CHAPTER 1
Understanding How the Web Works

What You'll Learn in This Chapter:

- A very brief history of the World Wide Web
- What is meant by the term *web page*, and why that term doesn't always reflect all the content involved
- How content gets from your personal computer to someone else's web browser
- How to select a web hosting provider
- How different web browsers and device types can affect your content
- How to transfer files to your web server using FTP
- Where files should be placed on a web server
- How to distribute web content without a web server

Before you learn the intricacies of HTML (Hypertext Markup Language), CSS (Cascading Style Sheets), and JavaScript—not to mention the back-end programming language PHP—it is important to gain a solid understanding of the technologies that help transform these plain-text files to the rich multimedia displays you see on your computer, tablet, or smartphone when browsing the World Wide Web.

For example, a file containing markup and client-side code (HTML, CSS, and JavaScript) is useless without a web browser to view it, and no one besides yourself will see your content unless a web server is involved—this is especially true when server-side technologies such as PHP are put into the mix. Web servers make your content available to others who, in turn, use their web browsers to navigate to an address and wait for the server to send information to them. You will be intimately involved in this publishing process because you must create files and then put them on a web server to make the content available in the first place, and you must ensure that your content will appear to the end user as you intended.

A Brief History of HTML and the World Wide Web

Once upon a time, back when there weren't any footprints on the moon, some farsighted folks decided to see whether they could connect several major computer networks. I'll spare you the names and stories (there are plenty of both), but the eventual result was the "mother of all networks," which we call the Internet.

Until 1990, accessing information through the Internet was a rather technical affair. It was so hard, in fact, that even Ph.D.-holding physicists were often frustrated when trying to exchange data and documents. One such physicist, the now-famous (and knighted) Sir Tim Berners-Lee, cooked up a way to easily cross-reference text on the Internet through hypertext links.

This wasn't a new idea, but his simple Hypertext Markup Language (HTML) managed to thrive while more ambitious hypertext projects floundered. *Hypertext* originally meant text stored in electronic form with cross-reference links between pages. It is now a broader term that refers to just about any object (text, images, files, and so on) that can be linked to other objects. *Hypertext Markup Language* is a language for describing how text, graphics, and files containing other information are organized and linked.

By 1993, only 100 or so computers throughout the world were equipped to serve up HTML pages. Those interlinked pages were dubbed the *World Wide Web (WWW)*, and several web browser programs had been written to enable people to view web content. Because of the growing popularity of the Web, a few programmers soon wrote web browsers that could view graphical images along with text. From that point forward, the continued development of web browser software and the standardization of web technologies including HTML, CSS, and JavaScript have led us to the world we live in today, one in which more than a billion websites serve trillions (or more) of text and multimedia files.

NOTE

For more information on the history of the World Wide Web, see the Wikipedia article on this topic: http://en.wikipedia.org/wiki/History_of_the_Web.

These few paragraphs really are a brief history of what has been a remarkable period. Today's college students have never known a time in which the World Wide Web didn't exist, and the idea of always-on information and ubiquitous computing will shape all aspects of our lives moving forward. Instead of seeing dynamic web content creation and management as a set of skills possessed by only a few technically oriented folks (okay, call them geeks, if you will), by the end of this book, you will see that these are skills that anyone can master, regardless of inherent geekiness.

Creating Web Content

You might have noticed the use of the term *web content* rather than *web pages*—that was intentional. Although we talk of "visiting a web page," what we really mean is something like "looking at all the text and the images at one address on our computer." The text that we read and the images that we see are rendered by our web browsers, which are given certain instructions found in individual files.

Those files contain text that is *marked up* with, or surrounded by, HTML codes that tell the browser how to display the text—as a heading, as a paragraph, in a bulleted list, and so on. Some HTML markup tells the browser to display an image or video file rather than plain text, which brings me back to this point: Different types of content are sent to your web browser, so simply saying *web page* doesn't begin to cover it. Here we use the term *web content* instead, to cover the full range of text, image, audio, video, and other media found online.

In later chapters, you'll learn the basics of linking to or creating the various types of multimedia web content found in websites, and for creating dynamic content from server-side scripts using PHP. All you need to remember at this point is that *you* are in control of the content a user sees when visiting your website. Beginning with the file that contains text to display or code that tells the server to send a graphic along to the user's web browser, you have to plan, design, and implement all the pieces that will eventually make up your web presence. As you will learn throughout this book, it is not a difficult process as long as you understand all the little steps along the way.

In its most fundamental form, web content begins with a simple text file containing HTML markup. In this book, you'll learn about and compose standards-compliant HTML5 markup. One of the many benefits of writing standards-compliant code is that, in the future, you will not have to worry about having to go back to your code to fundamentally alter it so that it works on multiple types of browsers and devices. Instead, your code will (likely) always work as intended for as long as web browsers adhere to standards and the backwards compatibility to previous standards (which is hopefully a long time).

Understanding Web Content Delivery

Several processes occur, in many different locations, to eventually produce web content that you can see. These processes occur very quickly—on the order of milliseconds—and happen behind the scenes. In other words, although we might think all we are doing is opening a web browser, typing in a web address, and instantaneously seeing the content we requested,

technology in the background is working hard on our behalf. Figure 1.1 shows the basic interaction between a browser and a server.

