implement a workaround that replicates the HTML5 feature you wanted to use (best).

Unfortunately, because HTML5 is, at its heart, a loose collection of related standards, there's no single HTML5 support test. Instead, you need dozens of different tests to check for dozens of different features—and sometimes even to check if a specific *part* of a feature is supported, which gets ugly fast.

Checking for support usually involves looking for a property on a programming object, or creating an object and trying to use it a certain way. But think twice before you write this sort of feature-testing code, because it's so easy to do it badly. For example, your feature-testing code might fail on certain browsers for some obscure reason or another, or quickly become out of date. Instead, consider using Modernizr (*http://modernizr.com*), a small, constantly updated tool that tests the support of a wide range of HTML5 and related features. It also has a cool trick for implementing fallback support when you're using new CSS3 features, which you'll see on page 180.

Here's how to use Modernizr in one of your web pages:

1. **Visit the Modernizr download page at *http://modernizr.com/download*.**

   Look for the "Development version" link, which points to the latest all-in-one JavaScript file for Modernizr.

2. **Right click the "Development version" link and choose "Save link as" or "Save target as."**

   Both commands are the same thing—the wording just depends on the browser you're using.

3. **Choose a place on your computer to save the file, and click Save.**

   The JavaScript file has the name *modernizr-latest.js*, unless you pick something different.

4. **When you're ready to use Modernizr, place that file in the same folder as your web page.**

   Or, place it in a subfolder and modify the path in the JavaScript reference accordingly.

5. **Add a reference to the JavaScript file in your web page's `<head>` section.**

   Here's an example of what your markup might look like, assuming the *modernizr-latest.js* file is in the same folder as your web page:

```
<head>
  <meta charset="utf-8">
  <title>HTML5 Feature Detection</title>
  <script src="modernizr-latest.js"></script>
  ...
</head>
```

   Now, when your page loads, the Modernizr script runs. It tests for a couple of dozen new features in mere milliseconds, and then creates a JavaScript object called `modernizr` that contains the results. You can test the properties of this object to check the browser's support for a specific feature.

TIP For the full list of features that Modernizr tests, and for the JavaScript code that you need to examine each one, refer to the documentation at *http://modernizr.com/docs*.

6. **Write some script code that tests for the feature you want and then carries out the appropriate action.**

   For example, here's how you might test whether Modernizr supports the HTML5 drag-and-drop feature, and show the result in the page:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>HTML5 Feature Detection</title>
  <script src="modernizr-latest.js"></script>
</head>

<body>
  <p>The verdict is... <span id="result"></span></p>

  <script>
// Find the element on the page (named result) where you can show
// the results.
var result = document.getElementById("result");
if (Modernizr.draganddrop) {
    result.innerHTML = "Rejoice! Your browser supports drag-and-drop.";
}
else {
    result.innerHTML = "Your feeble browser doesn't support drag-and-drop.";
}
  </script>
</body>

</html>
```

Figure 1-6 shows the result.

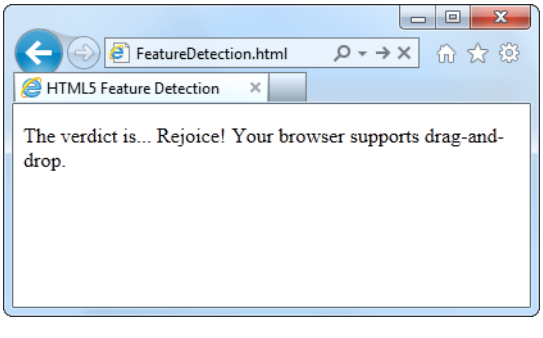


FIGURE 1-6

Although this example shows the right way to test for features, it shows a less-than-ideal approach for dealing with them. Instead of telling your website visitor about a missing feature, it's far, far better to implement some sort of workaround (even if it's not as neat or fully featured as the HTML5 equivalent) or to simply ignore the problem altogether (if the missing feature is a minor frill that's not necessary for the visitor to enjoy the page).

TIP This example uses basic and time-honored JavaScript techniques—looking up an element by ID and changing its content. If you find it a bit perplexing, you can brush up with the JavaScript review in Appendix B, "JavaScript: The Brains of Your Page."

The full Modernizr script is a bit bulky. It's intended for testing purposes while you're still working on your website. Once you've finished development and you're ready to go live, you can create a slimmed-down version of the Modernizr script that tests only for the features you need. To do so, go to the download page at *http://modernizr.com/download*. But this time, instead of using the "Development version" link, peruse the checkboxes below. Click the ones that correspond to the features you need to detect. Finally, click the Generate button to create your own custom Modernizr version, and then click the Download button to save it on your computer (Figure 1-7).

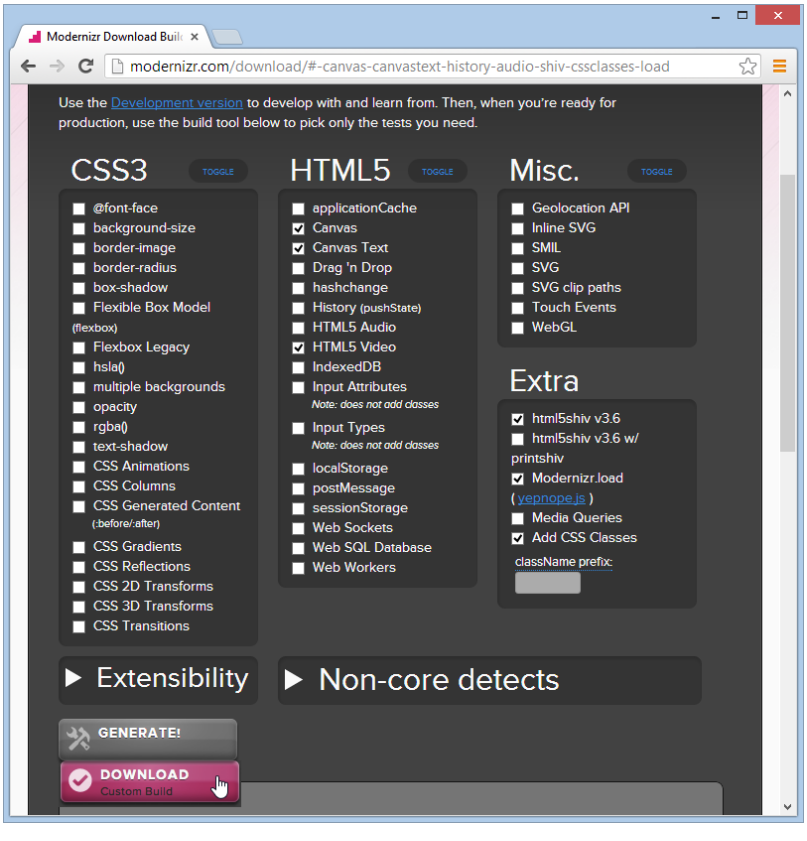


FIGURE 1-7

You're about to download a custom build of Modernizr that can detect support for the HTML5 canvas, the canvas text feature, and HTML5 video. This build of Modernizr won't be able to check for other features.

Feature "Filling" with Polyfills

Modernizr helps you spot the holes in browser support. It alerts you when a feature won't work. However, it doesn't do anything to patch these problems. That's where *polyfills* come in. Basically, polyfills are a hodgepodge collection of techniques for filling the gaps in HTML5 support on aging browsers. The word *polyfills* is borrowed from the product polyfiller, a compound that's used to fill in drywall holes before painting (also known as spackling paste). In HTML5, the ideal polyfill is one you can drop into a page without any extra work. It takes care of backward compatibility in a seamless, unobtrusive way, so you can work with pure HTML5 while someone else worries about the workarounds.

But polyfills aren't perfect. Some rely on other technologies that may be only partly supported. For example, one polyfill allows you to emulate the HTML5 canvas on old versions of Internet Explorer using the Silverlight plug-in. But if the web visitor isn't willing or able to install Silverlight, then you need to fall back on something else. Other polyfills may have fewer features than the real HTML5 feature, or poorer performance.

Occasionally, this book will point you to a potential polyfill. If you want more information, you can find the closest thing there is to a comprehensive catalog of HTML5 polyfills on GitHub at *http://tinyurl.com/polyfill*. But be warned—polyfills differ greatly in quality, performance, and support.

TIP Remember, it's not enough to simply know that a polyfill exists for a given HTML5 feature. You must test it and check how well it works on various old browsers *before* you risk incorporating the corresponding feature into your website.

With tools like browser statistics, feature detection, and polyfills, you're ready to think in depth about integrating HTML5 features into your own web pages. In the next chapter, you'll take the first step, with some HTML5 elements that can function in browsers both new and old.

CHAPTER

2

Structuring Pages with Semantic Elements

Over the two decades that the Web's been around, websites have changed dramatically. But the greatest surprise isn't how much the Web has changed, but how well ancient HTML elements have held up. In fact, web developers use the same set of elements to build today's modern sites that they used to build their predecessors 10 years ago.

One element in particular—the humble `<div>` (or *division*)—is the cornerstone of nearly every modern web page. Using `<div>` elements, you can carve an HTML document into headers, side panels, navigation bars, and more. Add a healthy pinch of CSS formatting, and you can turn these sections into bordered boxes or shaded columns, and place each one exactly where it belongs.

This `<div>`-and-style technique is straightforward, powerful, and flexible, but it's not *transparent*. When you look at someone else's markup, you have to put some effort into figuring out what each `<div>` represents and how the whole page fits together. To make sense of it all, you need to jump back and forth among the markup, the style sheet, and the displayed page in the browser. And you'll face this confusion every time you crack open anyone else's halfway-sophisticated page, even though you're probably using the same design techniques in your own websites.

This situation got people thinking. What if there was a way to replace `<div>` with something better? Something that worked like `<div>`, but conveyed a bit more meaning. Something that might help separate the sidebars from the headers and the ad bars from the menus. HTML5 fulfills this dream with a set of new elements for structuring pages.

TIP If your CSS skills are so rusty that you need a tetanus shot before you can crack open a style sheet, then you're not quite ready for this chapter. Fortunately, Appendix A, "Essential CSS," has a condensed introduction that covers the fundamentals.

Introducing the Semantic Elements

To improve the structure of your web pages, you need HTML5's *semantic elements*. These elements give extra meaning to the content they enclose. For example, the new `<time>` element flags a valid date or time in your web page. Here's an example of the `<time>` element at its very simplest:

```
Registration begins on <time>2014-11-25</time>.
```

And this is what someone sees when viewing the page:

> Registration begins on 2014-11-25.

The important thing to understand is that the `<time>` element doesn't have any built-in formatting. In fact, the web page reader has no way of knowing that there's an extra element wrapping the date. You can add your own formatting to the `<time>` element using a style sheet, but by default, the text inside a `<time>` element is indistinguishable from ordinary text.

The `<time>` element is designed to wrap a single piece of information. However, most of HTML5's semantic elements are designed to identify larger sections of content. For example, the `<nav>` element identifies a set of navigation links. The `<footer>` element wraps the footer that sits at the bottom of a page. And so on, for a dozen (or so) new elements.

NOTE Although semantic elements are the least showy of HTML5's new features, they're one of the largest. In fact, the majority of the new elements in HTML5 are semantic elements.

All the semantic elements share a distinguishing feature: They don't really do anything. By contrast, the `<video>` element, for example, embeds a fully capable video player in your page (page 147). So why bother using all these new elements that don't change the way your web page looks?

There are several good reasons:

- **Easier editing and maintenance.** It can be difficult to interpret the markup in a traditional HTML page. To understand the overall layout and the significance of various sections, you'll often need to scour a web page's style sheet. But by using HTML5's semantic elements, you provide extra structural information in the markup. That makes your life easier when you need to edit the page months later, and it's even more important if someone else needs to revise your work.

- **Accessibility.** One of the key themes of modern web design is making *accessible* pages—that is, pages that people can navigate using screen readers and other assistive tools. Accessibility tools that understand HTML5 can provide a far better browsing experience for disabled visitors. (For just one example, imagine how a screen reader can home in on the <nav> sections to quickly find the navigation links for a website.)

TIP To learn more about the best practices for web accessibility, you can visit the WAI (Web Accessibility Initiative) website at *www.w3.org/WAI*. Or, to get a quick look at what life is like behind a screen reader (and to learn why properly arranged headings are so important), check out the YouTube video at *http://tinyurl.com/6bu4pe*.

- **Search-engine optimization.** Search engines like Google use powerful *search bots*—automated programs that crawl the Web and fetch every page they can—to scan your content and index it in their search databases. The better Google understands your site, the better the chance that it can match a web searcher's query to your content, and the better the chance that your website will turn up in someone's search results. Search bots already check for some of HTML5's semantic elements to glean more information about the pages they're indexing.
- **Future features.** New browsers and web editing tools are sure to take advantage of semantic elements. For example, a browser could provide an outline that lets visitors jump to the appropriate section in a page. (In fact, Chrome already has a plug-in that does exactly that—see page 65.) Similarly, web design tools can include features that let you build or edit navigation menus by managing the content you've placed in the <nav> section.

The bottom line is this: If you can apply the semantic elements correctly, you can create cleaner, clearer pages that are ready for the next wave of browsers and web tools. But if your brain is still tied up with the old-fashioned practices of traditional HTML, the future may pass you by.

Retrofitting a Traditional HTML Page

The easiest way to introduce yourself to the new semantic elements—and to learn how to use them to structure a page—is to take a classic HTML document and inject it with some HTML5 goodness. Figure 2-1 shows the first example you'll tackle. It's a simple, standalone web page that holds an article, although other types of content (like a blog posting, a product description, or a short story) would work equally well.

TIP You can view or download the example in Figure 2-1 from the try-out site at *http://prosetech.com/html5*, along with all the examples for this chapter. Start with *ApocalypsePage_Original.html* if you'd like to try to remodel the HTML yourself, or *ApocalypsePage_Revised.html* if you want to jump straight to the HTML5-improved final product.

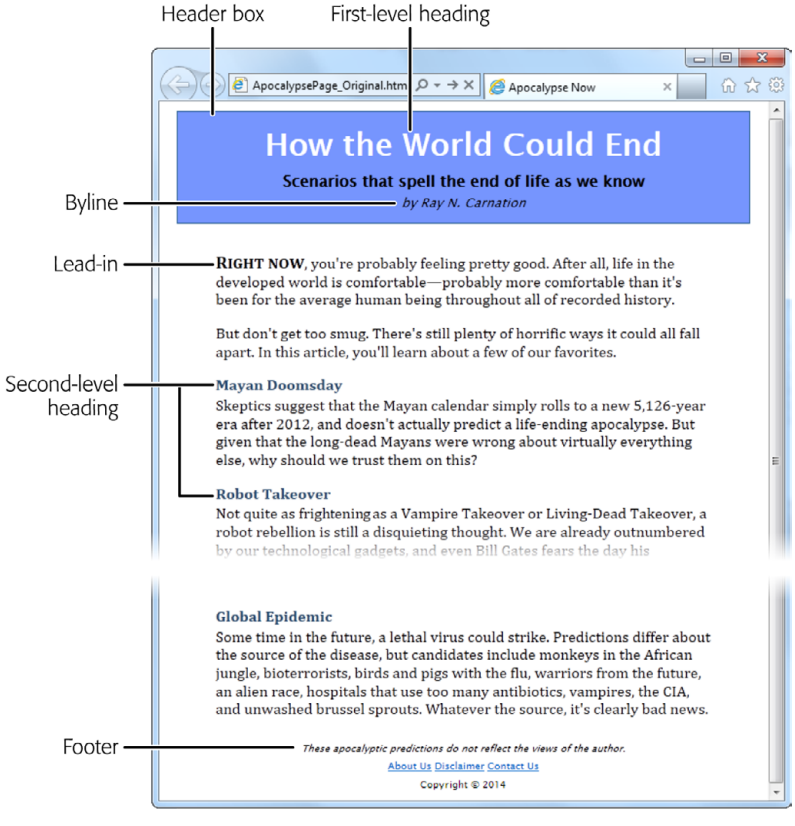


FIGURE 2-1

This ordinary HTML page has a basic, document-like structure. A linked style sheet provides all the formatting.

Page Structure the Old Way

There are a number of ways to format a page like the one shown in Figure 2-1. Happily, this example uses HTML best practices, which means the markup doesn't have a lick of formatting logic. There are no bold or italic elements, no inline styles, and certainly nothing as hideous as the obsolete `<font>` element. Instead, it's a neatly formatted document that's bound to an external style sheet.

Here's a shortened version of the markup, which highlights where the document plugs into its style sheet:

```
<div class="Header">
  <h1>How the World Could End</h1>
  <p class="Teaser">Scenarios that spell the end of life as we know it</p>
  <p class="Byline">by Ray N. Carnation</p>
</div>

