

Pro JavaScript™ RIA Techniques: Best Practices, Performance, and Presentation

Copyright © 2009 by Den Odell

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-4302-1934-7

ISBN-13 (electronic): 978-1-4302-1935-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Java™ and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc., in the US and other countries. Apress, Inc., is not affiliated with Sun Microsystems, Inc., and this book was written without endorsement from Sun Microsystems, Inc.

Lead Editors: Clay Andres and Jonathan Hassell

Technical Reviewer: Kunal Mittal

Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Project Manager: Sofia Marchant

Copy Editor: Marilyn Smith

Associate Production Director: Kari Brooks-Copony

Production Editor: Laura Esterman

Compositor: Lynn L'Heureux

Proofreader: Martha Whitt

Indexer: Carol Burbo

Artist: April Milne

Cover Designer: Kurt Krames

Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit <http://www.springeronline.com>.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at <http://www.apress.com/info/bulksales>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com>.

PART 1



Best Practices

In this first part of the book, I will present some tried-and-tested guidelines for building rich Internet applications (RIAs). Applying these guidelines will allow you to build the foundations of a web site structure that's scalable from a single page with a few lines of code up to many thousands of pages and thousands of lines of code. I will show you how to follow best practices in a sensible, pragmatic way that won't make the tasks of application maintenance and bug fixing daunting—during construction or in the future.



Building a Solid Foundation

If you're reading this book, chances are that you have felt the proud sense of achievement that comes with building and releasing a web site. Perhaps you completed the project solo; perhaps you built it as part of a team. Maybe it's a simple site, with just a few pages you're using to establish a presence for yourself on the Internet for an audience of a few of your friends, or maybe it's a cutting-edge rich Internet application (RIA) with social networking features for a potential audience of millions. In any case, congratulations on completing your project! You deserve to feel proud.

Looking back to the start of your project with the knowledge and experience you have garnered, I bet you can think of at least one thing that, if done differently, would have saved you from bashing your head against the wall. If you're just starting out in the web development industry, it might be that you wish you had kept a backup of a previous version of your files, because it cost you precious time trying to recover your changes after an unexpected power outage. Or it might be that you wish you hadn't decided to rely on that third-party software library that seemed like it would be up to the task at the start of the project, but soon proved itself to be a huge waste of time and effort. In the course of my own career, I've been in exactly these situations and come out the other side a little wiser. I've learned from those mistakes and fed that new knowledge back into the next project.

Based on my experiences and what I've learned from others, I've developed an effective, sensible approach to web development. This approach, along with a handful of smart techniques thrown in the mix, should minimize those head-bashing moments and ensure things run more smoothly right from the get-go all the way through to the launch of your next web site or application.

Best Practice Overview

Let's start by considering what is meant by the term *best practice*. If you've been in the development profession for long, you'll have heard this expression being tossed around quite a lot to justify a particular coding technique or approach. It is a bit of a loaded phrase, however, and should be treated with caution. I'll explain why.

Who Put the “Best” in Best Practice?

The landscape of web development is constantly changing. Browsers rise and fall in popularity, feature adoption between them is not always in parallel, and the technologies we use to construct web sites for display in such browsers are still fairly immature, constantly undergoing revisions and updates. In an environment that is in flux, what we might consider to be a best-practice solution to a problem right now could be obsolete in six months’ time.

The use of the word *best* implies that a benchmark exists for comparison or that some kind of scientific testing has been adopted to make the distinction. However, very rarely have such tests been undertaken. You should consider carefully any techniques, technologies, and components that have been labeled as *best practice*. Evaluate them for yourself and decide if they meet a set of criteria that benefit you as a developer, the end users of your site, and if relevant, the client for whom you are undertaking the work.

The guidelines, rules, and techniques I set out in this chapter are ones that I have personally tried out and can attest to their suitability for real-world web development. I consider them to be the best we have right now. Of course, some of these could be irrelevant by the time you are reading this book, so my advice to you is to stay up-to-date with changes in the industry. Read magazines, subscribe to blog feeds, chat with other developers, and scour the Web for knowledge. I will maintain a comprehensive list of sources I recommend on my personal web site at <http://www.denode11.com/> to give you a place to start.

By staying abreast of changes to these best practices, you should be able to remain at the forefront of the web development industry, armed with a set of tools and techniques that will help you make your day-to-day work more efficient, constructive, and rewarding.

Finally, don’t be afraid to review, rewrite, or refactor the code you write as you build your sites. No one has built a web site from scratch without needing to make code alterations. Don’t believe for a second that any code examples you see on the Web, or in this or any other book, were written in a way that worked perfectly the first time. With that said, knowledge and experience make things easier, so practice every chance you get to become the best web developer you can be.

Who Benefits from Best Practices?

The truth is that everyone should be able to benefit from the use of best practices in your code. Take a look at the following lists, and use these criteria to assess any guidelines, techniques, or technologies you come across for their suitability for your site.

Web Developers

Best practice starts at home. A site structure and code that work well for you and your web developer colleagues will make all your lives a lot easier, and reduce the pain that can be caused by poor coding.

- Will my code adhere to World Wide Web Consortium recommendations?
- Will my site be usable if a proprietary technology or plug-in is unavailable?
- Will my code pass testing and validation?

- Is my code easily understood, well structured, and maintainable?
- Can extra pages, sections, and assets be added to the site without significant unnecessary effort?
- Can my code be localized for different languages and world regions without a lot of extra effort?

Search Engines and Other Automated Systems

Believe it or not, a large percentage of site traffic is from automated machines and scripts, such as search engines, screen scrapers, and site analysis tools. Designing for these robots is every bit as important as for any other group of users.

- Will my code appear appropriately in search engine results according to sensible search terms used to find it?
- Can my code be read simply and easily by a machine or script that wishes to read or parse its contents for whatever reason?

End Users

The most important users of your code are your site visitors, so making your code work effectively for them is the number one priority.

- Will my code be usable in and accessible to any web browser or device, regardless of its age, screen size, or input method?
- If my site were read aloud by screen reader software, would the content and its order make sense to the listener?
- Can I be confident my code will not demonstrate erroneous behavior or display error messages when used in a certain way I have not anticipated?
- Can my site be found through search engines or other online tools when using appropriate search terms?
- Can my users access a localized version of my site easily if one is available?

General Best Practices

If you're like most developers, you probably want to spend as much of your time as possible constructing attractive user interface components and great-looking web sites, rather than refactoring your code base because of an unfortunate architectural decision. It's very important to keep your code well maintained. Without sensible structure and readability, it will become harder and harder to maintain your code as time passes. Bear in mind that all the guidelines in this chapter have been put together with a view to making things as easy on you, the developer, as possible.

Define the Project Goals

The following are the two most important things to consider while coding a web page:

- How will end users want to use this?
- How will other developers want to make changes to this?

Bear in mind that the end users may not be human. If you were to check the server request logs for one of your existing sites, you would discover that many of your site visitors are actually search engine spiders, RSS readers, or other online services capable of reading your raw page content and transforming it into something else.

This kind of machine-based access is likely to become more widespread over the coming years, as automatic content syndication, such as RSS feeds, becomes more commonplace. For example, content from the popular knowledge-sharing site Wikipedia (<http://www.wikipedia.org/>) is already being used in other places around the Web, including within Google Maps (<http://maps.google.com/>), where articles are placed according to the geographical position of the content described in each article.

Yahoo! and other search engine companies have been pushing for some time for web developers to incorporate extra context-specific markup within pages, so that they can better understand the content and perhaps present the results in their search engine in a different way. Recipes could be presented with images and ingredients, for example; movie-related results could contain reviews and a list of where the movie is showing near you. The possibilities of connecting your code together with other developers' code online like this are vast. By marking up your content in the correct way, you ensure the whole system fits together in a sensible, coherent, connected way, which helps users get the information they are looking for faster.

As for ensuring other developers (including yourself, if only for your own sanity when you return to a project after a long break) can follow your code, you need to consider what the usual site maintenance tasks might be. These usually fall into the following four categories:

- Making alterations to existing pages
- Adding new pages
- Redesigning or modifying the page layout
- Adding support for end users who need the page in other languages, or in region- or country-specific versions

By thinking about these tasks up-front, you reduce the likelihood of needing to refactor your code or rearrange and split up files, so the job of maintenance is made easier. Welcome back, sanity!

Know the Basic Rules

So how do we go about making sure that we get it right in the first place? The following seven rules of thumb seem to sum it up succinctly:

- Always follow mature, open, and well-supported web standards.
- Be aware of cross-browser differences between HTML, CSS, and JavaScript implementations, and learn how to deal with them.

- Assume HTML support, but allow your site to be usable regardless of whether any other technologies—such as CSS, JavaScript, or any plug-ins—are present in the browser.
- Name your folders and files consistently, and consider grouping files together according to purpose, site structure, and/or language.
- Regularly purge redundant code, files, and folders for a clean and tidy code base.
- Design your code for performance.
- Don't overuse technology for its own sake.

Let's go through each of these basic rules in order.

Follow Mature, Open, and Well-Supported Web Standards

Back in the early 1990s, a very clever man who worked at the technology research organization CERN (European Organization for Nuclear Research, <http://www.cern.ch/>), Tim Berners-Lee, invented what we know today as the World Wide Web. He developed the concepts of home pages, Hypertext Markup Language (HTML), and interconnected hyperlinks that form the foundation of web browsing. He also created the world's first web browser to demonstrate his invention.

The project became quite large and eventually took up many resources at CERN. When the decision was made to redirect funding and talent toward building the recently completed Large Hadron Collider project instead, Tim Berners-Lee made the decision to create a separate organization to manage the continuation of standards development for HTML and its related technologies. This new organization, the World Wide Web Consortium (W3C, <http://www.w3.org/>), was born in October 1994.

Since its inception, the W3C organization has documented more than 110 recommended standards and practices relating to the Web. The three that are most useful to readers of this book are those pertaining to HTML (including XHTML), Cascading Style Sheets (CSS), and Domain Object Model (DOM) scripting with JavaScript (also known as ECMAScript, since the JavaScript name is trademarked by Sun Microsystems).

Two popular browsers emerged in those early days of the Web: Netscape Navigator, released in December 1994, and Microsoft's Internet Explorer (IE), released in August 1995. Both browsers were based on similar underlying source code and rendered web pages in a similar way. Of course, a web page at the time was visibly very different from what we see today, so this wasn't particularly difficult for both to achieve.

Roll on a year to 1996, and things get a little more interesting. Microsoft introduced basic support for a new W3C recommendation, CSS level 1, in IE 3. This recommendation defined a way for web developers to apply font and color formatting; text alignment; and margins, borders, and padding to most page elements. Netscape soon followed suit, and competition began to intensify between the two browser manufacturers. They both were attempting to implement new and upcoming W3C recommendations, often before those recommendations were ready for the mainstream.

Naturally, such a variation in browser support for standards led to confusion for web developers, who often tended to design for an either/or scenario. This resulted in end users facing web sites that displayed the message "This web site works only in Internet Explorer. Please upgrade your browser."

Of course, the W3C recommendations are just that: recommendations for browser manufacturers and developers to follow. As developers, we must consider them only as useful as their actual implementation in common web browsers. Over time, browsers have certainly made strides toward convergence on their implementations of these web standards. Unfortunately, older versions of browsers with poorer quality of standards adoption in their rendering of web pages still exist, and these must be taken into account by web developers.

The principle here is to ensure you are up-to-date on common standards support in browsers, rather than just on the latest recommendations to emerge from the W3C. If the standard is well supported, you should use it. If not, it is best avoided.

Deal with Cross-Browser Issues

Web browsers are regularly updated, and they quite often feature better support for existing W3C recommendations and some first attempts at implementations of upcoming recommendations.

Historically, browsers have varied in their implementations of existing recommendations. This is also true of browser support for the newer recommendations. This means that developers must aim to stay up-to-date with changes made to browser software and be aware of the features and limitations of different browsers.

Most browser users, on the other hand, tend not to be quite as up-to-date with new browser releases as developers would wish. Even browsers that are capable of automatically updating themselves to the latest version often require the user to authorize the upgrade first. Many users actually find these notifications distracting to what they're trying to achieve in their web browser then and there, and so they tend to put off the upgrade.

