"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

Free Sampler

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: 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

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.

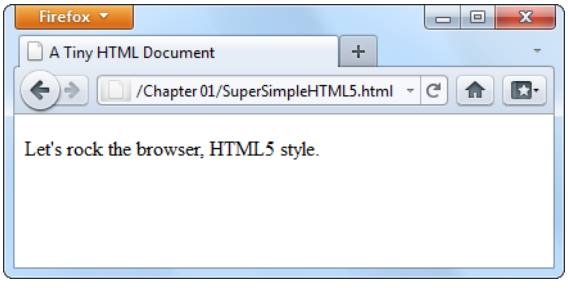


FIGURE 1-1
This super-simple HTML5 document holds a single line of text.

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.

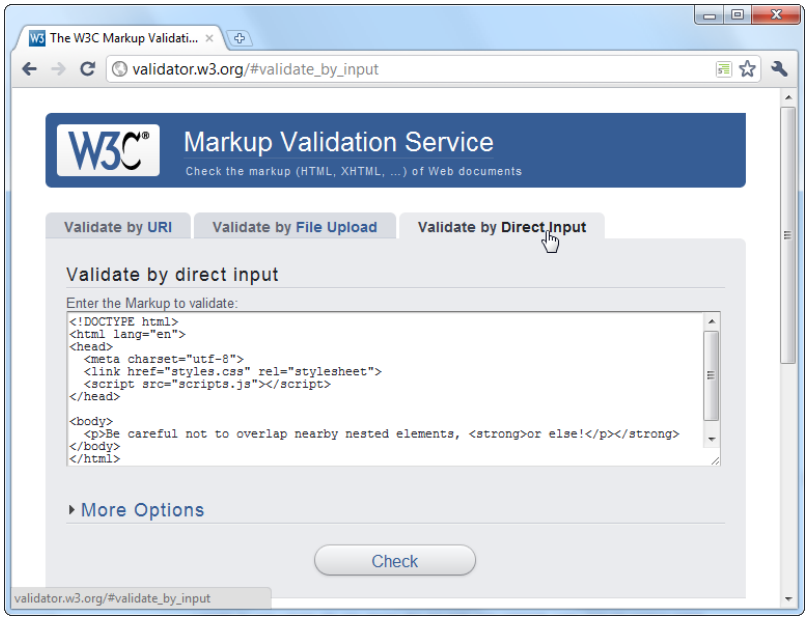


FIGURE 1-2

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.

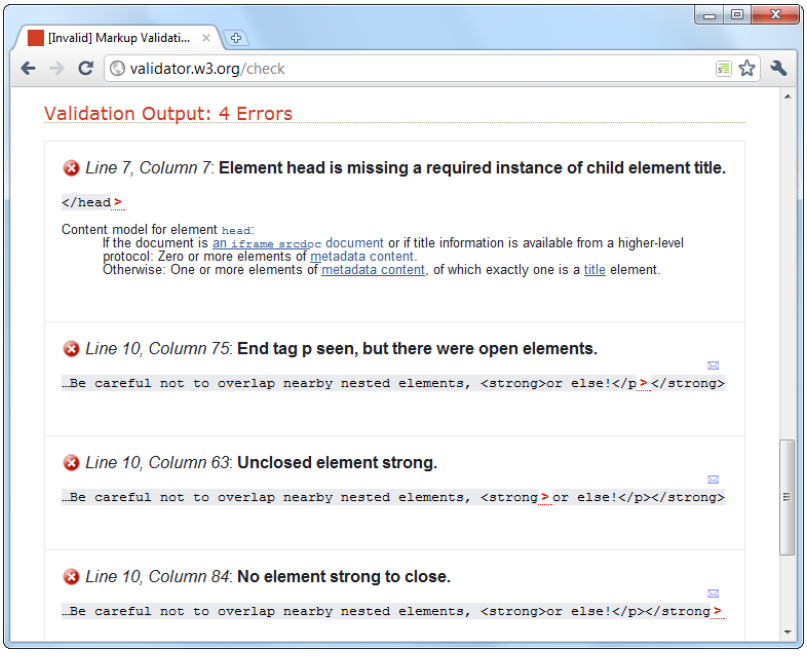


FIGURE 1-3
The validator has discovered four errors in this document that stem from two mistakes. First, the page is missing the mandatory <title> element. Second, it closes the element before closing the element that's nested inside. (To solve this problem, you would replace with .) Incidentally, this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