<div class="Content">
  <p><span class="LeadIn">Right now</span>, you're probably ...</p>
  <p>...</p>

  <h2>Mayan Doomsday</h2>
  <p>Skeptics suggest ...</p>
  ...
</div>

<div class="Footer">
  <p class="Disclaimer">These apocalyptic predictions ...</p>
  <p>
    <a href="AboutUs.html">About Us</a>
    ...
  </p>
  <p>Copyright © 2014</p>
</div>
```

UP TO SPEED

What Are These Dots (...)?

This book can't show you the full markup for every example—at least not without expanding itself to 12,000 pages and wiping out an entire old-growth forest. But it *can* show you basic structure of a page and all its important elements. To do that, many of the examples in this book use an ellipsis (a series of three dots) to show where some content has been left out.

For example, consider the markup shown above on this page. It includes the full body of the page shown in Figure 2-2, but it leaves out the full text of most paragraphs, most of the article after the "Mayan Doomsday" heading, and the full list of links in the footer. But, as you know, you can pore over every detail by examining the sample files for this chapter on the try-out site (*http://prosetech.com/html5*).

In a well-written, traditional HTML page (like this one), most of the work is farmed out to the style sheet using the and containers. The lets you format snippets of text inside another element. The allows you to format entire sections of content, and it establishes the overall structure of the page (Figure 2-2).

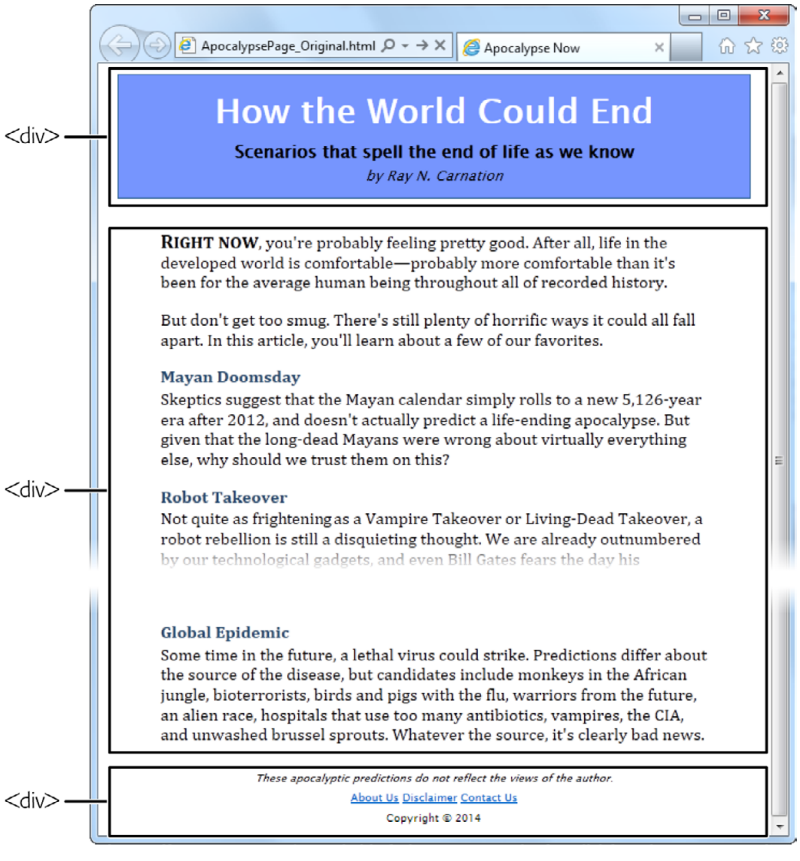


FIGURE 2-2

The elements carve this page into three logical sections: the header at the top, the content that follows, and the footer at the bottom.

Here, the style sheet formatting tasks are simple. The entire page is given a maximum width (800 pixels) to prevent really long text lines on widescreen monitors. The header is put in a bordered blue box. The content is padded on either side, and the footer is centered at the bottom of the page.

Thanks to the , formatting is easy. For example, the *ApocalypsePage_Original.css* style sheet uses the following rules to format the header box and the content inside:

```
/* Format the <div> that represents the header (as a blue, bordered box). */
.Header {
  background-color: #7695FE;
  border: thin #336699 solid;
  padding: 10px;
```

```
  margin: 10px;
  text-align: center;
}

/* Format any <h1> headings in the header <div> (that's the article title). */
.Header h1 {
  margin: 0px;
  color: white;
  font-size: xx-large;
}

/* Format the subtitle in the header <div>. */
.Header .Teaser {
  margin: 0px;
  font-weight: bold;
}

/* Format the byline in the header <div>. */
.Header .Byline {
  font-style: italic;
  font-size: small;
  margin: 0px;
}
```

You'll notice that this example makes good use of contextual selectors (page 441). For example, it uses the selector `.Header h1` to format all `<h1>` elements in the header box.

TIP This example is also described in the CSS review in Appendix A, "Essential CSS." If you'd like to take a detailed walk through the style sheet rules that format each section, flip to page 445.

Page Structure with HTML5

The `<div>` element is still a staple of web design. It's a straightforward, all-purpose container that you can use to apply formatting anywhere you want in a web page. The limitation of the `<div>` is that it doesn't provide any information about the page. When you (or a browser, or a design tool, or a screen reader, or a search bot) come across a `<div>`, you know that you've found a separate section of the page, but you don't know the purpose of that section.

To improve this situation in HTML5, you can replace some of your `<div>` elements with more descriptive semantic elements. The semantic elements behave exactly like `<div>` elements: They group a block of markup, they don't do anything on their own, and they give you a styling hook that lets you apply formatting. However, they also give your page a little more semantic smarts.

Here's a quick revision of the article shown in Figure 2-1. It removes two `<div>` elements and adds two semantic elements from HTML5: `<header>` and `<footer>`.

```
<header class="Header">
  <h1>How the World Could End</h1>
  <p class="Teaser">Scenarios that spell the end of life as we know it</p>
  <p class="Byline">by Ray N. Carnation</p>
</header>

<div class="Content">
  <p><span class="LeadIn">Right now</span>, you're probably ...</p>
  <p>...</p>

  <h2>Mayan Doomsday</h2>
  <p>Skeptics suggest ...</p>
  ...
</div>

<footer class="Footer">
  <p class="Disclaimer">These apocalyptic predictions ...</p>
  <p>
    <a href="AboutUs.html">About Us</a>
    ...
  </p>
  <p>Copyright © 2014</p>
</footer>
```

In this example, the <header> and <footer> elements take the place of the elements that were there before. Web developers who are revising a large website might start by wrapping the existing elements in the appropriate HTML5 semantic elements.

You'll also notice that the <header> and <footer> elements in this example still use the same class names. This way, you don't need to change the original style sheet. Thanks to the class names, the style sheet rules that used to format the elements now format the <header> and <footer> elements.

However, you might feel that the class names seem a bit redundant. If so, you can leave them out, like this:

```
<header>
  <h1>How the World Could End</h1>
  <p class="Teaser">Scenarios that spell the end of life as we know it</p>
  <p class="Byline">by Ray N. Carnation</p>
</header>
```

To make this work, you need to alter your style sheet rules so they apply themselves by element name. This works for the header and footer, because the current page has just a single <header> element and a single <footer> element.

Here's the revised style sheet that applies its formatting to the <header> element:

```
/* Format the <header> (as a blue, bordered box.) */
header {
  ...
}

/* Format any <h1> headings in the <header> (that's the article title). */
header h1 {
  ...
}

/* Format the subtitle in the <header>. */
header .Teaser {
  ...
}

/* Format the byline in the <header>. */
header .Byline {
  ...
}
```

Both approaches are equally valid. As with many design decisions in HTML5, there are plenty of discussions but no hard rules.

You'll notice that the section for the content remains. This is perfectly acceptable, as HTML5 web pages often contain a mix of semantic elements and the more generic containers. Because there's no HTML5 "content" element, an ordinary still makes sense.

NOTE Left to its own devices, this web page won't display correctly on versions of Internet Explorer before IE 9. To fix this issue, you need the simple workaround discussed on page 51. But first, check out a few more semantic elements that can enhance your pages.

Finally, there's one more element worth adding to this example. HTML5 includes an <article> element that represents a complete, self-contained piece of content, like a blog posting or a news story. The <article> element includes the whole shebang, including the title, author byline, and main content. Once you add the <article> element to the page, you get this structure:

```
<article>
  <header>
    <h1>How the World Could End</h1>
    ...
 </header>
```

```
    <div class="Content">
      <p><span class="LeadIn">Right now</span>, you're probably ...</p>
      <p>...</p>
      <h2>Mayan Doomsday</h2>
      <p>Skeptics suggest ...</p>
      ...
    </div>
  </article>

  <footer>
    <p class="Disclaimer">These apocalyptic predictions ...</p>
    ...
  </footer>
```

Figure 2-3 shows the final structure.

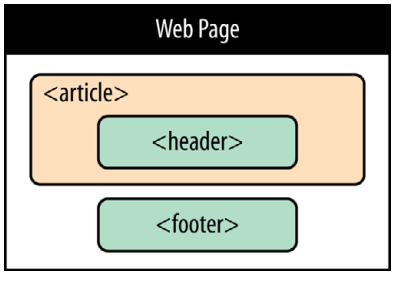


FIGURE 2-3

After the redesign, the page uses three of HTML5's semantic elements. If the old structure said, "Here is a page with three sections," then the new structure says, "Here is an article with a header, on a page with a footer."

Although the web page still looks the same in the browser, there's a fair bit of extra information lurking behind the scenes. For example, a search bot that stops by your site can quickly find your page's content (that's your article) and the title of that content (that's the header). It won't pay as much attention to the footer.

NOTE Sometimes articles are split over several web pages. The current consensus of webheads is that each part of the article should be wrapped in its own `<article>` element, even though it's not complete and self-contained. This messy compromise is just one of many that occur when semantics meet the practical, presentational considerations of the Web.

Adding a Figure with <figure>

Plenty of pages have images. But the concept of a *figure* is a bit different. The HTML5 specification suggests that you think of them much like figures in a book—in other words, a picture that's separate from the text, yet referred to in the text.

Generally, you let figures *float*, which means you put them in the nearest convenient spot alongside your text, rather than lock them in place next to a specific word or element. Often, figures have captions that float with them.

The following example shows some HTML markup that adds a figure to the apocalyptic article. It also includes the paragraph that immediately precedes the figure and the one that follows it, so you can see exactly where the figure is placed in the markup.

```
<p><span class="LeadIn">Right now</span>, you're probably ...</p>
<div class="FloatFigure">

  <img src="human_skull.jpg" alt="Human skull">
  <p>Will you be the last person standing if one of these apocalyptic
  scenarios plays out?</p>
</div>

<p>But don't get too smug ...</p>
```

This markup assumes that you've created a style sheet rule that positions the figure (and sets margins, controls the formatting of the caption text, and optionally draws a border around it). Here's an example:

```
/* Format the floating figure box. */
.FloatFigure {
  float: left;
  margin-left: 0px;
  margin-top: 0px;
  margin-right: 20px;
  margin-bottom: 0px;
}

/* Format the figure caption text. */
.FloatFigure p {
  max-width: 200px;
  font-size: small;
  font-style: italic;
  margin-bottom: 5px;
}
```

Figure 2-4 shows this example at work.

If you've created this sort of figure before, you'll be interested to know that HTML5 provides new semantic elements that are tailor-made for this pattern. Instead of using a boring to hold the figure box, you use a <figure> element. And if you have any caption text, you put that in a <figcaption> element inside the <figure>:

```
<figure class="FloatFigure">
   <img src="human_skull.jpg" alt="Human skull">
   <figcaption>Will you be the last person standing if one of these
   apocalyptic scenarios plays out?</figcaption>
</figure>
```

FIGURE 2-4

Now a figure graces the article. In the markup, it's defined just after the first paragraph, so it floats to the left of the following text. Notice that the width of the caption is limited, to create a nice, packed paragraph.

Of course it's still up to you to use a style sheet to position and format your figure box. In this example, you need to change the style rule selector that targets the caption text. Right now it uses `.FloatFigure p`, but the revised example requires `.FloatFigure figcaption`.

TIP In this example, the `<figure>` element still gets its formatting based on its class name (FloatFigure), not its element type. That's because you're likely to format figures in more than one way. For example, you might have figures that float on the left, figures that float on the right, ones that need different margin or caption settings, and so on. To preserve this sort of flexibility, it makes sense to format your figures with classes.

In the browser, the figure still looks the same. The difference is that the purpose of your figure markup is now perfectly clear. (Incidentally, `<figcaption>` isn't limited to holding text—you can use any HTML elements that make sense. Other possibilities include links and tiny icons.)

Finally, it's worth noting that in some cases the figure caption may include a complete description of the image, rendering the alt text redundant. In this situation, you can remove the `alt` attribute from the `<img>` element:

```
<figure class="FloatFigure">
   <img src="human_skull.jpg">
   <figcaption>A human skull lies on the sand</figcaption>
</figure>
```

Just make sure you don't set the alternate text with an empty string, because that means your image is purely presentational and screen readers should avoid it altogether.

Adding a Sidebar with <aside>

The new <aside> element represents content that is tangentially related to the text that surrounds it. For example, you can use an <aside> in the same way you use a sidebar in print—to introduce a related topic or to expand on a point that's raised in the main document. (See, for instance, the box at the bottom of page 50.) The <aside> element also makes sense if you need somewhere to stash a block of ads, some related content links, or even a pull-quote like the one shown in Figure 2-5.

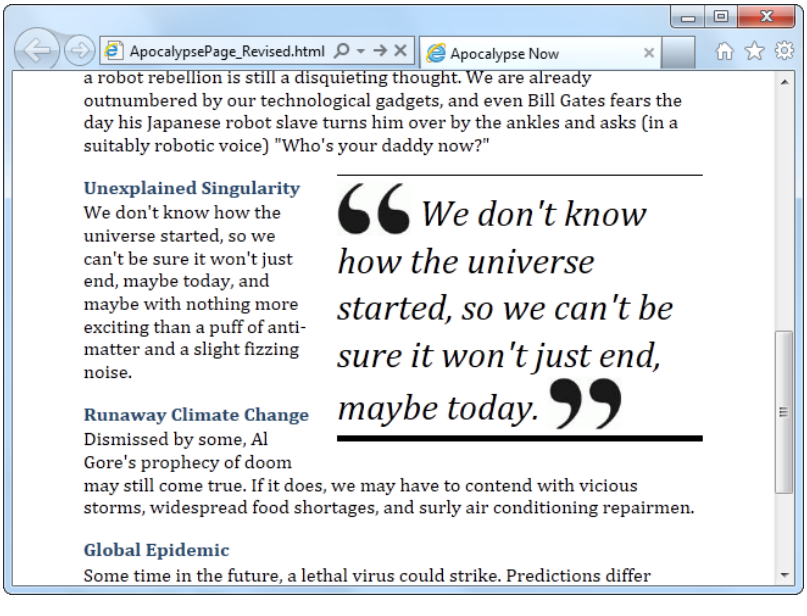


FIGURE 2-5
A pull-quote is a technique borrowed from print. It attracts the reader's attention and highlights important content.

You can easily create this effect with the well-worn element, but the <aside> element gives you a more meaningful way to mark up the same content:

```
<p>... (in a suitably robotic voice) "Who's your daddy now?"</p>

<aside class="PullQuote">
  <img src="quotes_start.png" alt="Quote">
```

```
    We don't know how the universe started, so we can't be sure it won't
    just end, maybe today.
    <img src="quotes_end.png" alt="End quote">
  </aside>

  <h2>Unexplained Singularity</h2>
```

This time, the style sheet rule floats the pull-quote to the right. Here are the styling details, just in case you're curious:

```
.PullQuote {
  float: right;
  max-width: 300px;
  border-top: thin black solid;
  border-bottom: thick black solid;
  font-size: 30px;
  line-height: 130%;
  font-style: italic;
  padding-top: 5px;
  padding-bottom: 5px;
  margin-left: 15px;
  margin-bottom: 10px;
}