As developers, we must be aware and acknowledge that there are many different web browsers and versions of web browsers in the world (some 10,000 different versions in total, and counting). We have no control over which particular piece of software the end user is using to browse our pages, nor should we.

What we do know from browser statistics sites, such as Net Applications' Market Share (<http://marketshare.hitslink.com/>), is that the five main web browsers in the world today are Microsoft's IE, Mozilla's Firefox, Apple's Safari, Opera Software's Opera, and Google's Chrome. These five browsers account for around 99% of all access to web pages through the desktop. However, just relying on testing in these browsers misses out on the burgeoning market in mobile web browsing, for example, so it is worth staying up-to-date with the latest progress in the web browser market.

Testing your pages across a multitude of browsers and operating systems allows you to locate the portions of your code that cause different browsers to interpret it in different ways. Minimizing these differences is one of the hardest tasks for any web developer and separates this role from most other software-related professions. This is a task that needs to be attacked from the get-go of a new project, as leaving it until too late can result in frantic midnight coding sessions and missed deadlines—never fun!

The smartest approach is to build the HTML, CSS, and JavaScript code that form the basic template or outline of the site before writing any page-specific code. Then test this bare-bones structure in as wide a range of browsers on as many different operating systems, and with as varied a range of monitor and window sizes, as possible. Tweak the code to ensure the template displays correctly before adding any page-specific code or content.

A particular source of variation is in the different interpretations of color within browsers. Some support the reading of color profile information from image files; some don't support this. Some apply a gamma correction value; some don't apply this value. Consequently, the same image or color can appear slightly different in various browsers, so it's worth checking that your design doesn't cause color mismatching to occur between objects on your page.

You should build and test individual page components one at a time in as many browsers as possible during development. Again, by bringing most of the testing up-front to coincide with development, you will experience fewer problems later on and have fewer bugs to squish. By the end of a project, developers are often feeling the pressure of last-minute client requests for changes, so minimizing bugs by this stage in the proceedings is a smart idea.

Assume Support for HTML Only

Your HTML markup must be visible and operate functionally in any available browser, device, or user agent without reliance on CSS, JavaScript, or plug-ins. Where CSS, JavaScript, or plug-ins provide additional content, layout, or functionality over and above the HTML, the end users should be able to access the content and a functional equivalent of the behavior in a sensible way, without reliance on these technologies. For example, if you're using a Flash movie to provide an animated navigation menu for your site, you need to ensure the same navigation is available through HTML; otherwise, you are preventing a whole group of users from accessing your site.

Obviously, this has a massive impact on the way you develop your web pages. You will build from the HTML foundations upward, ensuring no functionality gets lost when certain browser features are switched off or are nonexistent. Each "layer" of code should be unobtrusive; that is to say that no CSS style rules or JavaScript code should exist within the HTML markup—each should be in a separate file and stand alone.

In the context of modern web applications, which are often written in such a way so that communication between the browser and the server is handled via JavaScript, this means that those communication points must exist when JavaScript is switched off in the browser. For example, modern JavaScript allows data to be sent to and received from a web server without the need for the page to refresh when sending a form. In this case, you must ensure that the form can be submitted without the need for JavaScript—treat it like an optional extra, rather than a requirement.

You might hear this principle called *progressive enhancement*, referring to the adding or layering of extra functionality on top of the HTML, or *graceful degradation*, referring to the fact that the removal of features from the browser always results in a working web page. It is the central principle of what is termed *accessibility*, which refers to providing access to a web page regardless of browser or device.

This principle is best understood through real-life examples, so let's go through two of them now.

First, suppose that in your web application, you have a button that, when clicked, launches a login modal dialog box within the page, as shown in Figure 1-1. After the user fills in the form and clicks the submit button, JavaScript is used to send the supplied login credentials to the server, and then to perform a refresh of certain page elements, instead of the entire page, based on the user's logged-in status as shown in Figure 1-2.

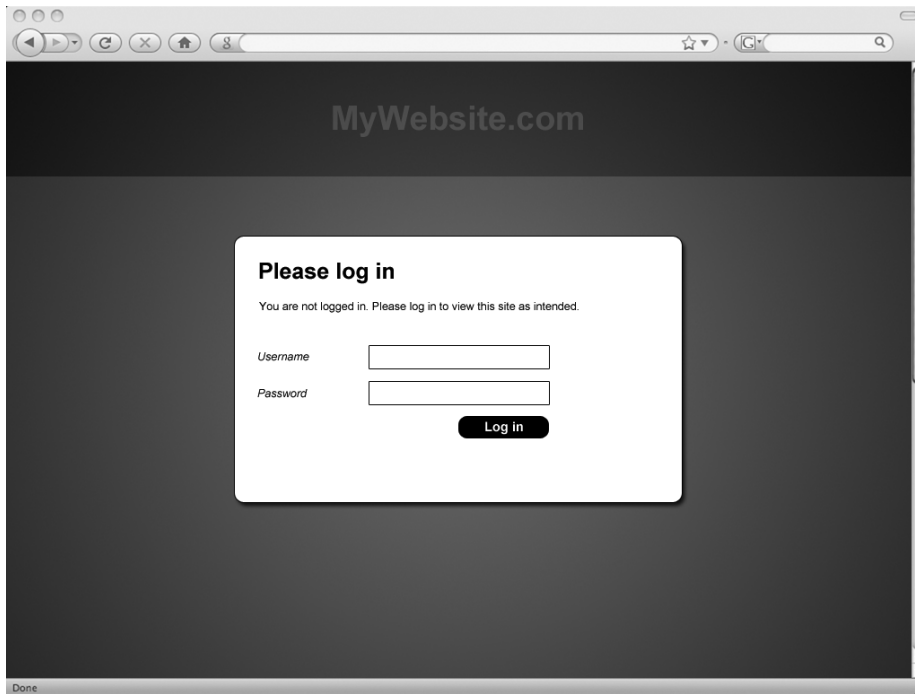


Figure 1-1. A modal-style login box

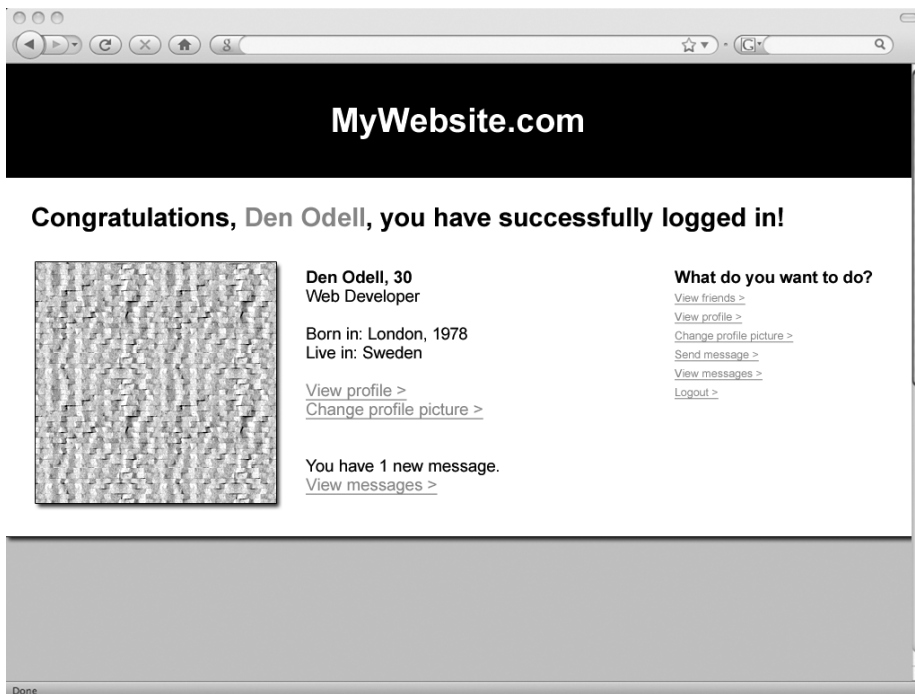


Figure 1-2. Successful login, loaded without a refresh if JavaScript is enabled

But what if JavaScript is disabled? You would need to ensure that your HTML code was structured such that the user would be taken to a separate page with the login form. Submitting this form would post the data back to the server, causing a refresh, and the server-side code would decide which page to send according to the user's status—either successfully logged in or not logged in. In this way, both scenarios are made to be functionally equivalent, although their user flow and creative treatment could potentially be different.

As another example, suppose you have a page that contains a form used for collecting payment information for an online booking system. Within this form, depending on the type of credit card selected, you would like certain fields to display only if the user selects a credit card, as shown in Figure 1-3, rather than a debit card, as shown in Figure 1-4. For instance, the Issue Number field is applicable only to debit cards, and perhaps you want to display the Valid from Date fields only for cards from certain suppliers. You probably also want to make it impossible for the user to submit an incorrect date, such as February 30.

As web developers, we use JavaScript to make this happen. JavaScript fires events when the user performs certain actions within the browser, and we are able to assign code to execute when these events are fired. We even have the power to cancel the event, meaning that if the user attempted to submit a form, for example, we could cancel that submission if we decided that form wasn't suitable for submission because it had failed some validation tests.

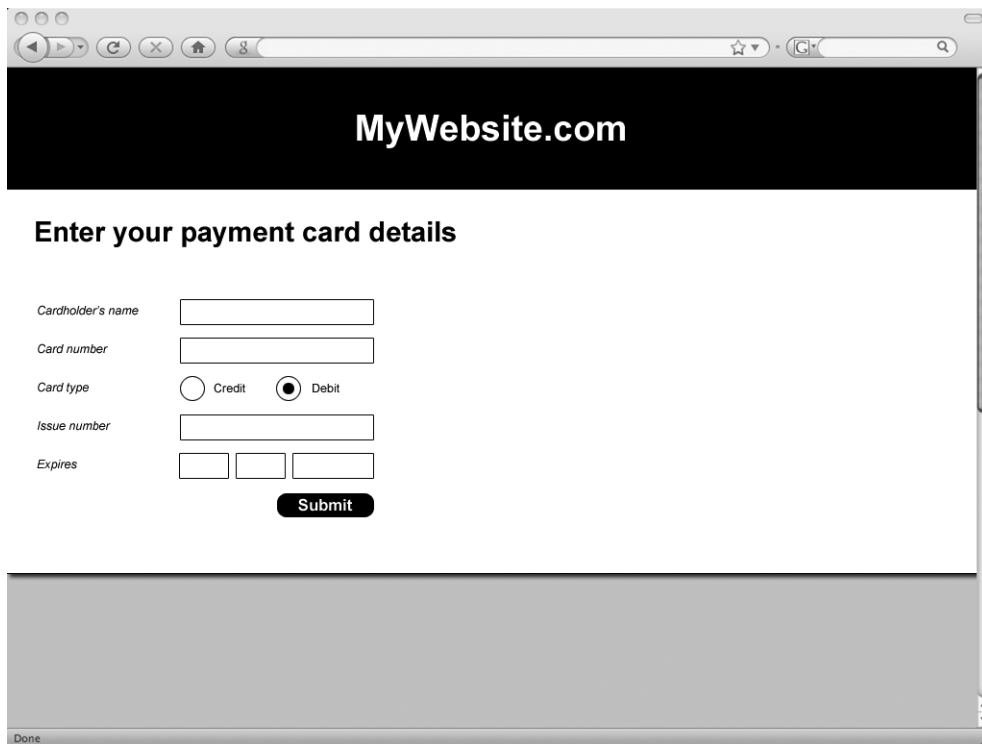
We use JavaScript to “listen” for changes to the Card Type field. This event gets fired when the user selects a different radio button option. When this event is fired, we can execute a piece of code that, depending on the card type selected, shows or hides the desired fields.

The screenshot shows a web browser window with the address bar displaying "MyWebsite.com". The main content area has a black header with the text "MyWebsite.com" in white. Below the header, the title "Enter your payment card details" is displayed in bold. The form contains the following fields and controls:

- Cardholder's name:** A single-line text input field.
- Card number:** A single-line text input field.
- Card type:** Two radio buttons labeled "Credit" and "Debit". The "Credit" radio button is selected.
- Valid from:** Three adjacent text input fields for the day, month, and year.
- Expires:** Three adjacent text input fields for the day, month, and year.
- Submit:** A black button with the text "Submit" in white.