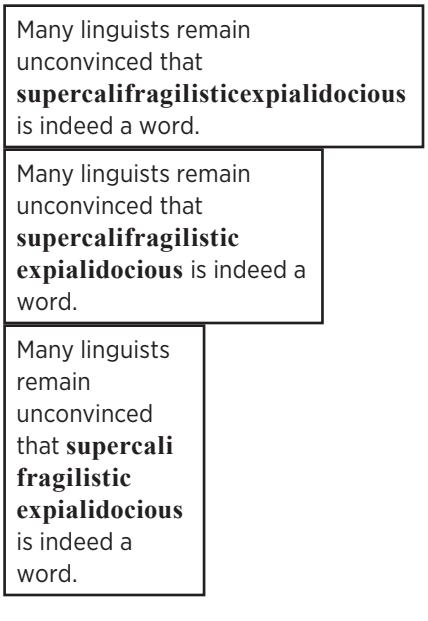
Standardized Elements

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.

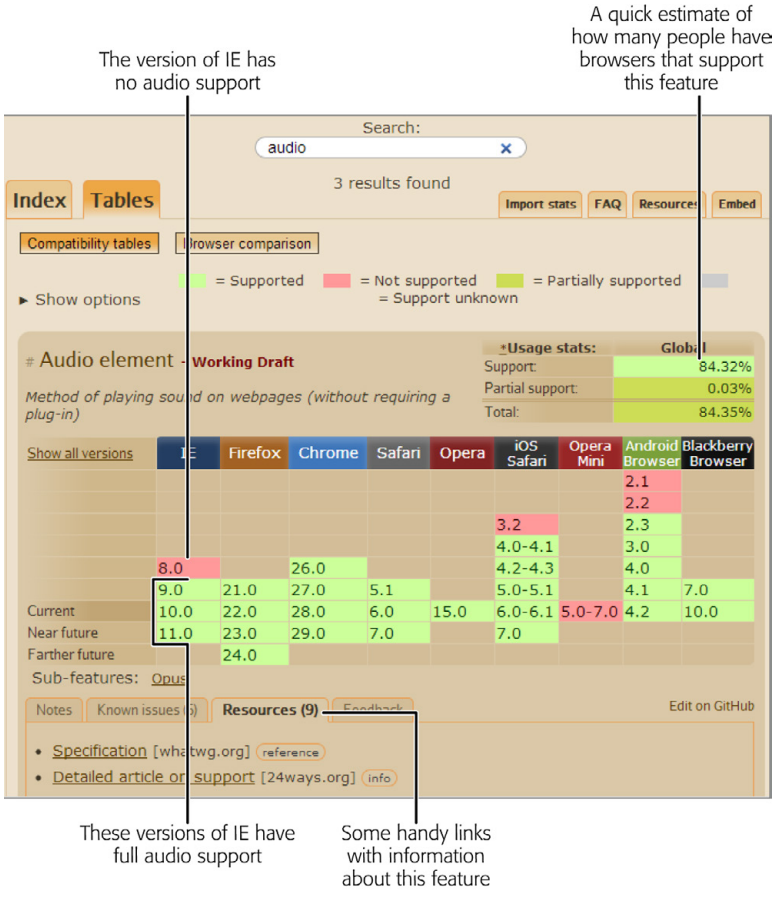


FIGURE 1-4

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).