.PullQuote img {
  vertical-align: bottom;
}
```

UP TO SPEED

How the Semantic Elements Were Chosen

Before inventing HTML5, its creators took a long look at the current crop of web pages. And they didn't just browse through their favorite sites; instead, they reviewed the Google-crunched statistics for over a *billion* web pages. (You can see the results of this remarkable survey at *http://tinyurl.com/state-of-the-web*.)

The Google survey analyzed the markup and compiled a list of the class names web authors were using in their pages. Their idea was that the class name might betray the purpose of the element and give a valuable clue about the way people were structuring pages. For example, if everyone has a `<div>` element that uses a class named header, then it's logical to assume everyone is putting headers at the tops of their web pages.

The first thing that Google found is that the vast majority of pages didn't use class names (or style sheets at all). Next, they compiled a short list of the most commonly used class names. Some of the most popular names were footer, header, title, menu, nav—which correspond well to HTML5's new semantic elements `<footer>`, `<header>`, and `<nav>`. A few others suggest possible semantic elements that haven't been created yet, like search and copyright.

In other words, the Web is awash with the same basic designs—for example, pages with headers, footers, sidebars, and navigation menus. But everyone has a slightly different way of doing more or less the same thing. From this realization, it's just a small leap to decide that the HTML language could be expanded with a few new elements that capture the semantics of what everyone is already doing. And this is exactly the insight that inspired HTML5's semantic elements.

Browser Compatibility for the Semantic Elements

So far, this exercise has been a lot of fun. But the best-structured page is useless if it won't work on older browsers.

Fortunately, HTML5's semantic elements are broadly supported on all modern browsers. It's almost impossible to find a version of Chrome, Firefox, Safari, or Opera that doesn't recognize them. The chief stumbling block is any version of Internet Explorer before IE 9, including the still-kicking IE 8.

Fortunately, this is one missing feature that's easy to patch up. After all, the semantic elements don't actually do anything. To support them, a browser simply needs to treat them like an ordinary `<div>` element. And to make that happen, all you need to do is fiddle with their styles, as described in the following sections. Do that, and you'll be rewarded with super-reliable semantic elements that work with any browser that's been released in the last 10 years.

NOTE If you're already using Modernizr (page 31), your pages are automatically immunized against semantic element issues, and you can safely skip the following discussion. But if you aren't using Modernizr, or if you're curious about how this fix works, read on.

Styling the Semantic Elements

When a browser meets an element it doesn't recognize, it treats it as an inline element. Most of HTML5's semantic elements (including all the ones you've seen in this chapter, except `<time>`) are *block* elements, which means the browser is supposed to render them on a separate line, with a little bit of space between them and the preceding and following elements.

Web browsers that don't recognize the HTML5 elements won't know to display some of them as block elements, so they're likely to end up in a clumped-together mess. To fix this problem, you simply need to add a new rule to your style sheet. Here's a super-rule that applies block display formatting to the nine HTML5 elements that need it in one step:

```
article, aside, figure, figcaption, footer, header, main, nav, section,
summary {
  display: block;
}
```

This style sheet rule won't have any effect for browsers that already recognize HTML5, because the `display` property is already set to `block`. And it won't affect any style rules you already use to format these elements. They will still apply *in addition* to this rule.

Using the HTML5 Shiv

That technique described in the previous section is enough to solve compatibility issues in most browsers, but "most" doesn't include Internet Explorer 8 and older. Old versions of IE introduce a second challenge: They refuse to apply style sheet formatting to elements they don't recognize. Fortunately, there is a workaround: You can trick IE into recognizing a foreign element by registering it with a JavaScript command. For example, here's a script block that gives IE the ability to recognize and style the <header> element:

```
<script>
  document.createElement('header')
</script>
```

Rather than write this sort of code yourself, you can find a ready-made script that does it for you. You simply need to add a reference to it in the <head> section of your page, like this:

```
<head>
  <title>...</title>
  <script src="http://html5shim.googlecode.com/svn/trunk/html5.js"></script>
  ...
<head>
```

This code grabs the script from the *html5shim.googlecode.com* web server and runs it before the browser starts processing the rest of the page. This script uses the JavaScript described above to create all the new HTML5 elements and goes one step further, by dynamically applying the styles described on page 51, to make sure the new elements display as proper block elements. The only remaining task is for you to use the elements and add your own style sheet rules to format them.

Incidentally, the *html5.js* script code is conditional—it runs only if it detects that you're running an old version of Internet Explorer. But if you want to avoid the overhead of requesting the JavaScript file at all, you can make the script reference conditional, like so:

```
<!--[if lt IE 9]>
  <script src="http://html5shim.googlecode.com/svn/trunk/html5.js"></script>
<![endif]-->
```

That way, other browsers (and IE 9 or later) will ignore this instruction, saving your page a few milliseconds of time.

TIP The previous example uses the HTML5 shiv straight from Google's code-hosting site. However, you can download your own copy from *http://tinyurl.com/the-shiv* and place it alongside your web pages. Just modify the script reference to point to the location where you upload the script file.

Finally, it's worth pointing out that if you test a web page on your own computer (rather than uploading it to a web server), Internet Explorer automatically places

the page in restricted mode. That means you'll see the infamous IE security bar at the top of the page, warning you that Internet Explorer has disabled all your scripts, including the HTML5 shiv. To run it, you need to explicitly click the security bar and choose to allow active content.

This problem disappears once you upload the page to a website, but it can be a hassle when testing your work. The solution is to add the "mark of the Web" comment to the beginning of your web page, as described on page 14.

Modernizr: An All-in-One Fix

There's one excellent alternate solution to the semantic styling problem: Use Modernizr (page 31). It has the HTML5 shiv built in, which means there's no need for you to fiddle with style rules or to include a reference to the *html5.js* script. So if you're already using Modernizr to test for feature support, consider the problem solved.

Designing a Site with the Semantic Elements

Adding the semantic elements to a simple, document-like page is easy. Adding them to a complete website isn't any harder, but it does raise a whole lot more questions. And because HTML5 is essentially virgin territory, there are a small number of settled conventions (but a large number of legitimate disagreements). That means when you have a choice between two markup approaches, and the HTML5 standard says they're both perfectly acceptable, it's up to you to decide which one makes the most sense for your content.

Figure 2-6 shows the more ambitious example that you'll consider next.

Deeper into Headers

There are two similar, but subtly different, ways to use the `<header>` element. First, you can use it to title some content. Second, you can use it to title your web page. Sometimes, these two tasks overlap (as with the single article example shown in Figure 2-1). But other times, you'll have a web page with both a page header and one or more pieces of headered content. Figure 2-6 is this sort of example.

What makes this situation a bit trickier is that the conventions for using the `<header>` element change based on its role. If you're dealing with content, you probably won't use a header unless you need it. And you need it only if you're creating a "fat" header. That is, one that includes the title and some other content—for example, a summary, the publication date, an author byline, an image, or subtopic links. Here's an example:

```
<header>
   <h1>How the World Could End</h1>
   <p class="Tagline">Scenarios that spell the end of life as we know it</p>
   <p class="Byline">by Ray N. Carnation</p>
</header>
```

FIGURE 2-6

Here, the single-page article you considered previously has been placed in a complete content-based website. A site header caps the page; the content is underneath; and a sidebar on the left provides navigation controls, "About Us" information, and an image ad.

However, when people create a header for a website, they almost always wrap it in a <header> element, even if there's nothing there but a title in a big CSS-formatted box. After all, it's a major design point of your website, and who knows when you might need to crack it open and add something new?

Here's the takeaway: Pages can have more than one <header> element (and they often will), even though these headers play different roles on the page.

The apocalyptic site (Figure 2-6) uses the <header> element for the website header and another <header> element for the article title. The <header> that caps the website holds a banner image, which combines graphical text and a picture:

```
<header class="SiteHeader">
  <img src="site_logo.png" alt="Apocalypse Today">
  <h1 style="display:none">Apocalypse Today</h1>
</header>
```

UP TO SPEED

Turning a Web Page into a Website

Figure 2-6 shows a single page from a fictional website.

In a real website, you'd have the same layout (and the same side panel) on *dozens* of different pages or more. The only thing that would change as the visitor clicks around the page is the main page content—in this case, the article.

HTML5 doesn't have any special magic for turning web pages into websites. Instead, you need to use the same tricks and technologies that web developers rely on in traditional HTML:

- **Server-side frameworks.** The idea is simple: When a browser requests a page, the web server assembles the pieces, including the common elements (like a navigation bar) and the content. This approach is by far the most common, and it's the only way to go on big, professional websites. Countless different technologies implement this approach in different ways, from web programming platforms like ASP.NET and PHP to content management systems like Drupal and WordPress.
- **Page templates.** Some high-powered web page editors (like Adobe Dreamweaver and Microsoft Visual Studio) include a page template feature. You begin by creating a template that defines the structure of your web pages and includes the repeating content you want to appear on every page (like the header and the sidebar). Then you use that template to create all your site pages. Here's the neat part: When you update the template, your web page editor automatically updates all the pages that use it.

Of course, you're free to use either technique, so this book focuses on the final result: the pasted-together markup that forms a complete page and is shown in the web browser.

Right away, you'll notice that this header adds a detail that you don't see on the page: an `<h1>` heading that duplicates the content that's in the picture. However, an inline style setting hides this heading.

This example raises a clear question: What's the point of adding a heading that you can't see? There are actually several reasons. First, all `<header>` elements require some level of heading inside, just to satisfy the rules of HTML5. Second, this design makes the page more accessible for people who are navigating it with screen readers, because they'll often jump from one heading to the next without listening to the content in between. And third, it establishes a heading structure that you can use in the rest of the page. That's a fancy way of saying that if you start with an `<h1>` for your website header, you may decide to use `<h2>` elements to title the other sections of the page (like "Articles" and "About Us" in the sidebar). For more about this design decision, see the box on page 56.

NOTE Of course, you could simplify your life by creating an ordinary text header. (And if you want fancy fonts, the CSS3 web font feature, described on page 206, can help you out.) But for the many web pages that put the title in a picture, the hidden heading trick is the next best solution.

FREQUENTLY ASKED QUESTION

The Heading Structure of a Site

Is it acceptable to have more than one level-1 heading on a page? Is it a good idea?

According to the official rules of HTML, you can have as many level-1 headings as you want. However, website creators often strive to have just a single level-1 heading per page, because it makes for a more accessible site—because people using screen readers might miss a level-1 heading as they skip from one level-2 heading to the next. There's also a school of webmaster thought that says every page should have exactly one level-1 heading, which is unique across the entire website and clearly tells search engines what content awaits.

The example in Figure 2-6 uses this style. The "Apocalypse Today" heading that tops the site is the only `<h1>` on the page. The other sections on the page, like "Articles" and "About Us" in the sidebar, use level-2 headings. The article title also uses a level-2 heading. (With a little bit of extra planning, you could vary the text of the level-1 heading to match the current article—after all, this heading isn't actually visible, and it could help the page match more focused queries in a search engine like Google.)

But there are other, equally valid approaches. For example, you could use level-1 headings to title each major section of your page, including the sidebar, the article, and so on.

Or, you could give the website a level-1 site heading and put level-2 headings in the sidebar (as in the current example) but make the article title into a second level-1 heading. This works fine in HTML5, because of its new outlining system. As you'll learn on page 65, some elements, including `<article>`, are treated as separate sections, with their own distinct outlines. So it makes perfect sense for these sections to start the heading hierarchy over again with a brand new `<h1>`. (However, HTML5 says it's fine to start with a different heading level, too.)

In short, there's no clear answer about how to structure your website. It seems likely that the "multiple `<h1>`" approach will become increasingly popular as HTML5 conquers the Web. But for now, many web developers are sticking with the "single `<h1>`" approach to keep screen readers happy.

Navigation Links with <nav>

The most interesting new feature in the apocalyptic website is the sidebar on the left, which provides the website's navigation, some extra information, and an image ad. (Typically, you'd throw in a block of JavaScript that fetches a randomly chosen ad using a service like Google AdSense. But this example just hard-codes a picture to stand in for that.)

In a traditional HTML website, you'd wrap the whole sidebar in a `<div>`. In HTML5, you almost always rely on two more specific elements: `<aside>` and `<nav>`.

The `<aside>` element is a bit like the `<header>` element in that it has a subtle, slightly stretchable meaning. You can use it to mark up a piece of related content, as you did with the pull-quote on page 49. Or, you can also use it to designate an entirely separate section of the page—one that's offset from the main flow.

The `<nav>` element wraps a block of links. These links may point to topics on the current page, or to other pages on the website. Most pages will have multiple `<nav>` sections in them. But not all links need a `<nav>` section—instead, it's generally reserved for the largest and most important navigational sections on a page. For

example, if you have a list of articles (as in Figure 2-6), you definitely need a <nav> section. But if you have just a couple of links at the bottom of the page with licensing and contact information, a full-blown <nav> isn't necessary.

With these two elements in mind, it's a good time to try a practice exercise. First, review the sidebar in Figure 2-6. Next, sketch out on a piece of paper how you would mark up the structure of this content. Then, read on to find out the best possible solution.

In fact, there are at least two reasonably good ways to structure this sidebar, as shown in Figure 2-7.

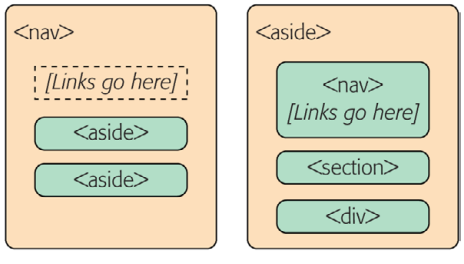


FIGURE 2-7

Left: You can think of the entire side panel as a navigation bar, with some other content wedged in. In this case, the whole panel can be a `<nav>`*, and the other content sections require an* `<aside>` *(because they aren't related to the sidebar's main content, the links).*

Right: Alternatively, consider the entire side panel to be a separate web page section that serves several purposes. In this case, the sidebar becomes an `<aside>` *while the navigational content inside is wrapped in a* `<nav>`*.*

The apocalyptic site uses the second approach (Figure 2-7, right). That's because the sidebar seems to serve several purposes, with none clearly being dominant. But if you have a lengthy and complex navigational section (like a collapsible menu) followed by a short bit of content, the first approach just might make more sense.

Here's the markup that shapes the sidebar, dividing it into three sections:

```
<aside class="NavSidebar">
  <nav>
    <h2>Articles</h2>
    <ul>
      <li><a href="...">How The World Could End</a></li>
      <li><a href="...">Would Aliens Enslave or Eradicate Us?</a></li>
      ...
    </ul>
  </nav>

  <section>
    <h2>About Us</h2>
    <p>Apocalypse Today is a world leader in conspiracy theories ..."
    </p>
  </section>
```

```
    <div>
      <img src="ad.jpg" alt="Luckies cigarette ad: it's toasted">
    </div>
  </aside>
```

Here are the key points:

- **The title sections ("Articles" and "About Us") are level-2 headings.** That way, they are clearly subordinate to the level-1 website heading, which makes the page more accessible to screen readers.
- **The links are marked up in an unordered list using the <ul> and <li> elements.** Website designers agree that a list is the best, most accessible way to deal with a series of links. However, you may need to use style sheet rules to remove the indent (as done here) and the bullets (not done in this example).
- **The "About Us" section is wrapped in a <section> element.** That's because there's no other semantic element that suits this content. A <section> is slightly more specific than a —it's suitable for any block of content that starts with a heading. If there were a more specific element to use (for example, a hypothetical <about> element), that would be preferable to a basic <section>, but there isn't.
- **The image ad is wrapped in a .** The <section> element is appropriate only for content that starts with a title, and the image section doesn't have a heading. (Although if it did—say, "A Word from Our Sponsors"—a <section> element would be the better choice.) Technically, it's not necessary to put any other element around the image, but the makes it easier to separate this section, style it, and throw in some JavaScript code that manipulates it, if necessary.

There are also some details that this sidebar *doesn't* have but many others do. For example, complex sidebars may start with a <header> and end with a <footer>. They may also include multiple <nav> sections—for example, one for a list of archived content, one with a list of late-breaking news, one with a blogroll or list of related sites, and so on. For an example, check out the sidebar of a typical blog, which is packed full of sections, many of which are navigational.

The style sheet rules you use to format the <aside> sidebar are the same as the ones you'd use to format a traditional sidebar. They place the sidebar in the correct spot, using absolute positioning, and set some formatting details, like padding and background color:

```
aside.NavSidebar
{
  position: absolute;
  top: 179px;
  left: 0px;
  padding: 5px 15px 0px 15px;
  width: 203px;
```