The browser's status bar at the bottom shows the word "Done".

Figure 1-3. A payment card form showing credit card fields



The image shows a web browser window with the address bar displaying 'MyWebsite.com'. The page has a black header with the site name in white. Below the header, the main content area is white and contains the title 'Enter your payment card details' in bold. The form consists of several input fields: 'Cardholder's name' (a single text box), 'Card number' (a single text box), 'Card type' (two radio buttons, 'Credit' and 'Debit', with 'Debit' selected), 'Issue number' (a single text box), and 'Expires' (three separate text boxes for month, day, and year). A black 'Submit' button is positioned below the 'Expires' fields. The browser's status bar at the bottom shows 'Done'.

Figure 1-4. A payment card form showing debit card fields

We also listen for the submit event of the form to fire and, when it does, we run a small JavaScript routine to check that the date fields contain valid values. We can force the form submission to fail if we decide the values entered are not up to scratch.

Now what happens when someone visits your web page with a browser that doesn't support JavaScript? She selects her card type using the radio button, but nothing changes. In fact, in order to instantiate a change to the appearance of the page, the form must be submitted to the server to allow the server to perform the kind of processing you had been performing using JavaScript.

In terms of usability, you might consider it odd to ask the users to submit the form after they have selected their card type, as the fields are already displayed below. Probably the ideal way to structure your page in this case is to have all of the fields existing in the page's HTML, and simply allow the users to fill in the information they have available on their card. When they finally submit the form, the processing that exists on the server can validate their card data and check whether they have entered a valid date, and if there is an error, reload the page displaying an error message.

Name and Group Folders and Files Consistently

By establishing rules and conventions regarding the naming of folders, files, and their contents, you make the task of locating files and code a lot easier for yourself and other developers. The task of maintenance and future additions is made simpler with a consistent naming convention, ensuring developers always know how to name their assets. See the "Structuring

Your Folders, Files, and Assets” section later in this chapter for some examples of directory structures you might adopt.

Maintain a Tidy Code Base

You should ensure that the files and code associated with a project are the only ones necessary for the web site to do its job—no more and no less. Over time, certain files may be superseded by others, and certain CSS style rules or JavaScript files by others.

I recommend that you purge all redundant files, folders, and code from your code base on a regular basis during development. This reduces the size of the project, which aids comprehension of the code by other developers and ensures the end users of your site are not downloading files that are never used, consuming bandwidth that they could potentially be paying for.

To avoid problems with the accidental deletion of files or the situation where you later require files you’ve deleted, you should consider using a source code management system. Such a system will keep backups of changes made to project files and ensure you can always revert to a previous version of a particular folder or file—even a deleted one. See the “Storing Your Files: Version Control System” section later in this chapter for more information.

Design Your Code for Performance

Your site visitors, whether they realize it or not, demand a responsive user interface on the Web. If a button is clicked, the users expect some kind of reaction to indicate that their action was recognized and is being acted upon.

HTML, CSS, and JavaScript code run within the browser and are reliant on the power of the end user’s machine or device. Your code needs to be lightweight and efficient so it downloads quickly, displays correctly, and reacts promptly. Part 2 of this book focuses on performance and explains how you can make your code lighter, leaner, and faster for the benefit of your end users.

Don’t Use Technology for Its Own Sake

Within the wider web development community, you will often hear hype about new technologies that will make your web pages better in some way. Most recently, this hype has focused around the Asynchronous JavaScript and XML (Ajax) technique, which is the practice of communicating with the server using JavaScript within the browser, meaning that page refreshes can be less frequent. This became the favorite technique to be used by web developers on any new project.

The problem is that sites were built so that they worked only with the Ajax technique, and so relied exclusively upon JavaScript. Those users without this capability in their browsers—users with some mobile web browsers, users with restrictions in place in their office environment, users with special browser requirements due to a disability, and external robots such as search engine spiders—could not access the information that would normally have been provided through HTML pages, connected together through hyperlinks and buttons. Conversely, some users with capable browsers were finding that if they remained on certain sites that relied heavily on the Ajax technique, eventually their browser would become slow or unresponsive. Some web developers, keen to jump onboard the new craze, forgot to code in a way that would prevent memory leaks from occurring in the browser.

Build your sites on sound foundations and solid principles, ensuring you test and push new technologies to usable limits before deciding they are a good choice for your project. You'll learn about the Ajax technique in Chapter 2, and how to deal with memory leaks in browsers in Chapter 4.

Markup Best Practice: Semantic HTML

HTML or XHTML forms the basic foundation of every web page. Technically, these are the only web standards that need to be supported by all web browsers and user agents out there in the wild. The term *semantic* in this context refers to applying the correct tags to match the meaning behind the content (according to the dictionary, the word *semantic* literally means *meaning*).

Knowledge of as many of the HTML/XHTML tags and attributes as possible will put you in good stead. Make sure that your content is marked up with exactly the right tag for the content it encompasses: table tags for tabular data, heading tags for section headlines, and so on. The more meaning you are able to give your content, the more capable web browsers, search engine spiders, and other software will be at interpreting your content in the intended way.

It is advisable to include all semantic information in your markup for a page, even if you chose to use CSS style rules to hide some elements visually within the browser. A useful guideline is that you should code your markup according to how it would sound if read aloud. Imagine the tag name were read aloud, followed by the contents of that tag. In fact, this is how most screen reader browsers work, providing audio descriptions of web pages for those with visual impairments.

For example, suppose you've built a web site for movie reviews, and you want to display an image that denotes the movie has scored four out of five possible stars. Now consider how you would want this information to be read aloud—something like, “rated four out of a possible five stars.” Say you put this text within the HTML, so that everyone can access it. But you don't want this text to be displayed on the page; you want only the image of four stars to appear. This is where CSS comes into play. You can apply a `class` attribute to the tag surrounding this text, and target this class using CSS to specify that the text be hidden and an image displayed according to a specified size. The style rules for hiding portions of text in a way that works for all browsers, including screen readers, are covered in the “Formatting Best Practice: CSS” section later in this chapter. The HTML for this part of the page might look like this:

```
<div class="rated-four-out-of-five">  
    This movie was rated four out of a possible five stars.  
</div>
```

Learn the HTML Tags

If you're an experienced web developer who has worked on multiple sites, and you've been marking up your content semantically, you're already familiar with a whole host of tags: `<h1>`, `<h2>`, `<p>`, ``, ``, and ``, to name a few. However, a number of less common tags are rarely at the forefront of developers' minds. Without some of these tags, you risk marking up your documents in the wrong way, missing an opportunity to add meaning to your content for the benefit of your users, search engines, and others.

The following are a few tags that add important meaning for the browser or end user, but are commonly forgotten:

`<abbr>`: Abbreviation, used for marking up inline text as an abbreviation. In many browsers, hovering the mouse over the text reveals the unabbreviated version.

```
<abbr title="et cetera">etc.</abbr>
```

`<acronym>`: Acronym, used for marking up inline text as an acronym. In many browsers, hovering the mouse over the text reveals the elongated version.

```
<acronym title="World Wide Web">WWW</acronym>
```

`<address>`: Contact information for page. At first glance, you may think this tag should be used to mark up postal addresses listed on the page. However, that is an incorrect usage of the tag. It should be used only to mark up the contact details of the page author. (Of course, a postal address could be part of that information.)

```
<address>
  Author: Den Odell<br />
  <a href="mailto:me@denodell.com">Email the author</a>
</address>
```

`<blockquote>`: Long quotation. An important point to note about block quotes that often gets missed is that the tag may contain only block-level elements. Therefore, the quote itself must, at the very least, be enclosed by a paragraph or other block-level element.

```
<blockquote>
  <p>If music be the food of love, play on,<br />
  Give me excess of it, that surfeiting,<br />
  The appetite may sicken, and so die.</p>
</blockquote>
```

`<ins>` and ``: Inserted and deleted copy. `` is used to show that one piece of content has been deleted. `<ins>` shows that another piece has been inserted into a page. For example, these tags might be used on a blog post where the author has, after publication, returned to the piece and edited it to alter a particular sentence. The tags can be used to show this in a semantic way. Often, content within a `` tag will be rendered in the browser as struck through with a line.

There are `50` `<ins>60</ins>` million inhabitants of the UK

Keep these tags in mind as you code your pages. See if you can spot opportunities to work them into your markup to denote the correct meaning of your content.

Tip Keep a reference list of tags and attributes on hand when developing, and revise that list occasionally. A great online resource for XHTML tags and attributes can be found at <http://www.w3schools.com/tags/>.

Start with a Document Type Definition

You should start every HTML or XHTML page with a Document Type Definition (DTD, or DOCTYPE). This should appear before any other HTML tags in a page. The DTD indicates the HTML standard used for the page content, which is important information for any software parsing the page contents. For example, if the browser knows that the content in the rest of the document is XHTML, it then may assume that each tag is closed, that the casing of tags and attributes is consistent, and that tags are properly nested, as these are the rules that apply to XHTML documents.

The DTD is not a standard tag and does not need to be closed. Here is an example:

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

This DTD declares the rest of the page contents to be in the XHTML 1.0 Transitional format and tells the content reader which URL it can visit to get the specification to follow.

By omitting the DTD, you run the risk of having the browser itself try to figure out which standard to use, which can result in some odd rendering bugs.

DOCTYPE Switching

One of the huge complaints that arose from earlier releases of Microsoft's IE browser (up to and including version 5.5) was that it would not actually render some styles as the W3C recommendation suggested. More specifically, the *box model*, which determines how the browser should apply CSS width and height dimensions to a block-level element that also has padding, was not in line with the W3C's recommendation, nor with implementations in other browsers.

Microsoft developers faced a predicament with the release of IE 6: they could either adopt the correct implementation and break the rendering of all existing pages designed for previous versions of the browser, or they could leave it as is and force all developers to use different style sheets for IE than for other browsers. Obviously, neither option was desirable. As a solution, they built in both rendering methods and came up with a way of switching between them using the DTD at the start of the document.

By supplying a DTD that omitted the URL portion, the developer forced the browser into *quirks mode*—the original but incorrect way of rendering the box model within IE. By supplying a DTD with the full URL portion, the developer forced the browser into *standards mode*, complying with W3C standards. Thus, the choice was left to developers to pick which rendering method to use for their site.

DTD Selection

As developers, we want to push forward standards adoption across the Web, both in terms of our code and the software used to interpret it. However, we must realize that simply adopting the latest recommendation from the W3C is not always the smartest move. We must take into account the proportion of existing browsers that support that recommendation.

As it stands at the time of printing, the following selection of DTDs have full cross-browser support and are recommended:

- HTML 4.01 Strict
- HTML 4.01 Transitional

- HTML 4.01 Frameset
- XHTML 1.0 Strict
- XHTML 1.0 Transitional
- XHTML 1.0 Frameset

HTML 4.01 is a trimmed-down version of HTML 4 that emphasizes structure over presentation. HTML 4.01 Strict should be used as a default DTD, unless there is an overriding reason to use another version. It renders the document in the strictest, most standards-compliant mode.

HTML 4.01 Transitional includes all elements and attributes of HTML 4.01 Strict, but adds presentational attributes, deprecated elements, and link targets. HTML 4.01 Transitional should be used if integration with legacy code is required.

HTML 4.01 Frameset includes all elements and attributes of HTML 4.01 Transitional but adds support for framesets. You should not use it, except in cases where using framesets is unavoidable.

As discussed shortly, XHTML is essentially HTML in the format of XML. It can be read by XML parsers, and transformed using Extensible Stylesheet Language Transformations (XSLT) to any other text-based form.

DTD Validation

Selecting a DTD to describe your document doesn't mean that the markup you have written adheres to the specification contained within that DTD. To avoid errors, you should run your page through an HTML validator, which will check your page's adherence to the DTD you specified.

One of the best validators is the online tool supplied by the W3C, at <http://validator.w3.org/>. You can run the validation from a public-facing URL by uploading your HTML file or by directly entering your markup into a text field on the site. Clicking the Check button on the W3C validator site runs the validation. Any resulting errors are listed in document source code order.