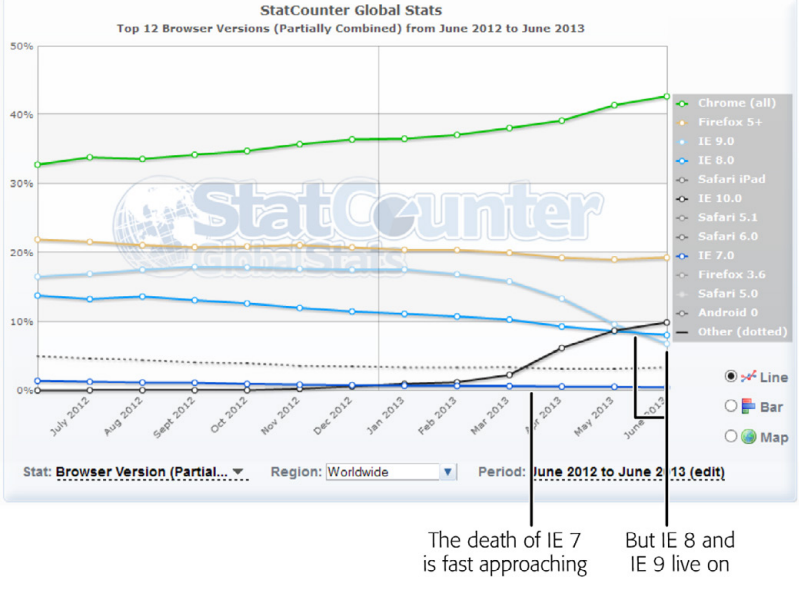


FIGURE 1-5

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.

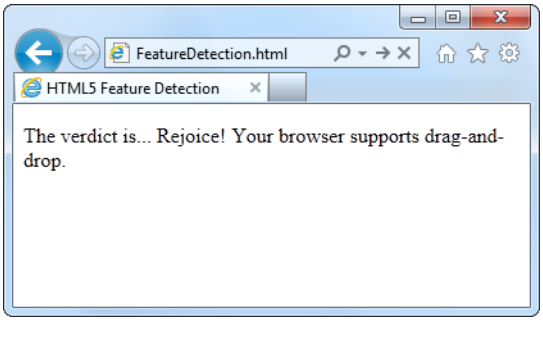


FIGURE 1-6

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).

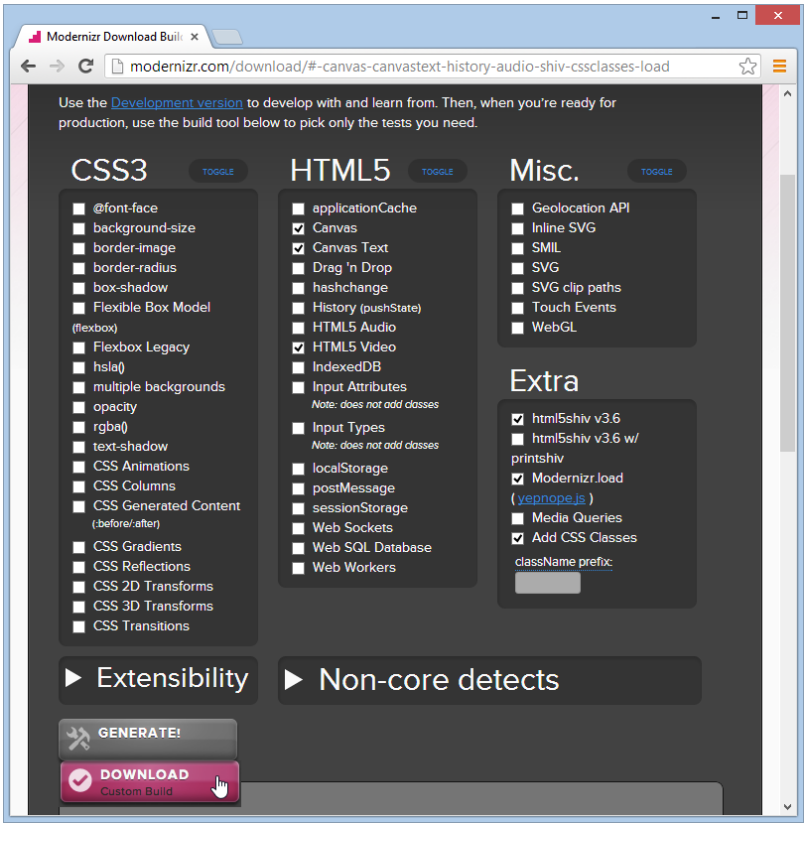


FIGURE 1-7

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.