```
    min-height: 1500px;
    background-color:#eee;
    font-size: small;
  }
```

This rule is followed by contextual style sheet rules that format the <h2>, <ul>, <li>, and <img> elements in the sidebar. (As always, you can get the sample code from *http://prosetech.com/html5*, and peruse the complete style sheet.)

Now that you understand how the sidebar is put together, you'll understand how it fits into the layout of the entire page, as shown in Figure 2-8.

NOTE As you've learned, the <nav> is often found on its own, or in an <aside>. There's one more common place for it to crop up: in the <header> element that tops a web page.

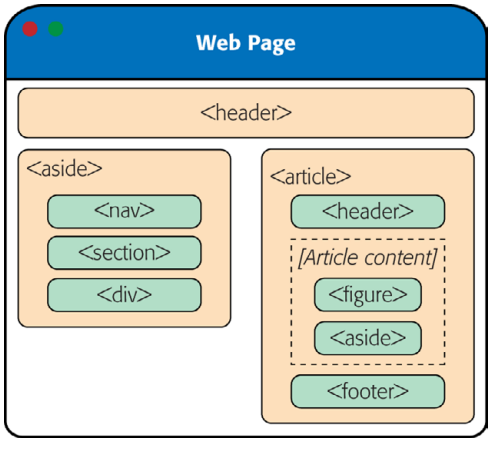


FIGURE 2-8
Here are all the semantic elements that you'll find in the apocalyptic web page shown in Figure 2-6.

Deeper into Sections

As you've already learned, the <section> is the semantic element of last resort. If you have a titled block of content, and the other semantic elements aren't appropriate, then the <section> element is generally a better choice than .

So what goes in a typical section? Depending on your perspective, the <section> element is either a flexible tool that fits many needs, or a loose and baggy monster with no clear identity. That's because sections play a variety of different web page roles. They can mark up any of the following:

- Small blocks of content that are displayed alongside the main page, like the "About Us" paragraph in the apocalyptic website.
- Self-contained content that can't really be described as an article, like a customer billing record or a product listing.

- Groups of content—for example, a collection of articles on a news site.
- A *portion* of a longer document. For example, in the apocalyptic article, you could mark up each separate end-of-the-world scenario as a distinct section. Sometimes you'll use sections in this way to ensure a correct outline for your document, as explained in the next section.

The last two items in the list are the most surprising. Many web developers find it's a bit of a stretch to use the same element to deal with a small fragment of an article and an entire group of articles. Some think that HTML5 should have at least two different elements to deal with these different scenarios. But the creators of HTML5 decided to keep things simple (by limiting the number of new elements) while making the new elements as flexible and practical as possible.

There's one last consideration. The `<section>` element also has an effect on a web page's outline, which is the concept you'll explore on page 65.

GEM IN THE ROUGH

Collapsible Boxes with <details> and <summary>

You've no doubt seen collapsible boxes on the Web—sections of content that you can show or hide by clicking a heading. Collapsible boxes are one of the easiest feats to pull off with basic JavaScript. You simply need to react when the heading is clicked, and then change a style setting to hide your box:

```
var box = document.
getElementById("myBox");
box.style.display = "none";
```

And then back again to make it reappear:

```
var box = document.
getElementById("myBox");
box.style.display = "block";
```

Interestingly, HTML5 adds two semantic elements that aim to make this behavior automatic. The idea is that you wrap your collapsible section in a `<details>` element and wrap the heading inside in a `<summary>` element. The final result is something like this:

```
<details>
  <summary>Section #1</summary>
  <p>If you can see this content, the
  section is expanded</p>
</details>
```

Browsers that support these elements (currently, that's just Chrome), will show just the heading, possibly with some sort of visual adornment (like a tiny triangle icon next to the heading). Then, if the user clicks the heading, the full content expands into view. Browsers that don't support the `<details>` and `<summary>` elements will show the full content right from the start, without giving the user any way to collapse it.

The `<details>` and `<summary>` elements are controversial. Many web developers feel that they aren't really semantic, because they're more about visual style than logical structure.

For now, it's best to avoid the `<details>` and `<summary>` elements because they have such poor browser support. Although you could write a workaround that uses JavaScript on browsers that don't support them, writing this workaround is more effort than just using a few lines of JavaScript to perform the collapsing on your own, on any browser.

Deeper into Footers

HTML5 and fat headers were meant for each other. Not only can you stuff in subtitles and bylines, but you can also add images, navigational sections (with the <nav> element), and virtually anything else that belongs at the top of your page.

Oddly, HTML5 isn't as accommodating when it comes to footers. The footer is supposed to be limited to a few basic details about the website's copyright, authorship, legal restrictions, and links. Footers aren't supposed to hold long lists of links, important pieces of content, or extraneous details like ads, social media buttons, and website widgets.

This raises a question: What should you do if your website design calls for a fat footer? After all, fat footers are wildly popular in website design right now (see Figure 2-9 for an example). They incorporate a number of fancy techniques, sometimes including the following:

- **Fixed positioning**, so the footer is always attached to the bottom of the browser window, no matter where the visitor scrolls (as with the example in Figure 2-9).
- **A close button**, so the visitor can close the footer and reclaim the space after reading the content (as with the example in Figure 2-9). To make this work, you use a simple piece of JavaScript that hides the element that wraps the footer (like the code shown in the box on page 60).

FIGURE 2-9

This absurdly fat footer is stuffed with garish extras, like an award picture and social media buttons. It uses fixed positioning to lock itself to the bottom of the browser window, like a toolbar. Fortunately, this footer has one redeeming quality: the close button in the top-right corner that lets anyone banish it from view.

- **A partially transparent background**, so you can see the page content *through* the footer. This setup works well if the footer is advertising breaking news or an important disclaimer, and it's usually used in conjunction with a close button.
- **Animation**, so the footer springs or slides into view. (For an example, see the related-article box that pops up when you reach the bottom of an article at *www.nytimes.com*.)

If your site includes this sort of footer, you have a choice. The simple approach is to disregard the rules. This approach is not as terrible as it sounds, because other website developers are sure to commit the same mistake, and over time the official rules may be loosened, allowing fancier footers. But if you want to be on the right side of the standard right now, you need to adjust your markup. Fortunately, it's not too difficult.

The trick is to split the standard footer details from the extras. In the browser, these can appear to be a single footer, but in the markup, they won't all belong to the `<footer>` element. For example, here's the structure of the fat footer in Figure 2-9:

```
<div id="FatFooter">
  <!-- Fat footer content goes here. -->
  <img onclick="CloseBox()" src="close_icon.png" class="CloseButton">
  ...
  <footer>
    <!-- Standard footer content goes here. -->
    <p>The views expressed on this site do not ... </p>
 </footer>
</div>
```

The outer `<div>` has no semantic meaning. Instead, it's a convenient package that bundles the extra "fat" content with the bare-bones footer details. It also lets you apply the style sheet formatting rule that locks the fat footer into place:

```
#FatFooter {
  position: fixed;
  bottom: 0px;
  height: 145px;
  width: 100%;
  background: #ECD672;
  border-top: thin solid black;
  font-size: small;
}
```

NOTE In this example, the style sheet rule applies its formatting by ID name (using the `#FatFooter` selector) rather than by class name (for example, using a `.FatFooter` selector). That's because the fat footer already needs a unique ID, so the JavaScript code can find it and hide it when someone clicks the close button. It makes more sense to use this unique ID in the style sheet than to add a class name for the same purpose.

You could also choose to put the footer in an <aside> element, to clearly indicate that the footer content is a separate section, and tangentially related to the rest of the content on the page. Here's what that structure looks like:

```
<div id="FatFooter">
 <aside>
    <!-- Fat footer content goes here. -->
    <img onclick="CloseBox()" src="close_icon.png" class="CloseButton">
    ...
 </aside>

  <footer>
    <!-- Standard footer content goes here. -->
    <p>The views expressed on this site do not ... </p>
  </footer>
</div>
```

The important detail here is that the <footer> is not placed inside the <aside> element. That's because the <footer> doesn't apply to the <aside> but to the entire website. Similarly, if you have a <footer> that applies to some piece of content, your <footer> needs to be placed inside the element that wraps that content.

NOTE The rules and guidelines for the proper use of HTML5's semantic elements are still evolving. Questions about the proper way to mark up large, complex sites stir ferocious debate in the HTML community. The best advice is this: If something doesn't seem true to your content, don't do it. Or you can discuss it online, where you can get feedback from dozens of super-smart HTML gurus. (One particularly good site is *http://html5doctor.com*, where you can see these ongoing debates unfolding in the comments section of most articles.)

Identifying the Main Content with <main>

HTML5 includes a sometimes-overlooked <main> element that identifies a web page's primary content. In the apocalypse site, for example, the main content is the entire article, not including the website header, sidebar, or footer. You should strongly consider using it on your own pages.

A properly applied <main> element wraps the <article> element precisely. Here's how it looks in the apocalypse page:

```
<!DOCTYPE html>
<html lang="en">
<head>
  ...
</head>

<body>
  <header>
      ...
  </header>
```

```
    <aside>
        ...
    </aside>

    <main>
      <article>
        ...
      </article>
    </main>
    <footer>
      ...
    </footer>
  </body>
  </html>
```

You can't put the <main> element inside the <article> element (or in any other semantic element). That's because the <main> element is meant to hold the page's full main content. It's *not* meant to indicate a portion of important content inside your document. For the same reason, unlike the other semantic elements, the <main> element can be used only *once* in a page.

At first glance, the <main> element doesn't seem terribly useful. However, it can be important for screen readers, because it lets them skip over extraneous material—like website headers, navigation menus, ads, sidebars, and so on—to get to the real content. And although the <main> element clings to the <article> element in this example, that's not necessarily the case in a more complex page. For example, if you created a page that lists multiple article summaries, each one wrapped in an <article> element, the <main> element would wrap the complete list of <article> elements, like this:

```
<main>
  <article>
    ...
  </article>

  <article>
    ...
  </article>

  <article>
    ...
  </article>

  ...
</main>
```

Here, the distinction is clear—each `<article>` represents a self-contained piece of content, but the main content of the page is the full set of articles.

It's appropriate to use the `<main>` element on any type of page, even if that page doesn't include an article. For example, if you build a game or an app, the main content is the bunch of markup that creates that game or app. You can use the `<main>` element to wrap the whole shebang, not including outside details like headers and footers.

NOTE The `<main>` element is a relative newcomer. It was introduced in the slightly tweaked version of the HTML5 standard called HTML 5.1 (page xv).

The HTML5 Outlining System

HTML5 defines a set of rules that dictate how you can create a *document outline* for any web page. A web page's outline could come in handy in a lot of ways. For example, a browser could let you jump from one part of an outline to another. A design tool could let you rearrange sections by dragging and dropping them in an outline view. A search engine could use the outline to build a better page preview, and screen readers could benefit the most of all, by using outlines to guide users with vision difficulties through a deeply nested hierarchy of sections and subsections.

However, none of these scenarios is real yet, because—except for the small set of developer tools you'll consider in the next section—almost no one uses HTML5 outlines today.

NOTE It's hard to get excited about a feature that doesn't affect the way the page is presented in a browser and isn't used by other tools. However, it's still a good idea to review the outline of your web pages (or at least the outline of a typical page from your website) to make sure that its structure makes sense and that you aren't breaking any HTML5 rules.

How to View an Outline

To understand outlines, you can simply take a look at the outlines your own pages produce. Right now, no browser implements the rules of HTML5 outlines (or gives you a way to peek at one). However, there are several tools that fill the gap:

- **Online HTML outliner.** Visit *http://gsnedders.html5.org/outliner* and tell the outliner which page you want to outline. As with the HTML5 validator you used in Chapter 1 (page 17), you can submit the page you want to outline in any of three ways: by uploading a file from your computer, by supplying a URL, or by pasting the markup into a text box.
- **Chrome extension.** You can use the h5o plug-in to analyze the outlines of pages when you view them in Chrome. Install it at *http://code.google.com/p/h5o* and then surf to an HTML5 page somewhere on the Web (sadly, h5o doesn't

work with files that are stored on your computer). An outline icon appears in the address bar, which reveals the structure of the page when clicked (Figure 2-10). The h5o page also provides a *bookmarklet* (a piece of JavaScript code that you can add to your web browser's bookmark list) which lets you display page outlines in Firefox and Internet Explorer, albeit with a few quirks.

- **Opera extension.** There's an Opera version of the h5o Chrome extension. Get it at *http://tinyurl.com/3k3ecdy*.

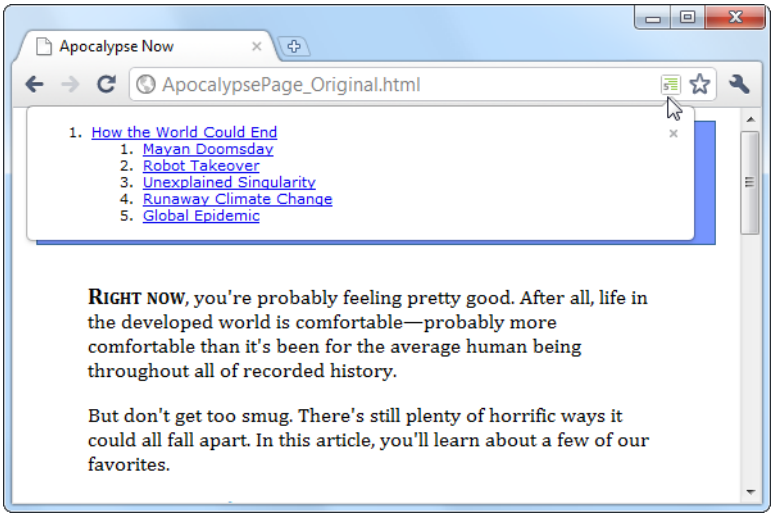


FIGURE 2-10

When you visit an HTML5 page with the Chrome h5o extension installed, an outline icon appears in the address bar. Click the icon to pop open a window with the full page outline.

Basic Outlines

To visualize the outline of your web page, imagine it stripped of all content except for the text in a numbered heading element (<h1>, <h2>, <h3>, and so on). Then, indent those headings based on their place in your markup, so more deeply nested headings are indented more in the outline.

For example, consider the apocalypse article in its initial, pre-HTML5 state:

```
<body>
  <div class="Header">
    <h1>How the World Could End</h1>
    ...
  </div>
  ...
  <h2>Mayan Doomsday</h2>
  ...
```

```
    <h2>Robot Takeover</h2>
    ...
    <h2>Unexplained Singularity</h2>
    ...
    <h2>Runaway Climate Change</h2>
    ...
    <h2>Global Epidemic</h2>
    ...
    <div class="Footer">
      ...
    </div>
  </body>
```

This simple structure leads to an outline like this:

1. How the World Could End
    1. Mayan Doomsday
    2. Robot Takeover
    3. Unexplained Singularity
    4. Runaway Climate Change
    5. Global Epidemic

Two levels of headings (`<h1>` and `<h2>`) create a two-level outline. This scheme is similar to the outline features in many word processing programs—for example, you can see much the same thing in Microsoft Word's Navigation pane.

On the other hand, markup like this:

```
<h1>Level-1 Heading</h1>
<h2>Level-2 Heading</h2>
<h2>Level-2 Heading</h2>
<h3>Level-3 Heading</h3>
<h2>Level-2 Heading</h2>
```

Gets an outline like this:

1. Level-1 Heading
    1. Level-2 Heading
    2. Level-2 Heading
        1. Level-3 Heading
    3. Level-2 Heading

Again, there are no surprises.

Finally, the outline algorithm is smart enough to ignore skipped levels. For example, if you write this slightly wobbly markup, which skips from <h1> directly to <h3>:

```
<h1>Level-1 Heading</h1>
<h2>Level-2 Heading</h2>
<h1>Level-1 Heading</h1>
<h3>Level-3 Heading</h3>
<h2>Level-2 Heading</h2>
```

You get this outline:

1. Level-1 Heading
    1. Level-2 Heading
2. **Level-1 Heading**
    1. **Level-3 Heading**
    2. Level-2 Heading

Now the level-3 heading has level-2 status in the outline, based on its position in the document. This might seem like one of those automatic error corrections browsers love to make, but it actually serves a practical purpose. In some situations, a web page may be assembled out of separate pieces—for example, it might contain a copy of an article that's published elsewhere. In this case, the heading levels of the embedded content might not line up perfectly with the rest of the web page. But because the outlining algorithm smooths these differences out, it's unlikely to be a problem.

Sectioning Elements

Sectioning elements are the ones that create a new, nested outline inside your page: <article>, <aside>, <nav>, and <section>. To understand how sectioning elements work, imagine a page that contains two <article> elements. Because <article> is a sectioning element, this page has (at least) three outlines—the outline of the overall page and one nested outline for each article.

To get a better grasp of this situation, consider the structure of the apocalypse article, after it's been revised with HTML5:

```
<body>
  <article>
    <header>
      <h1>How the World Could End</h1>
      ...
    </header>

    <div class="Content">
      ...
      <h2>Mayan Doomsday</h2>
      ...
```

```
      <h2>Robot Takeover</h2>
      ...
      <h2>Unexplained Singularity</h2>
      ...
      <h2>Runaway Climate Change</h2>
      ...
      <h2>Global Epidemic</h2>
      ...
    </div>
  </article>