How Do You Put the X in XHTML?

As XHTML is essentially HTML with XML rules applied, the rules are the same as for XML:

- All tags must be well formed.
- All elements and attribute names should be in either lowercase or uppercase, as XML is case-sensitive. Many find lowercase to be easier to read.
- Values for attributes must be quoted and provided. Even simple numeric values must be quoted. In instances where the attribute serves as a Boolean on-or-off type of value (for example, the checked and disabled attributes for <input> elements), the value should be set to be the same as the attribute name, as in this example:

```
<input checked="checked" type="checkbox" name="item1" id="item1" value="1" />
```

Unlike HTML, XHTML is well formed—every tag that's opened must be closed. This means that it is simpler for a browser to parse XHTML than HTML, and therefore its use is also suited for mobile applications, where processors are slow and memory small. Specifically for

mobile usage, there is XHTML Mobile Profile (now known as XHTML Basic), a subset of full XHTML.

In fact, the general transformability and readability of XHTML makes it suitable for other web services and computer programs to access and parse, meaning it is incredibly versatile, and as such, its use is highly recommended. All examples in the rest of this book will favor XHTML over HTML.

Well-Formed Documents

An XHTML document is well formed when all elements have closing tags or are self-closing and all elements are nested properly with respect to others. That is to say that any tag opened must be closed before its parent tag is closed.

Here is an example of the correct nesting of tags:

```
<p>
  Here is a paragraph with <em>emphasized</em> text.
</p>
```

And here is an example that shows incorrect nesting:

```
<p>
  Here is a paragraph with <em>emphasized</p> text.
</em>
```

In XHTML documents, elements with no inner content, such as `
` and ``, must be self-closing. A space should be placed between the final character of the tag name and the slash character at the end of the tag. Omitting this space has been known to cause issues with rendering in some older browsers, including Netscape 4.

Element Prohibitions

XHTML does place some restrictions on the nesting of elements. Inline elements cannot be placed directly inside the `<body>` element, for example, and must be wholly nested inside block-level elements. Block-level elements cannot be placed inside inline elements. Certain elements are considered both block and inline, depending on the content within them, such as `<ins>` and ``. It is worth noting which elements these are by looking through your HTML reference guide. Be aware that certain elements cannot contain other elements, such as those listed in Table 1-1.

Table 1-1. Some Tags with Content Restrictions

| Tag | Restriction |
|----------|--|
| <a> | Cannot contain other <a> tags |
| <pre> | Cannot contain the , <object>, <big>, <small>, <sub>, or <sup> elements |
| <button> | Cannot contain the <input>, <select>, <textarea>, <label>, <button>, <form>, <fieldset>, <iframe>, or <isindex> elements |
| <label> | Cannot contain other <label> elements |
| <form> | Cannot contain other <form> elements |

Put Best Practice into Practice

So you're sitting in front of your computer ready to code up an HTML page according to best practices. How do you go about doing that?

Remember that your goal is to write code that is consistent and easy to follow for other developers, and results in a web page that is correctly marked up and accessible to end users. Let's go through some guidelines and rules that should help you follow best practices.

Write Code That's Neat and Tidy

Code indentation enhances code readability. By indenting sections of code based on their grouping or degree of nesting, your code will be easier to read and understand. Remember that one of the goals here is for maintainability by other developers. Just as you tidy up your living room in case you have guests, you should keep your code tidy for when you might have visitors.

Tab characters should be used for indentation instead of whitespace. This facilitates maintenance as well as readability, while reducing the overall weight of a page. In your development environment, you can configure tab spacing to map to a certain number of character spaces. Usually two, four, or eight character spaces are sufficient for readability.

Code blocks residing inside other tags should be indented. For every level of nesting, the code should be indented one tab inward. Tags that contain only text content for display, or inline elements, need not have their content indented with respect to their parent. Take a look at the following example, which shows some typical spacing.

```
<div id="container">
  <p>A paragraph of content</p>
  <table>
    <tr>
      <th>Name</th>
      <th>Value</th>
    </tr>
    <tr>
      <td>Red</td>
      <td>ff0000</td>
    </tr>
  </table>
</div>
```

Cut Down on Comments

I'm sure you've seen many examples of HTML comments strewn throughout web sites. They are in this format:

```
<!-- comment goes here -->
```

Often, they are used to note the beginning and end of particular tags or sections of the page. While this may be useful when trying to establish which server-generated code is doing what to your front-end output, most development environments and built-in browser development tools allow you to locate the starting and ending points of blocks and sections of content, so this usage of comments is somewhat redundant.

Including large numbers of comments within your HTML means the end users must download more markup data to their browser across the network before they have a page displayed that they can interact with. I recommend steering clear of HTML comments, with the following exceptions:

- Where the use of particular markup might seem odd to another developer viewing your code at a later date. A comment can provide an explanation and avoid confusion.
- Where it causes a particular browser to have a certain, specific behavior. This is the case with conditional comments in IE, which we will look at next.

Use Conditional Comments for IE

As of IE version 5 and above (Windows-only), Microsoft added a very useful feature called *conditional comments*. The idea is that if a developer needs to write code to specifically target a particular version of IE, or for IE itself, this can be done by a specially formatted comment tag, rather than by using JavaScript or server-side browser detection. To all other browsers, the contents of the tag appear as a standard comment, so they are ignored.

Here is an example of a conditional comment targeting IE version 6 and above:

```
<!--[if gte IE 6]>
  <p>You are browsing with Internet Explorer version 6 or above.</p>
<![endif]-->
```

The following includes a conditional comment targeting IE users only:

```
<!--[if IE]>
  <p>You are using Internet Explorer browser.</p>
<![endif]-->
```

And, as a final example, this conditional comment targets versions of IE older than version 7:

```
<!--[if lt IE 7]>
  <p>You are using a version of Internet Explorer older than version 7.</p>
<![endif]-->
```

Within conditional comments, *lt* denotes less than, *gt* means greater than, *lte* means less than or equal to, and *gte* means greater than or equal to.

This technique really comes into play with regard to importing external style sheets. Consider the following code, which would sit within the `<head>` block of an XHTML document. It conditionally loads an external style sheet for IE 6 users,

```
<link rel="stylesheet" href="master.css" type="text/css" />
<!--[if IE 6]>
  <link rel="stylesheet" href="master.ie6.css" type="text/css" />
<![endif]-->
```

When they read in this code, most web browsers see a reference to a single style sheet, `master.css`, followed by a comment (between the `<!--` and `-->` comment markers), which they ignore. IE 6 is the exception. Because it identifies the opening of a conditional comment block that specifies that the containing code should be read by only that specific version of IE, it chooses to read and parse the code within that comment.

This allows developers to maintain their master style sheet for all standards-compliant web browsers. But, for those instances where specific versions of IE just won't play ball, they can include a reference to a smaller style sheet containing style fixes to only those elements that are out of whack. In the future, when IE version 6 is a distant memory, developers need only delete the conditional comment and the code within it to remove any version-specific code or styles from their site.

Set the <html> Tag Correctly

After the DTD declaration in every HTML document comes the <html> tag, wrapped around the rest of the document's contents.

For HTML documents, this can be left as just the tag without any attributes specified, but in the case of XHTML, certain attributes must be included. Take a look at this example from an XHTML document:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-GB" lang="en-GB" dir="ltr">
```

This example has the following components:

xmlns: This attribute defines the namespace for custom tags within the document (this is required, although it is rarely used in real-world applications at present).

xml:lang and lang: These attributes specify the language, such as en for English (and often including the region locale, such as en-GB for Great Britain) of the entire document. Where the language of specific content changes in the document, such as when using the French expression *c'est la vie*, this text must be marked up with a tag surrounding it. In many cases, you would use a tag, setting the xml:lang and lang attributes for this element. In the preceding example, this attribute's value would be fr-FR.

dir: This specifies the text direction of the content within the document. For most Western languages, this value will be ltr (left to right). However, if the text direction changes within the content, you should note this by setting the dir attribute on a tag surrounding that content. The only other possible value is rtl (right to left).

Specify the Content Type

It is advisable to specify to the browser or user agent the content type of the document, in case it is incapable of reading this directly from the file itself, as it reduces assumptions and ensures that the content is readable in the form it was written. Use the <meta> tag within the <head> part of the HTML file to do this, as in this example:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

In this line of code, the browser is told that the content type is in the UTF-8 character set. This particular character set is very flexible, in that most worldwide characters can be inserted directly into your document without the need for conversion to HTML entity codes. This is especially useful for characters outside the standard English alphabet.

Set the Page Title

The `<title>` tag is an essential part of any document. It needs to be a single heading that describes the document in as few words as possible. It is used as the text for the link to your page within search engine results, and is displayed at the top of the window within your browser.

With this in mind, you might decide to reveal some other information in your page title about the location of the page within the site structure, to provide some context for those who have stumbled across your page through a Google search. The following format seems sensible enough to portray this information while being readable and fairly succinct:

```
<title>Page title - Section name - Site name</title>
```

The *Page title* in this example will almost always match up with the main title of the page, usually contained within the `<h1>` tag of the document. Of course, you can use whichever separator you like between the values; the order is the essential part.

It is sensible to ensure that each distinct page in your site has a unique page title within its section, so that duplicate results with the same name do not appear in search engine results or on a site map.

Separate Presentation, Content, and Behavior

It is important to separate the content of a document from the code needed to apply its design and layout. This level of separation allows you to make style changes easily, without needing to alter the markup. You can even swap out the entire layout of the web site for another layout fairly simply, without affecting its content.

You should avoid using inline styles within your markup (set with the `style` attribute of many tags), as this makes maintenance of your pages incredibly difficult. Developers should know to look within style sheet files for everything involving style and layout, and HTML documents for everything regarding content. The two should never be intermingled. You should also keep all your JavaScript code outside the HTML document.

Instead, within your HTML, reference style sheets through the use of the `<link>` tag. Reference JavaScript code with the `<script>` tag. Adding `class` and `id` attributes to tags within your HTML should be the only method for providing the connections between these files; style attributes and JavaScript on-method handlers should not be mingled with your content.

It is possible to reference style sheet files according to the device on which you want those styles to be displayed. For example, you will probably want to include separate style sheets for the printer and the screen, as in this example:

```
<link rel="stylesheet" href="master.css" media="screen" />  
<link rel="stylesheet" href="master-print.css" media="print" />
```

Where the `media` attribute is specified on a `<link>` tag, the printer will read a style sheet only when that attribute contains the `print` value, and the screen will read a style sheet only when this attribute contains a value of `screen`. This allows you to style your content differently depending on the presentation media. For example, on the printer, you probably need little more than the basic content of the page. You could create styles that hide the navigation, header, and footer of your page, leaving only the main body of content, which is usually what most people printing your page want to read when they are away from their browser.

Add a Wrapper Element to the Whole Document

Within the `<body>` tag of your pages, you often will want to add certain layout styles to the whole page. You will soon discover that applying these styles to the `<body>` tag alone is not sufficient for all the positioning and layout you wish to perform. As a solution, I recommend that you add a “wrapper” element, usually a `<div>` tag, around the page content, and place the extra layout styles within this element, rather than using the `<body>` tag itself.

A `<div>` tag merely defines a block of content and adds no extra meaning to the content within. Use the `id` attribute to set an appropriate name for this element to which you can hook the CSS styles, like this:

```
<body>
  <div id="page">
    ...
  </div>
</body>
```

If your design is really simple, you may be able to get away with just using the `<body>` tag. However, consider adding a wrapper element anyway, since you never know how your design may change in future.

Help CSS and JavaScript Target Individual Pages

A practice I advocate is to add a unique `id` attribute to the `<body>` tag on each page in your site. Suppose that you have generic styles created for a multipage web site, and you wish to target one specific style slightly differently on a particular page. In this case, you will need to create an exception to the rule. If each page on the site has a unique `id` attribute, you can create a style that targets just this one page over all others.

Imagine you wish all paragraph text in your site to appear with black text on a white background, but on your home page, you want it to be inverted: white text on a black background. The home page will have a unique `id` attribute, like this:

```
<body id="homepage">
```

and the styles for the paragraph text will look something like this:

```
p {
  background:white;
  color:black;
}

#homepage p {
  background:black;
  color:white;
}
```

In addition, by using the `id` attribute in this way, you can easily get a reference to a specific page with JavaScript code, using the `document.getElementById` method:

```
var homepage = document.getElementById("homepage");
```


Name Your ID and Class Attributes Consistently

You should wrap each logical section or block of the body copy in its own `<div>` tag using sensible, contextual values for their `id` attributes. Certain elements are common across most pages and sites, including the navigation, header, and footer. These types of elements should be kept consistent across pages and projects, where possible, to aid with future maintenance. This allows developers to recognize elements, even when working with unfamiliar pages.

Consider using `id` attribute values along the lines of the following:

```
<div id="header">
<div id="content">
<div id="aside">
<div id="footer">
<div id="navigation">
```

You should name your `id` or `class` attribute values according to the type of content they enclose, rather than any style they use. For example, this form:

```
<p class="error">
```

is better than this one:

```
<p class="red-text">
```

Tip Using a hyphen (-) to separate words in `id` and `class` attribute values makes them easier to read.

Labeling your `class` attributes with content-related information means that when your site designers alter specific parts of the page, you won't need to change your markup; only your style sheet will need modification. It also means that the HTML document stands alone as a description of only the content within, rather than any external file, style, or code.

Order Your Content Correctly

Getting the order of the content within your document correct is just as important as ensuring that it is marked up with the correct tags. Remember that you need to make sure that when the page is read aloud, the most important page content—the main body or subject of the page—is read first, and the less important content—such as navigation links and copyright information—is read last. Screen reader software vocalizes the content of the document strictly in source code order, which means what comes first in your markup is read aloud first.

So that users of screen readers get the content in the most meaningful way, you should organize the `<body>` content of your markup in the following order:

- Page title
- Short list of links that jumps the reader straight to content further down in the page (in-page links)
- Main body content

- Aside content (sidebar, related content, or context-sensitive links)
- Main navigation
- Footer and copyright information

You can use CSS to alter the visual layout of the page so that it matches designs that you must work with. This is explained in the “Accessibility Guidelines for Styles” section later on in this chapter.

Separate Foreground Images from Backgrounds

It is important to make a distinction between those images on your page that are directly related to the content and those that relate to the layout or template of the site.

Images that are directly relevant to the content of the page—figures, charts, pictures of your pets, and so on—should be marked up using the `` tag, as you might expect. All other images, including company logos and icons, should be referenced using background images within your style sheet files. This provides another way to distinguish page layout from relevant page content.

Ask yourself, “If I printed out only the body content of this page, would this image be out of place, or is it contextually relevant to the content?” In practice, you will find that the majority of image files will be referenced from within CSS and not be marked up with the `` tag.

In those instances where you do use `` tags, remember to use the `alt` attribute to describe the content of the image for those users who are unable to view images in their browser for whatever reason. If the image is complex, such as a graph or chart, it might need a more detailed explanation. In this case, consider using the `longdesc` attribute to point to the URL of a page that contains an in-depth description of the information conveyed by the image.

Use Tables Properly

Up until a few years ago, it was common practice to position page components using HTML `<table>` tags. The `<table>` tag should be used only to represent tabular data, and never for positioning content on the page; CSS is more than up to the task of positioning content and providing page layout.

There are many ways you can add extra semantic information to tabular data, all of which aid accessibility to the information contained within the table, and so are worth implementing. In a nutshell, these are as follows:

- Use the `<caption>` tag after your opening `<table>` tag to add a caption to the table. Remember that you can always use style sheets to hide the contents of this tag if you don’t want them displayed, but it is wise to add as much semantic data as possible to your pages.
- Use the `summary` attribute of the `<table>` tag to provide a brief overview of the contents of the table and what it aims to represent. This attribute can be read aloud to summarize the table’s contents.

- Group the header, body, and footer sections of the table together using the <thead>, <tbody>, and <tfoot> tags. Be aware that these must be placed in a specific order: <thead> and <tfoot> first, followed by <tbody>. This allows the browser to display the header and footer rows of the table while the rest of the content may still be loading.
- Use the <th> tag to mark up header cells and <td> for actual data.
- Give each header cell a unique id attribute value. Then, for each data cell, assign its headers attribute value to be a comma-separated list of the id values of the associated header cells.

The following is an example of a table using all of these markup techniques:

```
<table summary="Table showing that the average age of students in this course is 26">
  <caption>Table of students registered for this course with their ages</caption>
  <thead>
    <tr>
      <th id="student-name">Student name</th>
      <th id="age">Age</th>
    </tr>
  </thead>
  <tfoot>
    <tr>
      <th id="average-age">Average age</th>
      <td headers="age, average-age">26</td>
    </tr>
  </tfoot>
  <tbody>
    <tr>
      <td headers="student-name">John Lewis</td>
      <td headers="age">24</td>
    </tr>
    ...
    <tr>
      <td headers="student-name">Peter Jones</td>
      <td headers="age">28</td>
    </tr>
  </tbody>
</table>
```

Improve Your Forms

You should group logically related sections of a form together using the <fieldset> tag, with each containing one <legend> tag to provide a header for that particular grouping of fields within the form. For example, a credit card application form may contain sections for personal information, credit history, and bank details. In this case, the form could be created with three <fieldset> tags with <legend> tags of Personal information, Credit history, and Bank details, respectively, as follows:

```
<form method="get" action="/">
  <fieldset>
    <legend>Personal information</legend>
    ...
  </fieldset>

  <fieldset>
    <legend>Credit history</legend>
    ...
  </fieldset>

  <fieldset>
    <legend>Bank details</legend>
    ...
  </fieldset>

  <input type="submit" value="Save" />
</form>
```

Make sure that each field within your form that has an associated text label has that label marked up with the `<label>` tag. This tag takes a `for` attribute, which you should set to the same value as the `id` tag you set on the associated field. The default browser behavior in this case is to make the `<label>` tag clickable, bringing the focus of the form into the associated field, and allowing the user to interact with that form field when the label is selected. To allow for more scope of applying style rules to form fields and their associated labels, add a `<div>` wrapper tag around each field, which lets you target the fields through style sheets and set spacing and other visual properties. Here is an example:

```
<div class="field">
  <label for="first-name">First name</label>
  <input type="text" id="first-name" name="first-name" />
</div>
```

Pay special attention to the text used within the `<legend>` and `<label>` tags within a `<fieldset>`. When these field labels are read aloud by certain screen reader software, the text within the `<legend>` tag is prefixed to the text in the `<label>` tags to provide extra context to the listener. You must ensure that when this is read aloud, it makes sense to the listener.

Avoid Using Frames

You should avoid using framesets at all cost. They make the content and layout of a site difficult to maintain, are a pain to use on small-screen devices (such as mobile phones), and make navigating around content difficult for end users who do not use a mouse as their primary form of input to their computer.

Inline frames (using the `<iframe>` tag) should be used sparingly, if at all. They can be confusing to the end user and lack the inherent accessible nature of a single-page HTML document.

Accessibility Guidelines for Web Content

If you are not familiar with the Web Content Accessibility Guidelines (WCAG), read these W3C recommendations at <http://www.w3.org/TR/WCAG20/>. These guidelines denote three different levels of accessibility compliance of a web page: A, AA, and AAA, where AAA is the most accessible content.

The accessibility guidelines relate not only to technology, but also to creative design, copy writing, and information architecture, which together provide the total experience for the end user. AAA compliance is the holy grail of any web site developer and should always be strived for. However, AA compliance is usually achievable and is a happy medium for developers who are impeded by time or design constraints. Following the guidelines in this chapter will get you well on the way to providing the most accessible experience you can through technology, but you must be aware of how different users interact with web pages, so I encourage you to read the WCAG document.

Don't Be Fooled by Access Keys

Once touted as a fantastic addition to web pages, access keys are keyboard shortcuts that allow users to jump to certain content on the page or to external pages. However, modern thinking has reasoned their use away, and you should avoid using access keys.

Keyboard shortcuts are very useful to end users who do not use a mouse. In fact, those users often have their own shortcuts set up within their operating system. Unfortunately, access keys often conflict with user-defined keyboard shortcuts. Users will find it confusing when they think they are using their own shortcut, but are actually using the access key shortcut instead, or vice versa.

Access keys can also be confusing because different sites typically use different keys to perform the same action. This means that the end users must become familiar with a different set of shortcuts for each web site visited. This is not expected in the world of computer software, where certain keyboard shortcuts usually perform the same action, regardless of the program.

One of the golden rules of accessibility is to reduce confusion among end users. So, we must regard access keys as nice in theory, but bad in practice.

Don't Be Fooled by Tab Indexes

Another so-called accessibility addition that you should steer clear of is tab indexes. These set the order that links, form fields, and so on become active as the user presses the Tab key on the keyboard. Unfortunately, it is not possible to simply add a tab index to a single field. If you specify a tab index for one element, it must be specified for every element; otherwise, you cannot maintain control over the tab order. The daunting task of maintainability and its limited impact on accessibility makes this a write-off.

A better solution is to ensure your document is well structured, with important content toward the top of the page, and to specify links and form fields in the source code in the order that you wish for them to be tabbed. You can then use style sheets to position these elements on the screen in the desired location or order for visual effect.

Don't Rely on Plug-Ins

By definition, a browser plug-in is not inherently accessible. Since it is not part of the browser itself, the presence of any plug-in must never be assumed. If the end user does not have a particular plug-in, such as Adobe's Flash Player, alternative content should be provided, as explained earlier in this chapter.

It is not wise to wrap an entire web page within a Flash movie, as is so often the case on the Web. The content within the movie remains inaccessible to users of certain browsers and those without the plug-in, and is invisible to most search engines. Instead, consider the use of Flash components on your page—perhaps a movie trailer displayed within a Flash movie container or a special creative treatment to a navigation bar. Then ensure that those users without the plug-in are able to access the content in an alternative way. For example, you might include a transcription of the movie trailer or a plain navigation bar without creative treatment. In this way, you are creating beautiful, smart web pages without sacrificing accessibility.

Add Extra Semantics Where You Can

Some open standards are emerging for adding extra meaning to certain blocks of content within the document markup. One such standard is known as *microformats*. Microformats are blocks of HTML that represent things like people, events, tags, and so on in web pages. They are readable by humans and machines, and entail little more than assigning certain values to certain attributes.

The hCard microformat, for example, is used to mark up address card information, similar to the vCard format, which is a commonly used for exchanging address information between computer software. The following is an example of using the hCard microformat to mark up a name, web site URL, and mobile telephone number:

```
<div class="vcard">
  <a class="url fn" href="http://www.denodell.com/">
    Den Odell
  </a>
  <span class="tel">
    <span class="type">Mobile</span>
    <span class="value">+441234567890</span>
  </span>
</div>
```

The actual tags used are unimportant. Rather, it is the attribute names that are detected by software built to recognize them, including some existing web browsers. Style sheet rules can be used to hide any of the pieces of content that you do not want displayed on your page and will leave them as the pure semantic data within the markup.

Tip Other microformats exist for marking up calendar events (hCalendar), content reviews (hReview), and more. New microformats are being recommended as this book is being written. Follow the progress and see more examples online at <http://microformats.org/>.

Formatting Best Practice: CSS

As a developer, you have most likely opened a style sheet file that someone else worked on and thought to yourself, “How am I meant to understand this?” Once you separate all layout styles from the markup of the document, you will notice just how many style definitions make up a single page or site!

Because of the sheer number of style definitions required to lay out a page, it soon becomes cumbersome to maintain the layout without sticking to some sensible guidelines for structuring CSS files and styles.

Regarding Pixel-Perfect Reproduction of Designs

As developers, we should always strive to build pages that match the creative designs we are working from as pixel perfectly as possible. However, sometimes the effort required for this pixel-perfect consistency is not worth the visual gain that comes from it.

Consider form fields, for example. I have lost track of the number of creative designs I have worked from that showed `<select>` drop-down boxes with custom-designed handles (the *handle* is the downward-pointing arrow on the edge of the box). Those of us who have attempted to style these boxes in the past—and believe me, I have tried—have discovered the extreme variations between browsers.

It does appear that the more modern the browser, the more control we have over the native controls. That doesn’t help the majority of us, however. Many of us are still required to support older browsers such as IE 6, however arcane their form controls may appear.

The solution is to take a best-effort approach. For those browsers that can support it, offer the custom-styled form controls. For older browsers that don’t render the required results, it’s necessary to be pragmatic.

All major browsers allow you to style the background color, text color, font, and line height (although the height specification can give different results in different browsers, so some experimentation is needed). Not all browsers allow you to specify a border style or color, or to customize the appearance of a drop-down box handle. You should ensure your client is aware of this fact in advance. Otherwise, you will need to adjust the designs accordingly to achieve an identical appearance across different browsers.

From experience, and a healthy dose of common sense, we know that most end users run only one browser on their machine. Furthermore, most users are not examining the design elements. Typically, they will not spot a border on a `<select>` box that doesn’t match the other form fields on the page. End users are mostly concerned with whether they can get the information they want from your page quickly.

The following are a couple of rules of thumb for where to draw the line with cross-browser variations using the same CSS:

- You may exclude certain design aspects of the page if the development time required to build the support in the first place is unreasonable for the visual gain produced.
- You may exclude certain design aspects of the page if the maintenance tasks required to make changes to the code base in the future to support the layout would be too cumbersome to expect of any developer.

W3C CSS Standards

CSS 1.0 was recommended by the W3C to developers back in 1996. Its support is virtually universal in about 99% of the browsers in common use today.

CSS 2.1 is the current recommendation, and while it is not universally supported in all browsers (currently limited to the Firefox 3, Safari 3, IE 8, Opera 9.5, and Google Chrome versions of the most common browsers), partial support can be found in that same 99% of browsers. Support exists for CSS positioning (also referred to as CSS-P before CSS 2.1 reached recommendation) to place elements on the page at absolute locations with reference to the top-left corner of the browser window, and at relative locations with respect to the element in which they are contained.

Currently seeking recommendation is CSS 3 (though it has been in this stage for a long time, as suggestions keep being made for additions). The best CSS 3 support so far can be found in Safari 3, but recent releases of both Firefox and Opera have increasing levels of support. Don't be afraid to use CSS 2.1 and CSS 3 style rules, as long as you ensure you can adequately represent your design in as many browsers as possible.

Guidelines for Style Sheets

By following CSS best practices, you can achieve the main goals of making your web pages fully accessible and your code easy to maintain. The guidelines presented in this section will help prepare your style sheets for any changes that may be required in the future, while ensuring the needs of your end users are met.

Separate Common Style Rules

It is highly desirable to include on each page only the CSS style rules required to render that page. Having extra style rules that are not used results in extra data the browser must download and interpret before it knows to dismiss that data. However, two things must be kept in mind: maintainability and an understanding of how the browser works.

Consider a web site dedicated to up-to-the-minute sports scores, which is made up of four pages. The home page describes the purpose of the site to the visitor and provides links to the other pages. The other pages include one with the sports scores, a news update page, and a frequently asked questions (FAQ) page. According to the guideline, the only style rules on each page would be those required to render that specific page.

Now consider how these web pages might look on the screen. Most web sites have components that exist across every page, such as a header section with the site logo and navigation and a footer with copyright information. Also, most site designs are based on a layout template, so each page fits the same rough dimensions and adopts a grid layout specified by the designer. Think of the maintenance nightmare of having to update the header, footer, and general page layout if their styles were duplicated in each of the four style sheet files for the site's four pages. A more sensible approach is needed to ensure maintainability and avoid style rule duplication.

All major web browsers seek to give the end user the fastest and most pleasurable web-browsing experience they are able to offer, and they adopt many approaches in order to do that. One such approach is called *caching*, which involves storing a copy of all the HTML, images, style sheets, JavaScript files, and other static assets requested to display the web page locally on the end user's hard disk. This means that when the users click the web browser's back button or revisit a page, the browser only needs to look on the hard disk to retrieve the

data required to display the page. This avoids downloading these assets again over the comparatively slow Internet connection, so the page is displayed much faster than before.

In simple cases, an image—the site logo, for example—is downloaded once when the user visits the home page. Provided that each page is pointing to the same image file on the web server, and that image has not been updated in the meantime, the web browser will take that image from its cache on the hard disk, rather than waste time downloading it again. We can take this same principle and apply it to style sheets.

Let's say you have one style sheet for your web site that contains only the styles for the general page layout and common components that appear across all pages. You include a reference to this same file on each of your pages. The web browser will download this file once, when the end user first visits a page on the web site (usually the home page), and thereafter on subsequent page visits, the browser will use the file already downloaded. This speeds up the rendering of pages, as time isn't spent downloading more files than are necessary. This has the added benefit of keeping all the page styles common to the entire site together in one file, so you no longer have style rule duplication across your pages.

You can lay out the page-specific style rules in single files of their own, so each page will reference two external style sheets: one common to the site and one unique to the page.

Understand Cascade and Specificity

Two important terms used when discussing style sheets are *cascade* and *specificity*. Cascading (the letter *C* in the acronym CSS) is used to describe how styles are applied in an additive fashion. Specificity describes the specific levels of instance at which a particular style is applied to the page.

The most sensible way to organize your style rules is to start off writing generic styles and increase the specificity of your styles as you progress. Often, the more generic your styles, the more efficient your style sheet files can be. Try to avoid overspecifying a style rule. There is no need to list every tag, ID, or class applied to the element you are styling—keep things short and simple.

As an example, consider an HTML page with the following markup for the `<body>` tag:

```
<body>
  <div id="page">
    <div id="header">
      <h1>My Page Title</h1>
    </div>
    ...
  </div>
</body>
```

Rather than specifying the following to apply styles to the heading tag:

```
body #page #header h1 {
}
```

specify something simpler:

```
#header h1 {
}
```

Since you know that all `id` attribute values within HTML must be unique, there can be only one element on the page with the `id` value of `header`, so specifying its parent elements in the style rule adds redundancy. You do not need to specify the `body` tag selector either, since the only tags you are able to apply styles to are within the `<body>` tag, by definition. Also, in the preceding new style rule, you are able to keep `header` as a distinct component of the page that is not restricted by its parent element. If you choose to move the `<div id="header">` tag around the markup and place it within a different parent element, you know the style rule will apply to it in the same way.

Let's consider another example. Suppose you want two paragraphs of text to appear in the same font, and differ only in their size and color. Here's the HTML:

```
<body>
  <div id="page">
    <p>The quick, brown fox jumped over the lazy dog.</p>
    <div class="alternative">
      <p>The five boxing wizards jump quickly.</p>
    </div>
  </div>
</body>
```

Now check out the style rules:

```
#page p {
  font-family: Verdana, Arial, Helvetica, sans-serif;
  font-size: 1.3em;
  color: #000;
}

#page div.alternative p {
  color: #f00;
  font-size: 1em;
}
```

The styles described in the most generic style rule, the former of the two rules in the preceding code, will be applied to all paragraph tags within the `<div id="page">` tag. The latter style rules are applied to the more specific paragraph tags within the `<div class="alternative">` tag. This additive effect is what we call *cascade*.

Also of interest is the `!important` keyword, which can be added to the end of any style to force that rule to take precedence over any other rules at a more specific level. You could add this keyword to the preceding example, like this:

```
#page p {
  font-family: Verdana, Arial, Helvetica, sans-serif;
  font-size: 1.3em;
  color: #000!important;
}

#page div.alternative p {
  color: #f00;
  font-size: 1em;
}
```

Since the color of the more generic style has been marked as `!important`, its style will take precedence over a more specific instance of that style. In this case, the color `#000` is applied to all paragraph text within the sample page.

In many cases, the usage of the `!important` keyword to enforce incorrect precedence is wrong, and is the sign of a poorly architected style sheet. Needing to return to the style sheet file later to figure out why the styles specified are not being applied is not worth the time or headache. Consider refactoring your styles if you find yourself needing to use the `!important` keyword, so the correct rules are applied to the desired elements. In some cases, this may mean adding a little extra markup to the page to ensure the styles can be applied in the correct way. But this is better than having a poorly structured style sheet with confusing style rules.

Think About the Printer

When building the style sheets for a site, most developers neglect the printer, though rarely deliberately. It is important to know what does and what does not get printed when your users click that print button on their browser's toolbar. There are three items you need to consider:

Wide content: Content wider than the width of a printed page is usually chopped off—not printed at all. Some browsers allow scaling of content to fit the printed page, but this can result in text that's very difficult to read, depending on the width of your site layout.

Background images and colors: Since you should be including as foreground images only those graphics that directly relate to that page's content, all other images (such as logos) should be background images. In many cases, all the images on the page will actually be background images, specified within external CSS style sheet files. Most browsers will not print those background images and colors. This was a smart move on the part of the browser vendors to save ink and print only what's necessary: the textual content of the page. As noted earlier, in most cases, users print a web page to read its main body content—whether it's a train timetable, book review, or bank statement. Rarely will a site visitor want to print the page exactly as it appears on screen. For those who do, there are often hidden preference settings within the browser to enable the printing of background images and colors, but these options are rarely switched on by default.

Text color: Most users in an office environment will have access to a laser printer only, which will often print in black and white. Most home users will have an inkjet printer capable of full-color, photo-quality printing. Consider the lowest common denominator and test print your pages on a black-and-white printer, if you have access to one, or by choosing to print on your inkjet printer with black ink only. Text color also comes into play when you consider background images. Suppose your web page has white text on a dark background. Since that background color won't print, you will be left with white text printed on your crisp, white paper. Of course, the printer has no white ink to print with, so the user actually gets a blank page!

A printout of a web page usually shows large chunks of blank space where background images and colors once were, often with some content chopped off the page and some light-colored text missing entirely. This is hardly what the users want to see on their printed pages.

You don't want to be responsible for user frustration, so you should get around these limitations by specifying a style sheet that will be read only by the printer, using this markup:

```
<link rel="stylesheet" href="master-print.css" media="print" />
```

Within this print-only style sheet, you will want to minimize the whitespace that would appear on the printed page, removing unnecessary components, and ensure that no content is chopped off or hidden in the final printed article. For example, you can remove the navigation by applying the following style rule:

```
#navigation {  
    display: none;  
}
```

You may want to specify a font color, such as black, to override all other text colors on the page. This can be achieved with the following style rule:

```
* {  
    color: #000 !important;  
}
```

This applies the color black (in hexadecimal notation, black is #000000 or, in shorthand, #000) to all elements on the page (the * wildcard is used to target all elements on the page). Note the use of the keyword !important at the end of the style rule. As I noted earlier, this is a dangerous keyword to use, as it enforces this style's importance over any other style in the document. However, it is very useful in a printer style sheet, as allows you to override the font color of any element on the page, as specified by a master style sheet, without needing to consider its level of specificity.

Format Your Style Rules

One tried-and-tested layout for style rules that demonstrates good legibility is shown in the following example:

```
.module-heading,  
.module-subheading {  
    background: #333;  
    color: #f00;  
    float: left; /* inline comment */  
}
```

In this style rule, multiple style selectors are used to apply this style rule to page elements with two different CSS classes. Rather than putting the style definitions on the same line, which could be confusing, the style selectors are placed on separate lines. The style rule is opened at the end of the line following the last CSS class name and closed on a separate line after all the style rules. The style rules themselves are indented one tab stop with respect to the style definitions themselves. A space appears between the colon character after each style property name and its respective value.

Each style property exists on its own line, and each line is terminated with a semicolon. If you need to supply a comment to associate with any style rule to aid future development or maintenance, this comment should follow the semicolon on the same line as the rule itself. For consistency in your style rules, you may choose to order your style property names alphabetically, according to the type of style property (text, color, layout, and so on), or by any other grouping that is suitable for your application.

Apply Multiple Class Names to a Single Page Element

As mentioned earlier in this chapter, CSS class names specified within your HTML markup should describe the information that the element's content represents, rather than how that content should be displayed.

You may wish to apply multiple class names to the same element, to avoid duplicating your style rules. Within your markup, this can be achieved simply by separating each CSS class with the space character, as in this example:

```
<div class="article main-article">
  <p>Hello, world.</p>
</div>
```

Style rules written for both `article` and `main-article` classes will be applied to the `<div>` tag, from left to right, so styles specified within the `main-article` style rule will take precedence over those within the `article` style rule.

If you need to apply a special style to elements that contain multiple classes in this way, the following CSS selector syntax will allow you to do just that:

```
div.article.main-article p {
  color: #0f0;
}
```

This style rule applies to `<p>` tags whose parent tag is `<div class="article main-article">`. Note the lack of a space character between the CSS class names.

Reset the Browser's Default Styles

By default, a web browser will have its own set of default style rules to apply to a page without styles. This will include font face, size, line height, and color, as well as padding and margins to differing degrees on different elements and tags.

To provide a level playing field for your own styles, I recommend that you reset the browser's default styles. This passes the precise control over each element to your own style sheets, providing more consistency across different browsers.

At its very simplest, the following style rule levels all margin and padding applied to all page elements down to zero, ready for you to specify your own margin and padding spacing individually for each element that requires it in your style rules:

```
* {
  margin: 0;
  padding: 0;
}
```

CSS style guru Eric Meyer provides a very comprehensive reset style sheet that completely eradicates cross-browser differences between the default style rules (<http://meyerweb.com/eric/tools/css/reset/>). Its current incarnation is shown in Listing 1-1.

Listing 1-1. *Eric Meyer's Universal Cross-Browser Style Reset*

```
/* v1.0 | 20080212 */
html, body, div, span, applet, object, iframe,
h1, h2, h3, h4, h5, h6, p, blockquote, pre,
a, abbr, acronym, address, big, cite, code,
del, dfn, em, font, img, ins, kbd, q, s, samp,
small, strike, strong, sub, sup, tt, var,
b, u, i, center,
dl, dt, dd, ol, ul, li,
fieldset, form, label, legend,
table, caption, tbody, tfoot, thead, tr, th, td {
    margin: 0;
    padding: 0;
    border: 0;
    outline: 0;
    font-size: 100%;
    vertical-align: baseline;
    background: transparent;
}

body {
    line-height: 1;
}

ol, ul {
    list-style: none;
}

blockquote, q {
    quotes: none;
}

blockquote:before, blockquote:after,
q:before, q:after {
    content: '';
    content: none;
}

/* remember to define focus styles! */
:focus {
    outline: 0;
}
```

```

/* remember to highlight inserts somehow! */
ins {
    text-decoration: none;
}

del {
    text-decoration: line-through;
}

/* tables still need cellspacing="0" in the mark-up */
table {
    border-collapse: collapse;
    border-spacing: 0;
}

```

Eric Meyer has spent a lot of time researching and testing CSS support in each major browser, and his reset style is the much-appreciated fruit of this labor.

Whether you choose to use the simple margin/padding reset code shown at the beginning of this section, Eric Meyer's full reset-the-heck-out-of-everything code shown in Listing 1-1, or one of the other alternatives you can find through a quick search of the Web, you should specify these reset styles at the very start of your main site CSS file, before you apply any other style rules.

Master Shorthand Style Rules

Certain styles can be combined or shortened and, where possible, you should use these versions to reduce the size of your CSS files and save on the volume of data downloaded across the wire from the web server to the browser.

As a simple example, certain margin and padding values on an element could either be specified the long-winded way:

```

p {
    margin-top: 5px;
    margin-right: 7px;
    margin-bottom: 5px;
    margin-left: 7px;
    padding-top: 6px;
    padding-right: 3px;
    padding-bottom: 6px;
    padding-left: 3px;
}

```

or in shorthand:

```

p {
    margin: 5px 7px 5px 7px;
    padding: 6px 3px 6px 3px;
}

```

Note that the order of the shorthand values for `margin`, `padding`, and `border` is always as follows:

- Top
- Right
- Bottom
- Left

In the case where the values for `margin-top` and `margin-bottom` are identical, and the values for `margin-left` and `margin-right` are also identical, further shorthand notation may apply, like so:

```
p {  
  margin: 5px 7px; /* 5px top and bottom, 7px left and right */  
  padding: 6px 3px;  
}
```

Other styles—such as `font`, `color`, and `background`—also have their own form of shorthand notation. Chapter 4, which focuses on improving performance in CSS files, provides more details about the shorthand for these styles.

Accessibility Guidelines for Styles

By now, you know that one of our primary concerns as web developers is accessibility—ensuring our content is available to everyone, regardless of browser, device, or input method. Primarily, the focus is on ensuring semantic markup, as discussed earlier in this chapter, but you must ensure that your style sheets back up the principle. Users with displays of different sizes and resolutions should be able to view your page content in a clear manner.

Hide Content from CSS-Capable Browsers

You can use style sheets to visually hide the contents of a particular tag within your markup if you do not want them displayed on the screen. Recall the movie review site example earlier in this chapter, where you wanted to ensure the text was in your HTML, so that if the content were read aloud or read without any style sheets applied, the user would understand the rating the reviewer had assigned to that movie. Now it's time to apply your CSS. You want to show an image representing the equivalent four-out-of-five star rating, rather than show that text.

Hiding text in this way should be accomplished using the `text-indent` style property, choosing a value that positions the text off the left edge of the browser viewport so that it is no longer visible, as follows:

```
div#hide-me {  
  text-indent: -9999px;  
}
```

Using any other style property for hiding the text has the unfortunate side effect of rendering the text unreadable through the popular JAWS screen reader software, manufactured by Freedom Scientific (<http://www.freedomscientific.com/>).

Move Content Blocks to Maintain Correct Markup Source Order

Earlier in this chapter, you learned that the order of blocks of content within your markup should be such that when the page is read aloud, the most relevant page content comes first, and secondary content, including navigation and copyright information, comes last. Of course, this order may not suit the visual appearance of your site, where you most likely wish to have your navigation links appear above the main body of content.

If you have two block elements in sequence within your markup and would rather have the latter content appear in the browser before the former, you can use the style property `float` to reorder those two blocks of content. Look at the following snippet of HTML code:

```
<div id="body">
  ...
</div>

<div id="navigation">
  ...
</div>
```

By default, the `<div id="body">` tag would appear before the `<div id="navigation">` tag, but you can reverse this order with the `float` style property, as follows:

```
#body {
  float: right;
}

#navigation {
  float: left;
}
```

You may also wish to consider using combinations of absolute and relative positioning to relocate content blocks around the page to achieve the layout you need.

Use Relative Font Sizes

Some site visitors, who are perhaps visually impaired or simply have their screen resolution set high, may resize the base font size in their browser to view the site optimally. All the major browsers have this feature. As developers, we must consider the end user's needs in addition to the design criteria for our web pages.

To handle the possibility of font resizing, you should use relative units within your style sheets, so that as the browser's base font size changes, so too does the page text, proportionally. The `em` unit is one such relative unit, as is `%` (percent). You should absolutely not use `px` (pixel) or `pt` (point) values to specify font sizes in your style sheets, as these will not scale according to the end user's need.

To make calculation of font sizes easier when working with relative font size units, try setting the default font size on your `<body>` tag element to a size of `62.5%`. At this level in default browser settings, the size of `1em` becomes visually equivalent to the size of `10px`. Each `0.1em` increment then corresponds to a visible increase of `1px`, so `1.1em` is visually equivalent to `11px` and `1.2em` to `12px`. This trick is extremely useful when working with creative designs that invariably have font sizes measured in pixels. The following are examples of sizes:

```
body {
    font-size: 62.5%; /* resets font sizes so 1em is visually equivalent to 10px */
}

ul li {
    font-size: 1.1em; /* visually equivalent to 11px */
}

h2 {
    font-size: 1.5em; /* visually equivalent to 15px */
}
```

Comment Blocks

To aid with maintenance and to help other developers understand your style sheets, it is smart to group logically related style rules together within your style sheet files. Precede them with a comment describing the purpose of this group's styles, using a consistent notation, such as the following:

```
/* -----
   Form styles

   Group of styles for displaying form controls according to
   the agreed design
*/
```

Inline comments, placed next to style rules themselves, should be included only when necessary to clarify a particular rule. For example, where a style rule has been added to deal with an edge case in a particular browser, it makes sense to note this, so that a future developer does not deem the rule unnecessary and remove it without understanding its original intention.

At the very top of each style sheet file, it may be sensible to include an opening comment section, detailing the author(s) of that file and the purpose of the style rules in the file, and listing each of the logical groups contained within the file in the order in which they occur, using a consistent format. Here is an example of the format of a comment block describing an entire style sheet file and its purpose:

```
/* -----
   Filename      common.css
   Author        Den Odell me@denodell.com
   Description    Basic page layout and default styles for site

   Contents
   - Reset styles
   - Page column layout
   - Form control styles
```

```

Color palette
#777          Medium grey
#aaa          Light grey
#a3d60a       Site-wide green
-----
*/

```

Including such a comment block allows developers to see at a glance whether the file contains the code they are seeking. In the case of your common site-wide style sheet, it may be wise to include a list of some of the key colors used across the site. Developers can then easily find these colors, and copy and paste those values if necessary.

Browser Work-Arounds

If you need to include extra styles to target specific versions of IE, you should use the conditional comments technique described earlier in this chapter to include a separate style sheet file for those styles.

Caution Steer clear of hacks that involve setting style values that are out of the ordinary or use combinations of backslashes, comments, and other odd characters to confuse and bewilder certain browsers. If you cannot avoid the use of a particular hack, be sure to include a clear comment as to why you have chosen to leave it in and what purpose it serves, to warn other developers who may view that style sheet file in future.

When using PNG-24 images in IE 6 or earlier, transparent portions of the image files appear in gray/light blue, rather than as transparent. Unfortunately, Microsoft did not introduce support for transparencies in this file type until the release of IE 7. However, you can use a work-around that enables simple PNG-24 images to display as background or foreground images on the page.

Let's say that in your main style sheet, you have the following style rule defined:

```

#header {
    background:url(my-image.png) no-repeat;
}

```

In your IE 6-specific style sheet file, which you have referenced using conditional comments, you would include something like the following style rule:

```

#header {
    background: transparent none;
    filter:progid:DXImageTransform.Microsoft.AlphaImageLoader(src='my-image.png', ➡
        sizingMethod='scale');
}

```

This IE 6-specific style rule hides the background image specified in the original style sheet and instead uses a Microsoft-specific DirectX filter to reference the same image file. This filter is able to display the transparent portions of the image correctly and places it where

the background image once sat. There are several limitations with this technique, however, including a lack of support for background positioning and repeating, so use it with some caution.

Localization Considerations

One of the most important pieces of future-proofing for a web site is to consider that the site may need to be viewable in alternate languages or in country- or region-specific versions.

Not only will the text content of the web site be different, but there will most likely be some alteration to at least one portion of your style sheets. For example, you may need to replace an image file that contains text embedded in a certain language within it, or some other region-specific style, which could even mean altering the text direction for certain alphabets.

To aid this future-proofing, from the outset, you should aim to include all region-agnostic styles within your main style sheet files and to include a separate style sheet file to provide the locale-specific style rules that override the main style sheet on a per-locale basis. In this way, it becomes easy to add support for new languages and locales, as it is simply a case of creating a new style sheet file from the existing region-specific file and making substitutions to those styles to apply to the new region and/or language.

Structuring Your Folders, Files, and Assets

A simple but expandable folder structure provides a good foundation for the addition of future content and assets, without making the task of maintenance more of a burden to the developer. The following sections provide some guidelines for structuring your folders, files, and assets.

Readable URLs

You should create folders that correlate to the site map of your project, so that HTML pages and server-side scripts are stored in folders that cause their URLs to be sensible, readable, and meaningful. Often, the default or index page within each folder can be assigned within the web server configuration so that URLs can be requested by folder name only, and the index file is loaded by default.

You should also attempt to reflect the hierarchy of your site map similarly with folders, so that URLs read as hierarchy structures also. Some search engines may use text in the URL, as well as page contents, as search criteria.

For example, this URL:

```
http://www.mywebsite.com/news/
```

is neater, readable, and more memorable than this one:

```
http://www.mywebsite.com/news.php
```

Also, the sections of the URL should reflect the site map and a sensible hierarchical structure, such as in the following example:

```
http://www.mywebsite.com/music/rock/
```

File and Folder Naming

When naming folders and files, you should use lowercase alphanumeric characters, with each word separated by a hyphen character for legibility. Steer clear of using characters other than English letters, numbers, or hyphens. When referencing these folders and files from code, make sure the casing is consistently lowercase, as certain web server operating systems, including Unix, Linux, and Mac OS X, are case-sensitive. Also, never use the space character or characters reserved for URLs, including the ampersand, hash character, or the question mark, for naming folders or files, or you may find these cannot be referenced consistently from different web browsers.

One requirement that crops up time and time again is to provide your end users with site content localized into their language. It can be incredibly disruptive to perform this localization task after you have built your site, as it may require you to rewrite or refactor your HTML, CSS, and JavaScript files to support it. You should group together assets that relate to a specific region, language, or locale into their own folders, named after the locale (specified, for example, as language and location, such as en-us for US English). Store nonlocale-specific assets at the same level as this folder structure.

Tip Consider the use of XML files, a content management server, or server-side resource files to provide localized content directly into script-generated markup. This will make maintenance tasks simpler than editing copies of static HTML pages.

File Encoding

The UTF-8 encoding type should be used for all text-based files, as this allows the direct use of extended characters and non-English character sets within the files, without the need for special control characters and ASCII codes.

In practice, it is a lot easier to create files in UTF-8 format from the outset, rather than to translate them later. If you need to convert a file from another encoding format to UTF-8, make a backup of the existing file and use your development environment's file properties dialog box to set the format of the document to the appropriate encoding type. This may affect the encoding of certain characters already within the file, so use the backup file to copy and paste the characters from the backup to the newly UTF-8 encoded file.

In some cases, the byte-order mark (BOM) at the beginning of a UTF-8 encoded file can cause havoc with server-side technologies such as PHP. When encoding files that are to be processed by such technologies, ensure the BOM is removed using the development environment's file properties dialog box, if this option exists.

Organizing Assets

Consider whether placing all static asset files (style sheets, images, scripts, audio, and video), excluding HTML, within a common folder might be worthwhile. This logically separates client-side code from the structure of the site and its content, avoiding clutter at the root level

of the site's document tree. The following directory structure is an example of one that follows this approach, and has also been created with localization in mind:

```
\assets
  \images
    \en-gb
    \fr-fr
  \scripts
    \third-party
  \styles
    \en-gb
    \fr-fr
  \flash
  \documents
    \en-gb
    \fr-fr
  \video
    \en-gb
    \fr-fr
  \audio
    \en-gb
    \fr-fr
```

Image Guidelines

Group together contextually related images within folders with meaningful names, to make them easier to find. These names may correlate to the site map of the project or to their content or usage on the page.

Choose the image format that gives the lowest file size, while still retaining the best quality. Use compression carefully. The desired result is an image practically indistinguishable from the original to the human eye at a distance of about 0.5 meter (about 1.5 feet) from the screen.

The PNG file format often results in smaller file sizes for certain types of images. PNG is a lossless format and allows alpha transparencies for web browsers that provide support for that feature (unfortunately, IE 6 does not, by default, but you can try applying the work-around described earlier in this chapter). However, many image manipulation programs include extra gamma information in the image file. Unfortunately, this gamma information is not interpreted in a similar way across browsers, meaning that odd image artifacts and colors may appear. Fortunately, several tools exist to remove this gamma information from PNG files, without compromising the quality of the final image, which has the added benefit of reducing the file size of the image (since extra information is removed). Smush it (<http://www.smushit.com/>) is one such web-based tool for removing this extra information.

See Chapter 4 for more details on selecting the best image format for the most faithful reproduction of graphics, while ensuring the smallest possible file size.

Multimedia Guidelines

The ubiquitous Adobe Flash Player plug-in should be used to display video until all browsers include their own built-in codecs. Supply a download link to the displayed movie file, preferably in multiple formats, so users without the Flash plug-in may download the movie file for offline viewing.

Use the MP3 storage type for all audio, and harness Adobe's Flash Player to present it to the end user. A download link to the MP3 file should be supplied for users without the Flash plug-in.

Chapter 7 provides more details about including multimedia components in your pages.

Setting Up Your Development Environment

It's also important to consider how you will work on your projects, whether you are working alone or as part of a team. In this section, we'll look at which tools you can leverage to write your files, store your files, and test your pages in a modern, effective way.

Writing Your Files: Integrated Development Environments

Gone are the days of using Notepad for Windows or TextEdit for Mac OS X for writing client-side code for the Web. Modern web sites require a lot of code spread out over multiple files and folders, so an effective way of navigating these folders and editing these files is needed. Desktop application developers have long used integrated development environments (IDEs) to write, maintain, and debug their code. Now the time is right for web developers to follow suit. Each IDE is slightly different, and it really is a matter of personal taste which you find best for your own needs.

One IDE gaining a lot of popularity in the web development community, and my personal favorite, is Aptana Studio (<http://www.aptana.com/>). This IDE is built upon Eclipse, one of the more popular and well-established development environments for Java developers. It allows you to store project files together; supports syntax highlighting of HTML, CSS, and JavaScript files; and even contains a built-in web server for testing your code without needing to deploy to a separate web hosting infrastructure. Files can be synchronized between your computer and an FTP or SFTP server, and Aptana ensures that files aren't overwritten when they shouldn't be when performing a sync. The same plug-in architecture found in Eclipse is supported, so the multitude of extensions and add-ons already developed for that IDE are available for use within Aptana, making it more than just a basic package.

Other IDEs that are popular among developers include the following;

- Notepad++ (<http://notepad-plus.sourceforge.net/>) for Windows systems
- Microsoft Visual Web Developer (<http://www.microsoft.com/express/vwd/>) for Windows systems
- TextMate (<http://macromates.com/>) for Mac OS X systems
- Coda (<http://www.panic.com/coda/>) for Mac OS X systems
- Adobe Dreamweaver (<http://www.adobe.com/products/dreamweaver/>) for both Windows and Mac OS X systems

Investigate these IDEs, try them out, and find which works best for you.

Caution Be wary of any IDE that attempts to write code for you automatically. As a developer, you need to be confident of your own skills and consciously aware of all code that is added to your project.

Storing Your Files: Version Control Systems

One of the guidelines I proposed earlier in this chapter is to regularly purge your folder structure and files for content that is no longer relevant or needed. The result is a tidy, pruned project holding only the code that's needed for the site you are building. This is all well and good, but what if you accidentally delete some code or files you later need? This is where version control systems come into play.

Version control systems store revisions and backups of your files and folders, allowing you to step back in time and recover lost code. Such systems also manage team collaboration, so more than one developer can work on the same file at the same time. The version control system manages the merging of the changes made by each developer into a single file or central storage location for the code.

Subversion (<http://subversion.tigris.org/>) is a popular, open source version control system used in personal and professional development environments. You set up the server part of the system on a web server available via a URL, and the developers use a Subversion client tool to access that web server, taking a local copy of the code to their computer to work on it there. Making a change to a file is as simple as editing the file. When the developers are comfortable that they have completed the feature they were working on, they then “commit” their code back to the server, which manages any merging of files and creation of backup copies.

Subversion appears to be surprisingly simple for developers to use, which is probably why it has been so quickly adopted in the relatively short time it has been around. Of course, the lack of any price tag helps make it accessible to personal users as well as professionals.

You don't need to set up your own server if you want to take advantage of Subversion. Several companies offer services to store your code online, safely backed up each and every time you make a change. One such company is Beanstalk (<http://beanstalkapp.com/>), which offers different pricing models depending on your needs, but also provides a limited free storage plan (up to 20MB at the time of writing), which may be sufficient if you are working on a relatively small project. Google has its own Subversion hosting system known as Google Code (<http://code.google.com/hosting/>), which is free and also contains a wiki and bug-tracking system. Another open source online storage repository is GitHub (<http://github.com/>), which offers various pricing solutions, including a free service for smaller projects. This service uses the GIT technology, rather than Subversion, but the two version control systems are actually very similar in structure and setup.

Once again, investigate these systems and see which one might work best for you. Remember that if you do not use a version control system for your projects, large or small, you run the risk of losing valuable code at any time.

Testing Your Pages: Browsers and Development Tools

Now that you've decided which IDE and version control system you're going to use to work on your files, you need to consider your testing setup. Testing and code validation are vitally

important to your work as a professional web developer. You need to make sure that the sites you build work in all the major desktop web browsers (currently IE 6, 7, and 8; Firefox 2 and 3; Opera 9.5 and up; Safari 3 and up; and Google Chrome) and as many mobile devices you can lay your hands on. Of course, the guidelines I've endorsed in this chapter should mean that your sites work in all of these and more. But the only way to know for sure and to iron out subtle differences in HTML, CSS, and JavaScript implementations is to test everything you write thoroughly in as many different browsers and systems as possible.

Be sure you've installed copies of all the web browsers you can on your machine. Some browsers don't support the running of multiple versions on the same machine, so you may want to consider using virtual machine technology to run a secondary copy of your operating system within memory. For Windows users, Microsoft Virtual PC (<http://www.microsoft.com/windows/products/winfamily/virtualpc/>) is the ideal choice. It is free, and Microsoft even provides regularly updated machine images preinstalled with different versions of IE via the web site, to aid your testing. For Mac OS X users, VMware (<http://www.vmware.com/>) and Parallels (<http://www.parallels.com/>) are your best options.

If you are unable to run such virtualization systems, you might be able to use the services of a browser testing web site, such as BrowserShots (<http://www.browsershots.org/>), which run a URL you supply through various web browsers on different machines and relay the screen grab of the results to you for your comparison. This alternative is certainly not ideal, but it's a good backup or secondary option.

Different browsers have different development tools available for you to prod and poke within the HTML, CSS, and JavaScript output of your pages. Some such tools even allow you to make changes to your code on the fly and see the results immediately. The gold standard is the Firebug plug-in for Mozilla's Firefox (<http://getfirebug.com/>). I've never met a web developer who hasn't sworn by it for investigating which styles have been applied to their page elements and what the values of variables in their JavaScript are at any given time. When errors occur on your page, the Firebug console points these out to you, and shows on which line numbers in your code these errors occurred. If you're debugging JavaScript errors, you can see the list of function calls that occurred before the error took place, which can help you track down the error.

The Internet Explorer Developer Toolbar plug-in is available for IE 6 and 7. This tool pales in comparison to Firebug, but it does allow you to see which styles are applied to which page elements. IE 8 has its own built-in developer tools, which sit somewhere between those of the Developer Toolbar and Firebug. For debugging JavaScript issues within IE, Microsoft's Visual Web Developer IDE is a very useful tool. It is able to hook onto the internal processes that run IE on the Windows machine. When a JavaScript error occurs, Visual Web Developer allows you to see a full list of all the actions and code called prior to the error taking place, including links to where that error occurred. This is helpful for probing variable values and debugging the issue.

Opera, Safari, and Google Chrome have their own built-in developer tools. Most of these do not seem to be fully feature-complete at the time of writing, but they are shaping up to be about as useful as Firebug.

You should download and learn how to use each browser and its associated development tools, so when those elusive browser-specific bugs rear their heads, you are able to quickly and effectively track down the source of the problem and nip it in the bud.

One last word about testing: don't leave it until the last moment. The projects that consistently deliver on time without developer misery are those that have been tested in many

different browsers from the get-go and all the way through the development process, to ensure the final product is the best, most compatible web site possible.

Summary

This chapter discussed the importance of using tried-and-tested current ideas, known as best practices, and how to use discernment when selecting which guidelines to apply to your own projects. We have gone through HTML, CSS, file structures, and development environments, reviewing smart, modern, and effective techniques for writing and maintaining code. You now know that the whole purpose behind using these best practices is to ensure the pages you create are fully accessible by anyone, regardless of browser, device, or input method. You also want to make sure that the code you write can be easily read, understood, and updated in the future.

The next chapter guides you through JavaScript best practices. It covers how to structure your client-side code in such a way that it is similarly easy to read, understand, and update, regardless of the size of the code base. This will help you to build effective RIAs, based on solid, scalable code rules.