  <footer>
    ...
  </footer>
</body>
```

Plug this into an outline viewer like *http://gsnedders.html5.org/outliner*, and you'll see this:

1. *Untitled Section*
    1. How the World Could End
        1. Mayan Doomsday
        2. Robot Takeover
        3. Unexplained Singularity
        4. Runaway Climate Change
        5. Global Epidemic

Here, the outline starts with an untitled section, which is the root `<body>` element. The `<article>` element starts a new, nested outline, which contains a single `<h1>` and several `<h2>` elements.

Sometimes, the "Untitled Section" note indicates a mistake. Although it's considered acceptable for `<aside>` and `<nav>` elements to exist without titles, the same leniency isn't usually given to `<article>` or `<section>` elements. In the previous example, the untitled section is the main section for the page, which belongs to the `<body>` element. Because the page contains a single article, there's no reason for the page to have a separate heading, and you can ignore this quirk.

Now consider what happens with a more complex example, like the apocalypse site with the navigation sidebar (page 54). Put that through an outliner, and you'll get this outline:

1. *Apocalypse Today*
    1. *Untitled Section*
        1. Articles
        2. About Us
    2. How the World Could End
        1. Mayan Doomsday
        2. Robot Takeover
        3. *Untitled Section*
        4. Unexplained Singularity
        5. Runaway Climate Change
        6. Global Epidemic

Here, there are two sectioning elements, and two nested outlines: one for the sidebar and one for the article. There are also two untitled sections, both of which are legitimate. The first is the `<aside>` element for the sidebar, and the second is the `<aside>` element that represents the pull-quote in the article.

NOTE In addition to sectioning elements, some elements are called *section roots*. These elements aren't just branches of an existing outline; they start a new outline of their own that doesn't appear in the main outline of the containing page. The `<body>` element that contains your web page content is a sectioning root, which makes sense. But HTML5 also considers the following elements to be sectioning roots: `<blockquote>`, `<td>`, `<fieldset>`, `<figure>`, and `<details>`.

Solving an Outline Problem

So far, you've looked at the examples in this chapter and seen the outlines they generated. And so far, the outlines have made perfect sense. But sometimes, a problem can occur. For example, imagine you create a document with this structure:

```
<body>
  <article>
    <h1>Natural Wonders to Visit Before You Die</h1>
    ...
    <h2>In North America</h2>
    ...
    <h3>The Grand Canyon</h3>
    ...
    <h3>Yellowstone National Park</h3>
    ...
    <h2>In the Rest of the World</h2>
    ...
    <aside>...</aside>
```

```
    ...
    <h3>Galapagos Islands</h3>
    ...
    <h3>The Swiss Alps</h3>
    ...
  </article>
</body>
```

GEM IN THE ROUGH

How Sectioning Elements Help Complex Pages

Sectioning is a great help with *syndication* and *aggregation*—two examples of the fine art of taking content from one web page and injecting it into another.

For example, imagine you have a web page that includes excerpts from several articles, all of which are drawn from other sites. Now imagine that this page has a deeply nested structure of headings, and somewhere inside—let's say under an <h4> heading—there's an article with content pulled from another web page.

In traditional HTML, you'd like the first heading in this content to use the <h5> element, because it's nested under an <h4>. But this article was originally developed to be placed somewhere else, on a different page, with less nesting, so it probably starts with an <h2> or an <h1>. The page would still work, but its hierarchy would be scrambled, and the page could be more difficult for screen readers, search engines, and other software to process.

In HTML5, this page isn't a problem. As long as you wrap the nested article in an `<article>` element, the extracted content becomes part of its own nested outline. That outline can start with any heading—it doesn't matter. What matters is its position in the containing document. So if the `<article>` element falls after an <h4>, then the first level of heading in that article behaves like a logical <h5>, the second level acts like a logical <h6>, and so on.

The conclusion is this: HTML5 has a logical outline system that makes it easier to combine documents. In this outline system, the position of your headings becomes more important, and the exact level of each heading becomes less significant—making it harder to shoot yourself in the foot.

You probably expect an outline like this:

1. *Untitled Section for the <body>*
    1. Natural Wonders to Visit Before You Die
        1. In North America
            1. The Grand Canyon
            2. Yellowstone National Park
        2. In the Rest of the World
        3. *Untitled Section for the <aside>*
            1. Galapagos Islands
            2. The Swiss Alps

But the outline you actually get is this:

1. *Untitled Section for the <body>*
    1. Natural Wonders to Visit Before You Die
        1. In North America
            1. The Grand Canyon
            2. Yellowstone National Park
        2. In the Rest of the World
        3. *Untitled Section for the <aside>*
        4. Galapagos Islands
        5. The Swiss Alps

Somehow, the addition of the `<aside>` after the `<h2>` throws off the following `<h3>` elements, making them have the same logical level as the `<h2>`. This clearly isn't what you want.

To solve this problem, you first need to understand that the HTML5 outline system automatically creates a new section every time it finds a numbered heading element (like `<h1>`, `<h2>`, `<h3>`, and so on), *unless* that element is already at the top of a section.

In this example, the outline system doesn't do anything to the initial `<h1>` element, because it's at the top of the `<article>` section. But the outline algorithm does create new sections for the `<h2>` and `<h3>` elements that follow. It's as though you wrote this markup:

```
<body>
  <article>
    <h1>Natural Wonders to Visit Before You Die</h1>
    ...
    <section>
      <h2>In North America</h2>
      ...
      <section>
        <h3>The Grand Canyon</h3>
        ...
      </section>
      <section>
        <h3>Yellowstone National Park</h3>
        ...
      </section>
    </section>
```

```
      <section>
        <h2>In the Rest of the World</h2>
        ...
      </section>
      <aside>...</aside>
      ...
      <section>
      <h3>Galapagos Islands</h3>
        ...
      </section>
      <section>
        <h3>The Swiss Alps</h3>
        ...
      </section>
    </article>
  </body>
```

Most of the time, these automatically created sections aren't a problem. In fact, they're usually an asset, because they make sure incorrectly numbered headings are still placed in the right outline level. The cost for this convenience is an occasional glitch, like the one shown here.

As you can see in this listing, everything goes right at first. The top <h1> is left alone (because it's in an <article> already), there's a subsection created for the first <h2>, then a subsection for each <h3> inside, and so on. The problem starts when the outline algorithm runs into the <aside> element. It sees this as a cue to close the current section, which means that when the sections are created for the following <h3> elements, they're at the same logical level as the <h2> elements before.

To correct this problem, you need to take control of the sections and subsections by defining some yourself. In this example, the goal is to prevent the second <h2> section from being closed too early, which you can do by defining it explicitly in the markup:

```
<body>
  <article>
    <h1>Natural Wonders to Visit Before You Die</h1>
    ...
    <h2>In North America</h2>
    ...
    <h3>The Grand Canyon</h3>
    ...
    <h3>Yellowstone National Park</h3>
    ...
    <section>
      <h2>In the Rest of the World</h2>
      ...
      <aside>...</aside>
      ...
```

```
        <h3>Galapagos Islands</h3>
        ...
        <h3>The Swiss Alps</h3>
        ...
      </section>
    </article>
  </body>
```

Now, the outline algorithm doesn't need to create an automatic section for the second <h2>, and so there's no risk of it closing the section when it stumbles across the <aside>. Although you could define the section for every heading in this document, there's no need to clutter your markup, as this single change fixes the problem.

NOTE Another solution is to replace the <aside> with a . The is not a sectioning element, so it won't cause a section to close unexpectedly.

Using the <aside> element doesn't always cause this problem. The earlier article examples used the <aside> element for a pull-quote but worked fine, because the <aside> fell between two <h2> elements. But if you carelessly plunk a sectioning element between two different heading levels, you should check your outline to make sure it still makes sense.

TIP If the whole outline concept seems overwhelmingly theoretical, don't worry. Truthfully, it's a subtle concept that many web developers will ignore (at least for now). The best approach is to think of the HTML5 outlining system as a quality assurance tool that can help you out. If you review your pages in an outline generator (like one of the tools listed on page 65), you can catch mistakes that may indicate other problems and make sure that you're using the semantic elements correctly.

CHAPTER

3

Writing More Meaningful Markup

In the previous chapter, you met HTML5's semantic elements. With their help, you can give your pages a clean, logical structure and prepare for a future of super-smart browsers, search engines, and assistive devices.

But you haven't reached the end of the semantic story yet. Semantics are all about adding *meaning* to your markup, and there are several types of information you can inject. In Chapter 2, semantics were all about *page structure*—you used them to explain the purpose of large blocks of content and entire sections of your layout. But semantics can also include *text-level information*, which you add to explain much smaller pieces of content. You can use text-level semantics to point out important types of information that would otherwise be lost in a sea of web page content, like names, addresses, event listings, products, recipes, restaurant reviews, and so on. Then this content can be extracted and used by a host of different services—everything from nifty browser plug-ins to specialized search engines.

In this chapter, you'll start by returning to the small set of semantic elements that are built into the HTML5 language. You'll learn about a few text-level semantic elements that you can use today, effortlessly. Next, you'll look at the companion standards that tackle text-level semantics head-on. That means digging into *microdata*, which began its life as part of the original HTML5 specification but now lives on as a separate, still-evolving standard managed by the W3C. Using microdata, you'll learn how to enrich your pages and juice up your web search listings.

The Semantic Elements Revisited

There's a reason you began your exploration into semantics with the page structure elements (see Table 3-1 for a recap). Quite simply, page structure is an easy challenge. That's because the vast majority of websites use a small set of common design elements (headers, footers, sidebars, and menus) to create layouts that are—for all their cosmetic differences—very similar.

TABLE 3-1 *Semantic elements for page structure*

ELEMENT	DESCRIPTION
`<article>`	Represents whatever you think of as an article—a section of self-contained content like a newspaper article, a forum post, or a blog entry (not including frills like comments or the author bio).
`<aside>`	Represents a complete chunk of content that's separate from the main page content. For example, it makes sense to use `<aside>` to create a sidebar with related content or links next to a main article.
`<figure>` and `<figcaption>`	Represents a figure. The `<figcaption>` element wraps the caption text, and the `<figure>` element wraps the `<figcaption>` and the `<img>` element for the picture itself. The goal is to indicate the association between an image and its caption.
`<footer>`	Represents the footer at the bottom of the page. This is a tiny chunk of content that may include small print, a copyright notice, and a brief set of links (for example, "About Us" or "Get Support").
`<header>`	Represents an enhanced heading that includes a standard HTML heading and extra content. The extra content might include a logo, a byline, or a set of navigation links for the content that follows.
`<nav>`	Represents a significant collection of links on a page. These links may point to topics on the current page or to other pages on the website. In fact, it's not unusual to have a page with multiple `<nav>` sections.
`<section>`	Represents a section of a document or a group of documents. The `<section>` is an all-purpose container with a single rule: The content it holds should begin with a heading. Use `<section>` only if the other semantic elements (for example, `<article>` and `<aside>`) don't apply.
`<main>`	Represents the main content of the page—all of it. For example, `<main>` might wrap an `<article>` element but leave out site-wide headers, footers, and sidebars. The `<main>` element is a new addition to the HTML 5.1 revision of HTML5 (page xv).

Text-level semantics are a tougher nut to crack. That's because people use a huge number of different types of content. If HTML5 set out to create an element for every sort of information you might add to a page, the language would be swimming in a mess of elements. Complicating the problem is the fact that structured information is also made of smaller pieces that can be assembled in different ways. For example, even an ordinary postal address would require a handful of elements (like `<address>`, `<name>`, `<street>`, `<postalcode>`, `<country>`, and so on) before anyone could use it in a page.

HTML5 takes a two-pronged approach. First, it adds a very small number of text-level semantic elements. But second, and more importantly, HTML5 supports a separate microdata standard, which gives people an extensible way to define any sort of information they want and then flag it in their pages. You'll cover both of these topics in this chapter. First up are three new text-level semantic elements: `<time>`, `<output>`, and `<mark>`.

Dates and Times with <time>

Date and time information appears frequently in web pages. For example, it turns up at the end of most blog postings. Unfortunately, there's no standardized way to tag dates, so there's no easy way for other programs (like search engines) to extract them without guessing. The `<time>` element solves this problem. It allows you to mark up a date, time, or combined date and time. Here's an example:

```
The party starts <time>2014-03-21</time>.
```

NOTE It may seem a little counterintuitive to have a `<time>` element wrapping a date (with no time), but that's just one of the quirks of HTML5. A more sensible element name would be `<datetime>`, but that isn't what they chose.

The `<time>` element performs two roles. First, it indicates where a date or time value is in your markup. Second, it provides that date or time value in a form that any software program can understand. The previous example meets the second requirement using the universal date format, which includes a four-digit year, a two-digit month, and a two-digit day, in that order, with each piece separated by a dash. In other words, the format follows this pattern:

```
YYYY-MM-DD
```

However, it's perfectly acceptable to present the date in a different way to the person reading your web page. In fact, you can use whatever text you want, as long as you supply the computer-readable universal date with the `datetime` attribute, like this:

```
The party starts <time datetime="2014-03-21">March 21<sup>st</sup></time>.
```

Which looks like this in the browser:

The party starts March 21st.

The <time> element has similar rules about times, which you supply in this format:

```
HH:MM
```

That's a two-digit hour (using a 24-hour clock), followed by a two-digit number of minutes, like this:

```
Parties start every night at <time datetime="16:30">4:30 p.m.</time>.
```

Finally, you can specify a time on a specific date by combining these two standards. Just put the date first, followed by a space, and then the time information.

```
The party starts <time datetime="2014-03-21 16:30">March 21<sup>st</sup>
at 4:30 p.m.</time>.
```

NOTE Originally, the <time> element required a slightly different format to combine date and time information. Instead of separating the two components with a space, you had to separate them with an uppercase *T* (for *time*), as in 2014-03-21T16:30. This format is still acceptable, so you may encounter it while perusing other people's web pages.

When combining dates and times, you may choose to tack a time zone offset on the end. For example, New York is in the Eastern time zone, which is known as UTC-5:00. (You can figure out your time zone at *http://en.wikipedia.org/wiki/Time_zone*.) To indicate 4:30 p.m. in New York, you'd use this markup:

```
The party starts <time datetime="2014-03-21 16:30-05:00">March 21<sup>st</sup>
at 4:30 p.m.</time>.
```

This way, the people reading your page get the time in the format they expect, while search bots and other bits of software get an unambiguous datetime value that they can process.

The <time> element also supports a pubdate attribute. You should use this if your date corresponds to the publication date of the current content (for example, the <article> in which the <time> is placed). Here's an example:

```
Published on <time datetime="2014-03-21" pubdate>March 31, 2014</time>.
```

NOTE Because the <time> element is purely informational and doesn't have any associated formatting, you can use it with any browser. There are no compatibility issues to worry about. But if you want to style the <time> element, you need the Internet Explorer workaround described on page 51.

JavaScript Calculations with <output>

HTML5 includes one semantic element that's designed to make certain types of JavaScript-powered pages a bit clearer—the <output> element. It's nothing more than a placeholder that your code can use to show a piece of calculated information.

For example, imagine you create a page like the one shown in Figure 3-1. This figure lets the user enter some information. A script then takes this information, performs a calculation, and displays the result just underneath.

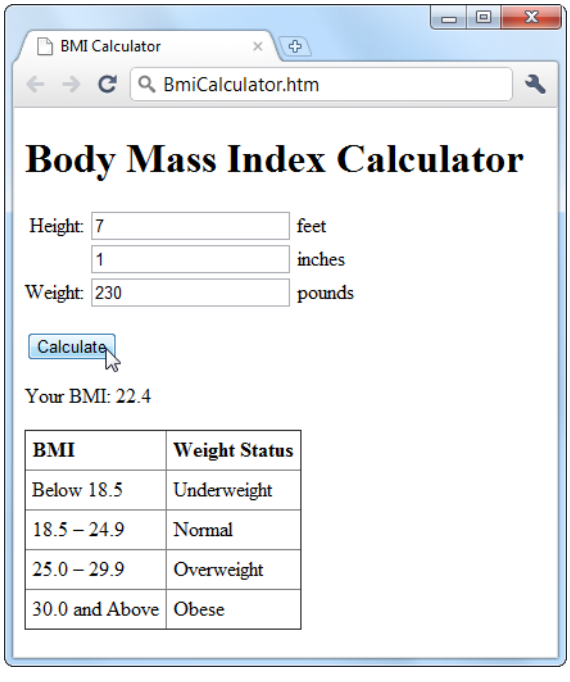


FIGURE 3-1

It's a time-honored web design pattern. Type some numbers, click a button, and let the page give you the answer.

The usual way of dealing with this is to assign a unique ID to the placeholder, so the JavaScript code can find it when it performs the calculation. Typically, web developers use the element, which works perfectly but doesn't provide any specific meaning:

```
<p>Your BMI: <span id="result"></span></p>
```

Here's the more meaningful version you'd use in HTML5:

```
<p>Your BMI: <output id="result"></output></p>
```

The actual JavaScript code doesn't need any changes, because it looks up the element by name and doesn't care about the element type:

```
var resultElement = document.getElementById("result");
```

NOTE Before you use <output>, make sure you've included the Internet Explorer workaround described on page 51. Otherwise, the element won't be accessible in JavaScript on old versions of Internet Explorer (IE 8 and earlier).

Often, this sort of page has its controls inside a <form> element. In this example, that's the three text boxes where people can type in information:

```
<form action="#" id="bmiCalculator">
  <label for="feet inches">Height:</label>
  <input name="feet"> feet<br>
  <input name="inches"> inches<br>

  <label for="pounds">Weight:</label>
  <input name="pounds"> pounds<br><br>
  ...
</form>
```

If you want to make your <output> element look even smarter, you can add the form attribute (which indicates the ID of the form that has the related controls) and the for attribute (which lists the IDs of the related controls, separated by spaces). Here's an example:

```
<p>Your BMI: <output id="result" form="bmiCalculator"for="feet inches pounds">
</output></p>
```

These attributes don't actually do anything, other than convey information about where your <output> element gets its goods. But they will earn you some serious semantic brownie points. And if other people need to edit your page, these attributes could help them sort out how it works.

TIP If you're a bit hazy about forms, you'll learn more in Chapter 4. If you know more about Esperanto than JavaScript, you can brush up on the programming language in Appendix B, "JavaScript: The Brains of Your Page." And if you want to try this page out for yourself, you can find the complete example at *http://prosetech.com/html5*.

Highlighted Text with <mark>

The <mark> element represents a section of text that's highlighted for reference. It's particularly appropriate when you're quoting someone else's text and you want to bring attention to something:

```
<p>In 2009, Facebook made a bold grab to own everyone's content,
<em>forever</em>. This is the text they put in their terms of service:</p>
<blockquote>You hereby grant Facebook an <mark>irrevocable, perpetual,
non-exclusive, transferable, fully paid, worldwide license</mark> (with the
right to sublicense) to <mark>use, copy, publish</mark>, stream, store,
retain, publicly perform or display, transmit, scan, reformat, modify, edit,
frame, translate, excerpt, adapt, create derivative works and distribute
(through multiple tiers), <mark>any user content you post</mark>
...
</blockquote>
```

The text in a <mark> element gets the yellow background shown in Figure 3-2.

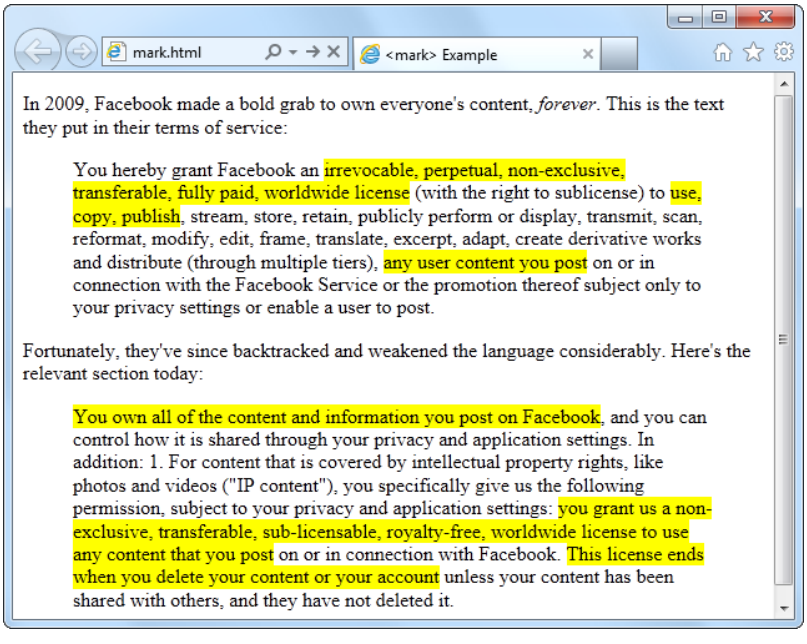


FIGURE 3-2
Here, the <mark> element highlights important details in a block of quoted text.

You can also use <mark> to flag important content or keywords, as search engines do when showing matching text in your search results, or to mark up document changes, in combination with <del> (for deleted text) and <ins> (for inserted text).

Truthfully, the <mark> element is a bit of a misfit. The HTML5 specification considers it to be a semantic element, but it plays a presentational role that's arguably more important. By default, marked-up text is highlighted with a bright yellow background (Figure 3-2), although you can apply your own style sheet rules to use a different formatting effect.

TIP The <mark> element isn't really about formatting. After all, there are lots of ways to make text stand out in a web page. Instead, you should use <mark> (coupled with any CSS formatting you like) when it's semantically appropriate. A good rule of thumb is to use <mark> to draw attention to ordinary text that has *become* important, either because of the discussion that frames it, or because of the task the user is performing.

Even if you stick with the default yellow-background formatting, you should add a style sheet fallback for browsers that don't support HTML5. Here's the sort of style rule you need:

```
mark {
  background-color: yellow;
  color: black;
}
```

You'll also need the Internet Explorer workaround described on page 51 to make the `<mark>` element style-able in old versions of IE.

Other Standards That Boost Semantics

At this point, it's probably occurring to you that there are a lot of potential semantic elements that HTML *doesn't* have. Sure, you can flag dates and highlighted text, but what about other common bits of information, like names, addresses, business listings, product descriptions, personal profiles, and so on? HTML5 deliberately doesn't wade into this arena, because its creators didn't want to bog the language down with dozens of specialized elements that would suit some people but leave others bored and unimpressed. To really get to the next level with semantics, you need to broaden your search beyond the core HTML5 language, and consider a few standards that can work with your web pages.

Semantically smart markup isn't a new idea. In fact, way back when HTML5 was still just a fantasy in WHATWG editor Ian Hickson's head, there were plenty of web developers clamoring for ways to make their markup more meaningful. Their goals weren't always the same—some wanted to boost accessibility, some were planning to do data mining, and others just wanted to dial up the cool factor on their resumés. But none of them could find what they wanted in the standard HTML language which is why several new specifications sprung up to fill the gap.

In the following sections, you'll learn about no fewer than *four* of these standards. First, you'll get the scoop on ARIA, a standard that's all about improving accessibility for screen readers. Then, you'll take a peek at three competing approaches for describing different types of content, whether it's contact details, addresses, business listings, or just about anything else you can fit between the tags of an HTML page.

ARIA (Accessible Rich Internet Applications)

ARIA is a developing standard that lets you supply extra information for screen readers through attributes on any HTML element. For example, ARIA introduces the `role` attribute, which indicates the purpose of a given element. For example, if you have a `<div>` that represents a header:

```
<div class="header">
```

You can announce that fact to screen readers by setting the ARIA `role` attribute to `banner`:

```
<div class="header" role="banner">
```

Of course, you learned last chapter that HTML5 also gives you a more meaningful way to mark up headers. So what you really should use is something like this:

```
<header role="banner">
```

This example demonstrates two important facts. First, ARIA requires you to use one of a short list of recommended role names. (For the full list, refer to the appropriate section of the specification at *http://tinyurl.com/roles-aria*.) Second, parts of ARIA overlap the new HTML5 semantic elements—which makes sense, because ARIA predates HTML5. But the overlap isn't complete. For example, some role names duplicate HTML5 (like `banner` and `article`), while others go further (like `toolbar` and `search`).

ARIA also adds two attributes that work with HTML forms. The `aria-required` attribute in a text box indicates that the user needs to enter a value. The `aria-invalid` attribute in a text box indicates that the current value isn't right. These attributes are helpful, because screen readers are likely to miss the visual cues that sighted users rely on, like an asterisk next to a missing field, or a flashing red error icon.

In order to apply ARIA properly, you need to learn the standard and spend some time reviewing your markup. Web developers are divided over whether it's a worthwhile investment, given that the standard is still developing and that HTML5 provides some of the same benefits with less trouble. However, if you want to create a truly accessible website today, you need to use both, because newer screen readers support ARIA but not yet HTML5.

NOTE For more information about ARIA (fully known as WAI-ARIA, because it was developed by the Web Accessibility Initiative group), you can read the specification at *www.w3.org/TR/wai-aria*.

RDFa (Resource Description Framework)

RDFa is a standard for embedding detailed metadata into your web documents using attributes. RDFa has a significant advantage: Unlike the other approaches discussed in this chapter, it's a stable, settled standard. RDFa also has two significant drawbacks. First, RDFa was originally designed for XHTML, not HTML5. It's a matter of debate how well the stricter, more elaborate RDFa syntax meshes with the more freewheeling philosophy of HTML5. Second, RDFa is complicated. Markup that's augmented with RDFa metadata is significantly longer and more cumbersome than ordinary HTML. And because of its complexity, RDFa is also more likely to contain errors—three times more likely, according to a recent Google web page survey.

RDFa isn't discussed in this chapter, although you will dig into its close HTML5 relative, microdata, on page 85. But if you prefer to learn more about RDFa, you can get a solid introduction on Wikipedia at *http://en.wikipedia.org/wiki/RDFa*, or you can visit the Google Rich Snippets page described later (page 94), which has RDFa versions of all its examples.

Microformats

Microformats are a simple, streamlined approach to putting metadata in your pages. Microformats don't attempt to be any sort of official standard. Instead, they're a loose collection of agreed-upon conventions that let pages share structured information without requiring the complexities of something like RDFa. This approach has given microformats tremendous success, and a recent web survey found that when a page has some sort of rich metadata, it's microformats 70 percent of the time.

Microformats work in an interesting way—they piggyback on the `class` attribute that's usually used for styling. You mark up your data using certain standardized style names, depending on the type of data. Then, another program can read your markup, extract the data, and check the attributes to figure out what everything means.

For example, you can use the hCard microformat to represent the contact details for a person, company, organization, or place. The first step is to add a root element that has the right class name. For hCard, the class name is `vcard`. (Usually, the class name matches the name of the microformat. The name `vcard` was chosen for historical reasons, because hCards are based on a much older format called Versitcard.)

Here's an example of a `<div>` that's ready to hold contact details using the hCard microformat:

```
<div class="vcard">
</div>
```

Inside this root element, you supply the contact information. Each detail must be wrapped in a separate element and marked up with the correct class name, as defined by the microformat you're using. For example, in an hCard you can use the `fn` class to flag a person's full name and the `url` class for that person's home page:

```
<div class="vcard">
  <h3 class="fn">Mike Rowe</h3>
  You can see Mike Rowe's website at
  <a class="url" href="http://www.magicsemantics.com">www.magicsemantics.com
  </a>
</div>
```

When you use class names for a microformat, you don't need to create matching styles in your style sheet. In the example above, that means that you don't need to write style rules for the `vcard`, `fn`, or `url` classes. Instead, the class names are put to a different use—advertising your data as a nicely structured, meaningful chunk of content.

NOTE Before you can mark up any data, you need to choose the microformat you want to use. There are only a few dozen microformats in widespread use, and most are still being tweaked and revised. You can see what's available and read detailed usage information about each microformat at *http://microformats.org/wiki*. To learn more about hCard, surf straight to *http://microformats.org/wiki/hCard*.

Once you've worked your way around hCard, you'll have no trouble understanding hCalendar, the world's second-most-popular microformat. Using hCalendar, you can mark up appointments, meetings, holidays, product releases, store openings, and so on. Just wrap the event listing in an element with the class name `vevent`. Inside, you need at least two pieces of information: the start date (marked up with the `dt-start` class) and a description (marked up with the `summary` class). You can also choose from a variety of optional attributes described at *http://microformats.org/wiki/hCalendar*, including an ending date or duration, a location, and a URL with more details. Here's an example:

```
<div class="vevent">
  <h2 class="summary">Web Developer Clam Bake</h2>
  <p>I'm hosting a party!</p>
  <p>It's
  <span class="dtstart" title="2014-10-25 13:30">Tuesday, October 25,
  1:30PM</span>
  at the <span class="location">Deep Sea Hotel, San Francisco, CA</span></p>
</div>
```

Based on the popularity of microformats, you might assume that the battle for the Semantic Web is settled. But not so fast—there are several caveats. First, the vast majority of pages have no rich semantic data at all. Second, most of the pages that have adopted microformats use them for just two purposes: contact information and event listings. So although microformats aren't going anywhere soon, there's still plenty of space for the competition. Third, the climate is beginning to shift to the more flexible but still lesser-known *microdata* specification. It seems increasingly likely that microformats were an interim stopping point on the way to the more sophisticated microdata standard, which is described in the next section.

Microdata

Microdata is a third take at solving the challenge of semantic markup. It began life as part of the HTML5 specification and later split into its own developing standard at *http://dev.w3.org/html5/md*. Microdata uses an approach that's similar to RDFa's, but simpler. Unlike microformats, microdata uses its own attributes and doesn't risk colliding with style sheet rules (or confusing the heck out of other web developers). This design means microdata is more logical, as well as easier to adapt for your own custom languages. But it also comes at the cost of brevity—microdata-enriched markup can bloat up a bit more than microformat-enriched markup.

Recently, microdata received a big boost when Microsoft, Google, Yahoo, and Yandex (Russia's largest search engine) teamed up to create a microdata-cataloguing site called *http://schema.org*. Here you'll find examples of all sorts of different microdata formats, including Person and Event (which echo the popular hCard and hEvent microformats) and more specialized types for marking up businesses, restaurants, reviews, products, books, movies, recipes, TV shows, bus stops, tourist attractions, medical conditions, medications, and more. Right now, only search engines pay any

attention to this information, but their traffic-driving, web-shaping clout is undeniable. (You'll see how search engines use this sort of information starting on page 94.)

NOTE It now seems possible that microdata just might catch on as the Goldilocks standard for metadata—a specification that's more flexible than microformats but not quite as complex as RDFa.

To begin a microdata section, you add the `itemscope` and `itemtype` attributes to any element (although a `<div>` makes a logical container, if you don't have one already). The `itemscope` attribute indicates that you're starting a new chunk of semantic content. The `itemtype` attribute indicates the specific type of data you're encoding:

```
<div itemscope itemtype="http://schema.org/Person">
</div>
```

To identify the data type, you use a predetermined, unique piece of text called an *XML namespace*. In this example, the XML namespace is *http://schema.org/Person*, which is a standardized microdata format for encoding contact details, as discussed in the box below.

UP TO SPEED

Understanding Microdata Namespaces

Every microdata format needs a namespace. Technically, the namespace identifies the *vocabulary* your microdata uses. For example, the namespace *http://schema.org/Person* indicates that this section of markup uses the Person vocabulary. You can go cross-eyed exploring dozens of microdata vocabularies at *http://schema.org* (see Figure 3-3).

XML namespaces are often URLs. Sometimes, you can even find a description of the corresponding data type by typing the URL into your web browser (as you can with the *http://schema.org/Person* data format). However, XML namespaces don't *need* to correlate to real web locations, and they don't need to be URLs at all. It just depends on what the developer chose when creating the format. The advantage of a URL is that it can incorporate a domain name belonging to a person or organization. This way, the namespace is more likely to be unique—no one else will create a different data format that shares the same namespace name and confuses everyone.

If a namespace begins with *http://schema.org*, it's an official vocabulary endorsed by the search engine dream team of Microsoft, Google, Yahoo, and Yandex. So if you use that vocabulary, you can be confident that the search engines of the world will understand what you're doing. If a namespace begins with *http://data-vocabulary.org*, it's using a slightly older set of microdata vocabularies. Most search engines will still understand your markup, but it's better to stick with the times and find an equivalent vocabulary at *http://schema.org*.

Once you have the container element, you're ready to move on to the next step. Inside your container element, you use the `itemprop` attribute to capture the important bits of information. The basic approach is the same as it was for microformats—you use a recognized `itemprop` name, and other pieces of software can grab the information from the associated elements.

Here's a microdata-fied version of the hCard microformat you saw earlier:

```
<div itemscope itemtype="http://schema.org/Person">
  <h3 itemprop="name">Mike Rowe</h3>
  You can see Mike Rowe's website at
  <a itemprop="url" href="http://www.magicsemantics.com">www.magicsemantics.
  com</a>
</div>
```

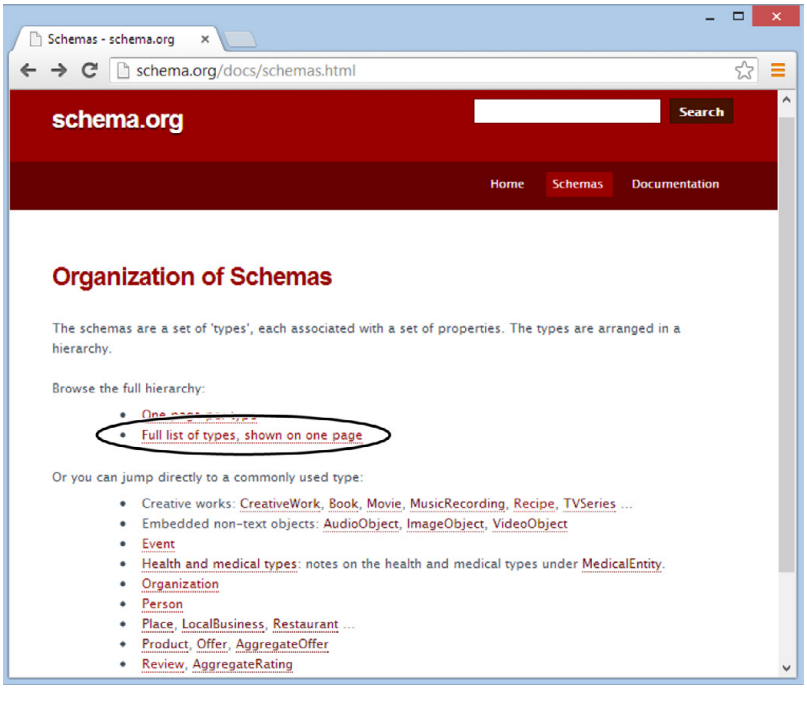


FIGURE 3-3

To find a microdata vocabulary that suits your information, there are few better starting points than the http://schema.org/docs/schemas.html page. Click a link to jump straight to the definition of a common vocabulary, like Person, Organization, or Event. Or, browse through the sprawling catalog by clicking the "Full list of types" link.

The most significant difference between microdata and microformats is that microdata uses the `itemprop` attribute to mark up elements instead of the `class` attribute.

NOTE Since microdata uses its own `itemscope`, `itemtype`, and `itemprop` attributes, rather than the `class` attribute, there's no chance you'll confuse your semantic markup with your style sheet formatting.

There are plenty of additional details you can mark up using the Person vocabulary. Common choices include postal and email address, telephone number, birth date, photo, job title, organization name, gender, nationality, and so on. For the full list of possible properties, refer to *http://schema.org/Person*.

NOTE The three standards for rich semantic data—RDFa, microdata, and microformats—all share broad similarities. They aren't quite compatible, but the markup is similar enough that the skills you pick up learning one system are mostly applicable to the others.

A Practical Example: Retrofitting an "About Me" Page

So far, you've learned about the basic structure of two semantic staples: microformats and microdata. Armed with this knowledge, you could look up a new microformat (from *http://microformats.org*) or microdata vocabulary (from *http://schema.org*) and start writing semantically rich markup.

However, life doesn't usually unfold this way—at least not for most web developers. Instead, you'll often need to take a web page that already has all the data it needs and retrofit the page with semantic data. This task is fairly easy if you keep a few points in mind:

- Often, you'll have important data mixed in with content that you want to ignore. In this case, you can add new elements around each piece of information you want to capture. Use a `<div>` if you want a block-level element or a `<span>` if you want to get a piece of inline content.
- Don't worry about the order of your information. As long as you use the right class names (for a microformat) or property names (for microdata), you can arrange your markup however you wish.
- If you're supplying a picture, you can use the `<img>` element. If you're supplying a link, you can use the `<a>` element. The rest of the time, you'll usually be marking up ordinary text.

Here's a typical example. Imagine you start with an "About Me" page (Figure 3-4) that has content like this:

```
<h1>About Me</h1>

<img src="face.jpg" alt="Mike's Face">
<p>This website is the work of <b>Mike Rowe Formatte</b>.
His friends know him as <b>The Big M</b>.</p>

<p>You can contact him where he works, at
The Magic Semantic Company (phone
641-545-0234 and ask for Mike) or email mike-f@magicsemantics.com.</p>

<p>Or, visit Mike on the job at:<br>
42 Jordan Gordon Street, 6th Floor<br>
San Francisco, CA 94105<br>
USA<br>
```

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

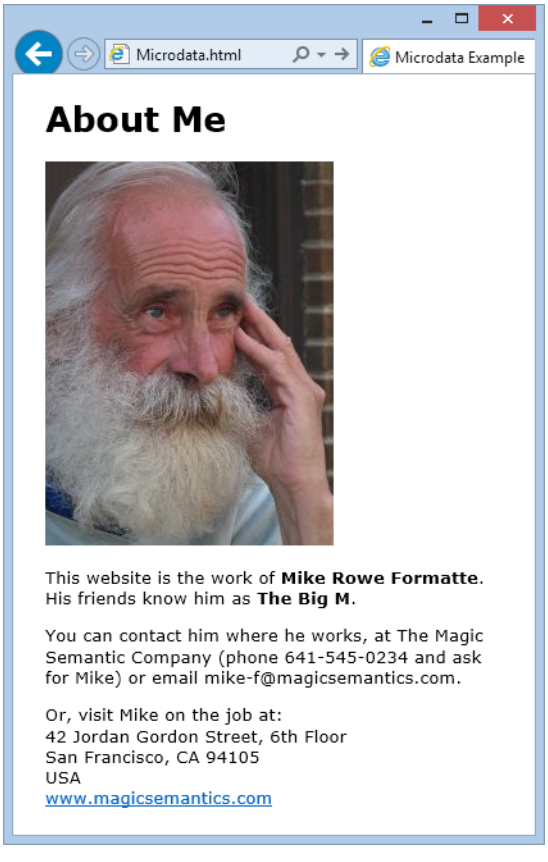


FIGURE 3-4

This "About Me" page includes the website author's contact details, before adding any microdata.

Clearly, this is a good fit for the familiar Person vocabulary (detailed at *http://schema.org/Person*). Here's a first attempt at weaving microdata around the key details in the "About Me" page. The newly inserted bits of microdata are emphasized with bold lettering:

```
<h1>About Me</h1>

<div itemscope itemtype="http://schema.org/Person">
  <img itemprop="image" src="face.jpg" alt="Mike's Face">
  <p>This website is the work of
  <span itemprop="jobTitle" style="display:none">Web Developer</span>
  <b itemprop="name">Mike Rowe Formatte</b>.
  His friends know him as <b itemprop="additionalname">The Big M</b>.</p>
```

```
  <p>You can contact him where he works, at
  The Magic Semantic Company</span> (phone
  <span itemprop="telephone">641-545-0234</span> and ask for Mike)
   or email <span itemprop="email">mike-f@magicsemantics.com</span>.</p>

  <p>Or, visit Mike on the job at:<br>
  42 Jordan Gordon Street, 6th Floor<br>
  San Francisco, CA 94105<br>
  USA<br>
  <a itemprop="url" href="http://www.magicsemantics.com">www.magicsemantics.
com</a>
  </p>
</div>
```

This example uses a few handy techniques:

- It adds new `<span>` elements to wrap the bits of content you need for the microdata.
- It adds the `itemprop` attribute to existing elements, where doing so makes sense. For example, the `<b>` element wraps the name information, so there's no need to add an additional `<span>`. (Of course, you *could* do so. For example, you might prefer to write something like `<b><span itemprop="name">Mike Rowe Formatte</span></b>`.
- It uses a hidden `<span>` to indicate the person's job title. (The text is hidden with an inline style rule that sets the `display` property to `none`, as you can see in the markup above.) This technique lets you hide redundant information, while still preserving it for search engines and other tools. That said, the content-hiding technique is a bit controversial, because some tools (like Google) ignore information that isn't made visible to the web page viewer.

It's common for microdata to have a *nested* structure that puts one microdata vocabulary inside another. For example, in the Person vocabulary you might have a set of address information nestled inside the personal details. Technically, the address information all belongs to a separate vocabulary, called PostalAddress.

To mark up the address information, you need to add a new `<div>` or `<span>` element that uses an `itemprop`, `itemscope`, or `itemtype` attribute. The `itemprop` attribute has the property name, the `itemscope` attribute indicates that you're starting a new vocabulary to supply the property data, and the `itemtype` property identifies the vocabulary by its XML namespace (in this case that's *http://schema.org/PostalAddress*). Here's how it all comes together:

```
<div itemscope itemtype="http://schema.org/Person">
  <img itemprop="image" src="face.jpg" alt="Mike's Face">
  <p>This website is the work of
  ...
```

```
    <p>Or, visit Mike on the job at:<br>
    <span itemprop="address" itemscope
     itemtype="http://schema.org/PostalAddress">
      ...
    </span>
  </div>
```

You can then fill in the address details inside the new section:

```
  <div itemscope itemtype="http://schema.org/Person">
    <img itemprop="image" src="face.jpg" alt="Mike's Face">
    <p>This website is the work of
    ...

    <p>Or, visit Mike on the job at:<br>
    <span itemprop="address" itemscope
     itemtype="http://schema.org/PostalAddress">
      <span itemprop="streetAddress">42 Jordan Gordon Street,
       6th Floor</span><br>
      <span itemprop="addressLocality">San Francisco</span>,
      <span itemprop="addressRegion">CA</span>
      <span itemprop="postalCode">94105</span><br>
      <span itemprop="addressCountry">USA</span><br>
    </span>
    ...
  </div>
```

This all makes perfect sense, but you might be wondering how you know *when* to define a new microdata section inside your first microdata section. Fortunately, the reference page on *http://schema.org* makes it clear (Figure 3-5).

A similar microdata-within-microdata trick takes place when you mark up the company name. Here, you need to set the person's `affiliation` property using the Organization vocabulary:

```
    <p>You can contact him where he works, at
    <span itemprop="affiliation" itemscope
     itemtype="http://schema.org/Organization">
      <span itemprop="name">The Magic Semantic Company</span>
    </span>
```

TIP If you don't fancy filling in all the itemtypes and itemprops yourself, there are online tools that you can use to generate properly formatted microdata-enriched markup. Two examples are *http://schema-creator.org* and *www.microdatagenerator.com*. With both sites, the idea is the same—you pick your vocabulary, type your data into the supplied text boxes, and then copy the finished markup.

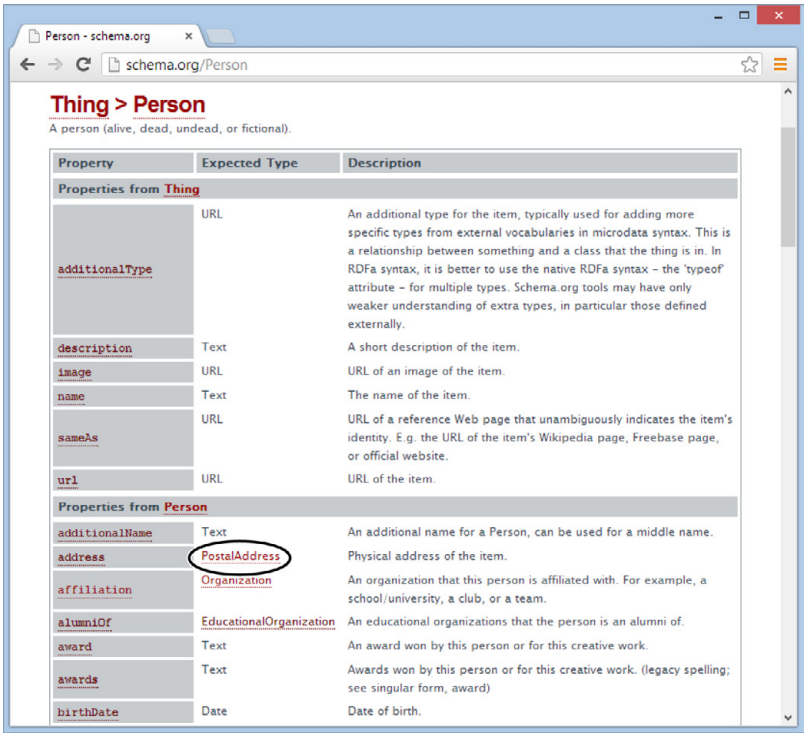


FIGURE 3-5

The Person vocabulary accepts a long list of properties (only some of which are shown here). Most properties take ordinary text, numbers, or date values. But some (like the `address` property, shown here) use their own vocabularies (like PostalAddress). To learn more about one of these subsections, just click the link.

Extracting Semantic Data in Your Browser

Now that you've gone to all this trouble, it's time to see what sort of benefits you can reap. Although no browser recognizes microformats on its own (at least at the time of this writing), there are a variety of plug-ins and scripts that can give browsers these capabilities. And it's not difficult to imagine useful scenarios. For example, a browser could detect the contact information on a page, list it in a side panel, and give you commands that would let you add a person to your address book as quickly as you bookmark a page. A similar trick could detect event information and let you add it to your calendar in a single click, or find locations and automatically plot them on a map.

Right now, no plug-in goes that far. However, some hardcore web developers have created a variety of JavaScript routines that can search for microformats or metadata, display it in pop-up boxes, or use it in another task. (One example is the JavaScript-powered Microdata Tool at *http://krofdrakula.github.io/microdata-tool*.) And some browsers have plug-ins that can spot different types of metadata on a web page (Figure 3-6).

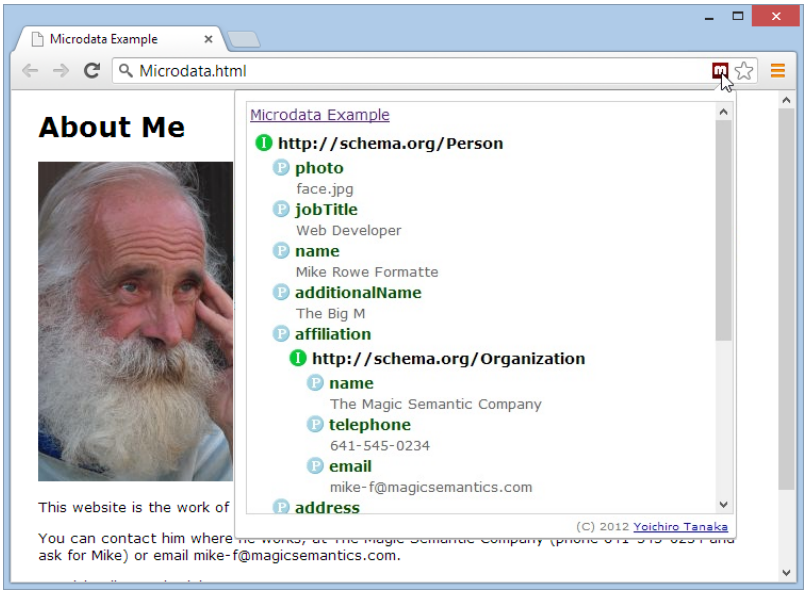


FIGURE 3-6

The Semantic Inspector is a Chrome extension that extracts all the microdata on the current page. Click its toolbar button, and you'll get the semantic skinny. Here, the Semantic Inspector examines the "About Me" page.

The best bet for a microdata future may be for mainstream browsers to incorporate direct support, just like Internet Explorer and Firefox do for feeds. A *feed* is a special sort of markup document that provides up-to-date content, like a list of recently published news articles. For example, if you visit a blog with Firefox, it automatically detects the blog's RSS feed and lets you create a "live" bookmark that fetches new content automatically. This is exactly the sort of value-added feature that could make microformats really useful.

How Search Engines Use Metadata

Stuffing your page with semantic details is a great way to win yourself some serious web-nerd cred. But even hardcore web developers need some sort of payoff to make the extra work (and the messier markup) worthwhile. It's nice to think about a world of super-smart, semantically aware browsers, but right now the cold, hard reality is that web surfers have little more than a few experimental and little-known browser plug-ins.

Fortunately, there is another reason to embrace rich semantics: *search engine optimization* (SEO). SEO is the art of making your website more visible in a search engine—in other words, making it turn up more often in a results page, helping it get a better ranking for certain keywords, and making it more likely to entice a

visitor to click through to your site. Good metadata can help with the last part of this equation. All you do is put the right semantic data on your page, and a search engine like Google will use it to present a fancier search listing, which can help your website stand out from the crowd.

Google Rich Snippets

Nowadays, most search engines can understand the metadata in the pages they catalog. In the rest of this chapter, you'll focus on what Google does with the metadata it finds. There are two reasons to go Google-centric. First, Google is the Earth's most popular search engine, with a commanding two-thirds share of worldwide web searches. Second, Google has been using and promoting metadata for years. The way it uses metadata today is the way other search engines will do so tomorrow.

Google uses the term *rich snippets* to lump together RDFa, microformats, and microdata. As you've already learned, these approaches share significant similarities and address the same problem. Google understands them all and attempts to treat them all equally, so it doesn't matter which approach you favor. (The following examples use microdata, with the aim of helping you get onboard with HTML5's newest semantic standard.)

To learn more about the metadata that Google supports, you can view Google's documentation at *http://tinyurl.com/GoogleRichSnippets*. Not only does it include a decent overview of RDFa, microformats, and microdata, it also shows many different snippet examples (like contact information, events, products, reviews, recipes, and so on). Best of all, Google includes an RDFa, microformat, and microdata version of each example, which can help you translate your semantic skills from one standard to another, if the need arises.

Enhanced Search Results

To see how Google's rich snippets feature works, you can use Google's Structured Data Testing Tool. This tool checks a page you supply, shows you the semantic data that Google can extract from the page, and then shows you how Google might use that information to customize the way it presents that page in someone's search results.

NOTE The Structured Data Testing Tool is useful for two reasons. First, it helps validate your semantic markup. If Google isn't able to extract all the information you put in the page, or if some of it is assigned to the wrong property, then you know you've done something wrong. Second, it shows you how the semantic data can change your page's appearance in Google's search results.

To use the Structured Data Testing Tool, follow these simple steps:

1. **Go to *www.google.com/webmasters/tools/richsnippets*.**

   This simple page includes a single text box (see Figure 3-7).

FIGURE 3-7

Here, Google found the person contact details and address information (from the microdata example shown on page 89). It used this information to add a gray byline under the page title with some of the personal details.

2. **If you want to paste in your markup, click the HTML tab.**

   There are two ways to use the Structured Data Testing Tool, represented by the two tabs on the page.

   - **The URL tab** asks Google to analyze a page that's already online. You simply put in its full web address.
   - **The HTML tab** lets you paste in the chunk of markup you want to analyze (the complete page isn't necessary) into a large text box. If you haven't yet uploaded your work, this is the most convenient approach.

3. **Type in your URL or paste in your markup. Then click Preview.**

   You can now review the results (see Figure 3-7). There are two important sections to review. The "Google search preview" section shows how the page may appear in a search result. The "Extracted rich snippet data from the page" shows all the raw semantic data that Google was able to pull out of your markup.

TIP If you see the dreaded "Insufficient data to generate the preview" error message, there are three possible causes. First, your markup may be faulty. Review the raw data that Google extracted, and make sure it found everything you put there. If you don't find a problem here, it's possible that you're trying to use a data type that Google doesn't yet support or you haven't included the bare minimum set of properties that Google needs. To figure out what the problem is, compare your markup with one of Google's examples at *http://tinyurl.com/GoogleRichSnippets*.

The method Google uses to emphasize contact details (Figure 3-7) is fairly restrained. However, contact details are only one of the rich data types that Google recognizes. Earlier in this chapter (page 85), you saw how to define events using microformats. Add a list of events to your page, and Google just might include them at the bottom of your search result, as shown in Figure 3-8.

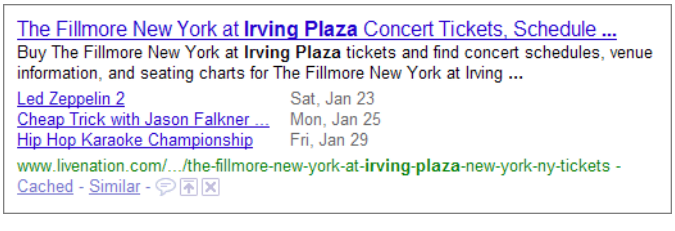


FIGURE 3-8

This example page has three events. If you supply a URL with your event listing (as done here), Google turns each event listing into a clickable link.

Google is also interested in business listings (which are treated in much the same way as personal contact details), recipes (which you'll take a peek at in the next section), and reviews (which you'll consider next).

The following example shows the markup you need to turn some review text into recognizable review microdata. The data standard is defined at *http://schema.org/Review*. Key properties include `itemReviewed` (in this case, a restaurant), `author` (the person making the review) and `reviewBody` (the full account of the review). You can also supply a one-sentence overview (`description`), the date when the review was made (`datePublished`, which supports HTML5's `<time>` element), and a score that's typically made on a scale from 0 to 5 (`reviewRating`).

Here's an example, with all the microdata details highlighted:

```
<div itemscope itemtype="http://schema.org/Review">
  <p itemprop="description">Pretty bad, and then the Health Department showed
    up.
  </p>
```

```
<div itemprop="itemReviewed" itemscope itemtype="http://schema.org/Thing">
  <span itemprop="name">Jan's Pizza House</span>
</div>

<div>
  <span itemprop="author" itemscope itemtype="http://schema.org/Person">
    Reviewed by: <span itemprop="name">Jared Elberadi</span>
  </span>
  on <time itemprop="datePublished" datetime="2014-06-26">January 26</time>
</div>

<div itemprop="reviewBody">I had an urge to mack on some pizza, and this
 place was the only joint around. It looked like a bit of a dive, but I went
 in hoping to find an undiscovered gem. Instead, I watched a Health
 Department inspector closing the place down. Verdict? I didn't get to
 finish my pizza, and the inspector recommends a Hep-C shot.</div>

<div itemprop="reviewRating" itemscope itemtype="http://schema.org/Rating">
  Rating:<span itemprop="ratingValue">1.5</span></div>
</div>
```

If you put this microdata-formatted review in a web page, Google gives it truly special treatment (Figure 3-9).

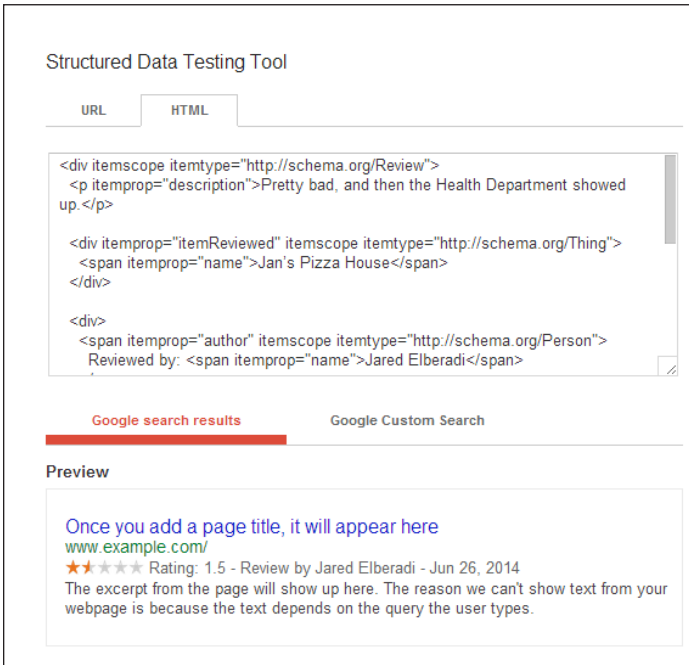


FIGURE 3-9

Reviews really stand out in search results. The ranking stars are eye-catching and attract immediate interest.

Keen eyes will notice that there are actually *four* microdata formats at work in this single review. There's one format for the review itself (*http://schema.org/Review*), one for the thing you're reviewing (*http://schema.org/Thing*), one for the person doing the reviewing (*http://schema.org/Person*), and one for the rating system (*http://schema.org/Rating*). You can use these standards to supply even more details with your review. For example, you can add the menu of the restaurant, the email address of the reviewer, or the minimum and maximum values of a custom rating system.

The *http://schema.org/Thing* data format, which is used by the `itemReviewed` property, is a bit different from the others. At first glance, a vocabulary for "things" sounds rather vague. This design is intentional—it gives you the flexibility to write a review about virtually anything, because Thing is a basic vocabulary upon which many more specialized categories are based. These include products, places, events, books, music recordings, and much more (see *http://schema.org/docs/full.html* for the full list of types). In this example, you could easily switch out the Thing vocabulary for the more specialized Restaurant vocabulary (defined at *http://schema.org/Restaurant*). However, as this page doesn't include any extra information about the restaurant, there's no need to take this step.

NOTE You can take a similar look at how Bing (Microsoft's search engine) deals with metadata using the Bing Markup Validator at *www.bing.com/toolbox/markup-validator*. And if you speak fluent Russian, why not take a look at Yandex's microformat validator? It's available at *http://webmaster.yandex.ru/microtest.xml*.

The Recipe Search Engine

Enhanced search listings are a neat trick, and they can drive new traffic into your website. But still, it's hard not to want something even more impressive to justify your newfound semantic skills. Happily, the geniuses at Google are busy dreaming up the future of search, and it has semantics all over it.

One brilliant idea is to use the semantic information not to tweak how an item is presented in a search, but to allow smarter search filtering. For example, if people marked up their résumés using RDFa, microformats, or microdata, Google could provide a specialized résumé searching feature that looks at this data, considering résumés from every popular career website and ignoring every other type of web content. This résumé search engine could also provide enhanced filtering options—for example, allowing companies to find candidates who have specific accreditations or have worked for specific companies.

TROUBLESHOOTING MOMENT

What to Do When Google Ignores Your Semantic Data

Just because Google *can* show a semantically enriched page in a certain special way doesn't mean it *will*. Google uses its own set of semi-secret rules to determine whether the semantic information is valuable to the searcher. But here are some surefire ways to make sure Google ignores your data:

- **The semantic data doesn't represent the main content.** In other words, if you slap your contact details on a page about fly-fishing, Google isn't likely to use your contact information. (After all, the odds are that when web searchers find this page, they're searching for something to do with fishing, and it doesn't make any sense to see a byline with your address and business underneath.) On the other hand, if you put your contact details on your résumé page, they're more likely to be used.
- **The semantic data is hidden.** Google won't use any content that's hidden via CSS.
- **Your website uses just a little semantic data.** If your site has relatively few pages that use semantic data, Google might inadvertently overlook the ones that do.

Avoid these mistakes and you stand a good chance of getting an enhanced listing.

Right now, Google doesn't have a resumé search engine. However, Google has experimented with job search technology for veterans (*http://tinyurl.com/vetjobsearch*) and product searches (*www.google.ca/merchants*). But one of its more mature metadata-powered search services is a tool called Recipe View that can hunt through millions of recipes.

By now, you can probably guess what recipe data looks like when it's marked up with microdata or a microformat. The entire recipe sits inside a container that uses the Recipe data format (that's *http://data-vocabulary.org/Recipe*). There are separate properties for the recipe name, the author, and a photo. You can also add a one-sentence summary and a ranking from user reviews.

Here's a portion of recipe markup:

```
<div itemscope itemtype="http://data-vocabulary.org/Recipe">
  <h1 itemprop="name">Elegant Tomato Soup</h1>
  <img itemprop="photo" src="soup.jpg" alt="A bowl of tomato soup">
  <p>By <span itemprop="author">Michael Chiarello</span></p>
  <p itemprop="summary">Roasted tomatoes are the key to developing the rich
  flavor of this tomato soup.</p>
  ...
```

After this, you can include key details about the recipe, including its prep time, cook time, and yield. You can also add a nested section for nutritional information (with details about serving size, calories, fat, and so on):

```
  ...
  <p>Prep time: <time itemprop="prepTime" datetime="PT30M">30 min</time></p>
  <p>Cook time: <time itemprop="cookTime" datetime="PT1H">40 min</time></p>
  <p>Yield: <span itemprop="yield">4 servings</span></p>
```

```
<div itemprop="nutrition" itemscope
  itemtype="http://data-vocabulary.org/Nutrition">
  Serving size: <span itemprop="servingSize">1 large bowl</span>
  Calories per serving: <span itemprop="calories">250</span>
  Fat per serving: <span itemprop="fat">3g</span>
</div>
...
```

NOTE The `prepTime` and `cookTime` properties are meant to represent a *duration* of time, not a single instant in time, and so they can't use the same format as the HTML5 `<time>` element. Instead, they use an ISO format that's detailed at *http://tinyurl.com/ISOdurations*.

After this is the recipe's ingredient list. Each ingredient is a separate nested section, which typically includes information like the ingredient name and quantity:

```
...
<ul>
  <li itemprop="ingredient" itemscope
   itemtype="http://data-vocabulary.org/RecipeIngredient">
   <span itemprop="amount">1</span>
   <span itemprop="name">yellow onion</span> (diced)
  </li>
  <li itemprop="ingredient" itemscope
   itemtype="http://data-vocabulary.org/RecipeIngredient">
   <span itemprop="amount">14-ounce can</span>
   <span itemprop="name">diced tomatoes</span>
  </li>
  ...
</ul>
...
```

Writing this part of the markup is tedious. But don't stop yet—the payoff is just ahead.

Finally, the directions are a series of paragraphs or a list of steps. They're wrapped up in a single property, like this:

```
...
<div itemprop="instructions">
  <ol>
    <li>Preheat oven to 450 degrees F.</li>
    <li>Strain the chopped canned tomatoes, reserving the juices.</li>
    ...
 </div>
 ...
</div>
```

For a full recipe example, see *http://tinyurl.com/RichSnippetsRecipe*.

NOTE Recipes tend to be long and fairly detailed, so marking them up is a long and involved project. This is a clear case where a good authoring tool could make a dramatic difference. Ideally, this tool would let web authors enter the recipe details in the text boxes of a nicely arranged window. It would then generate semantically correct markup that you could place in your web page.

Once Google indexes your marked-up recipe page, it will make that recipe available through the Recipe View search feature. Here's how to try out Recipe View:

1. **Surf to *www.google.com/landing/recipes*.**

   You arrive at the Recipe View feature homepage. It includes plenty of information about how Recipe View works, including a video that shows a recipe search in action.

2. **Click the "Try Google with Recipe View" button.**

   This button takes you to the familiar Google search page. However, there's something subtly different. Under the search box, the Recipes tab is highlighted in red, which indicates that you're performing a recipe search.

3. **Type a recipe name in the search box and click the search button.**

   Google starts you out with a search for *chicken pasta*, but you can do better.

4. **Click the "Search tools" button (which appears under the right side of the search box).**

   Because Google can *understand* the structure of every recipe, it can include smarter filtering options. When you click "Search tools," Google calls up three recipe-specific filtering features, which appear in drop-down lists, just above the search results (see Figure 3-10).

   - **Ingredients** lets you choose to see only the recipes that include or omit certain ingredients. You choose by clicking a tiny Yes or No checkbox next to the corresponding ingredient. (To create the ingredient list, Google grabs the most commonly used ingredients from the search results for your search.)
   - **Any cook time** lets you pinpoint fast recipes—for example, ones that take less than an hour or less than 10 minutes (raw chicken alert).
   - **Any calories** lets you filter out recipes that come in under a specific calorie-per-portion threshold, which is handy for dieters.

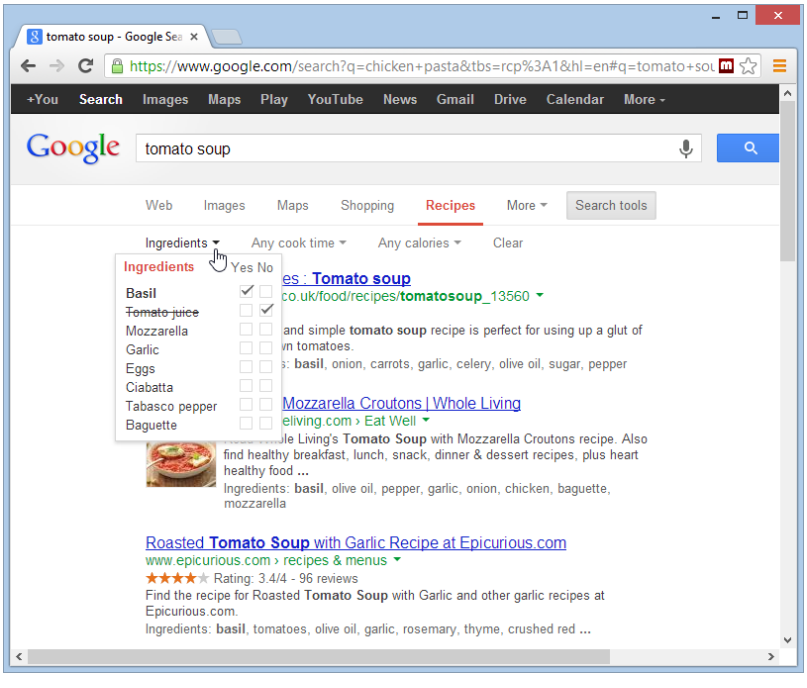


FIGURE 3-10

After you perform a recipe search, Google lets you filter the results based on some of the metadata that it found in the matching recipes. Here, a search hunts for a tomato soup recipe that's heavy on the basil but avoids tomato juice.

The semantic data you've learned about in this chapter gives web surfers a powerful information-hunting tool—and a more effective way to find your web pages.