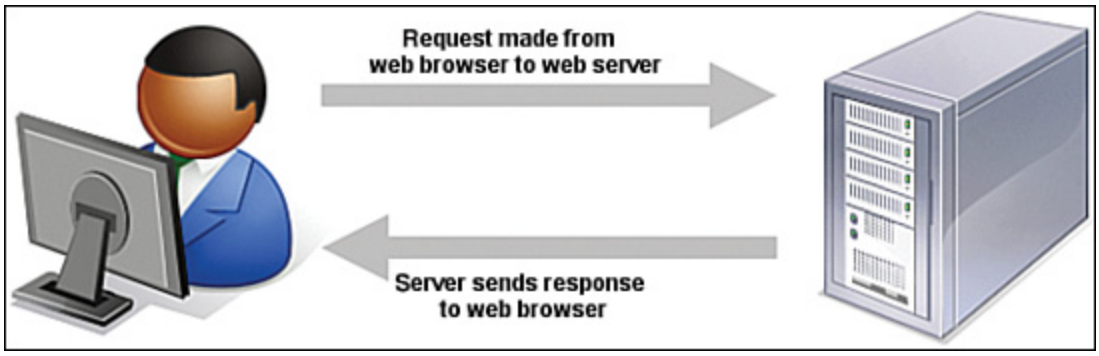

FIGURE 1.1
A browser request and a server response.

However, the process involves several steps—and potentially several trips between the browser and the server—before you see the entire content of the site you requested.

Suppose you want to do a Google search, so you dutifully type **www.google.com** in the address bar or select the Google bookmark from your bookmarks list. Almost immediately, your browser shows you something like what's shown in Figure 1.2.

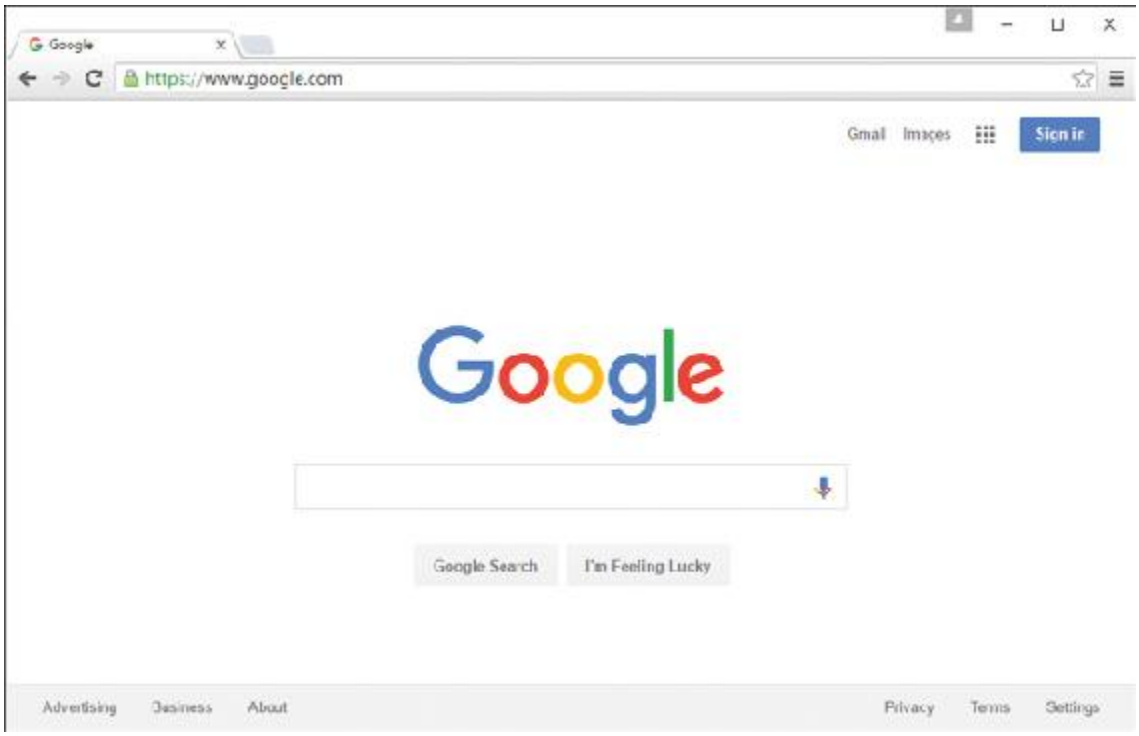

FIGURE 1.2
Visiting www.google.com.

Figure 1.2 shows a website that contains text plus one image (the Google logo). A simple version of the processes that occurred to retrieve that text and image from a web server and display it on your screen follows:

1. Your web browser sends a request for the `index.html` file located at the http://www.google.com address. The `index.html` file does not have to be part of the address that you type in the address bar; you'll learn more about the `index.html` file farther along in this chapter.
2. After receiving the request for a specific file, the web server process looks in its directory contents for the specific file, opens it, and sends the content of that file back to your web browser.
3. The web browser receives the content of the `index.html` file, which is text marked up with HTML codes, and renders the content based on these HTML codes. While rendering the content, the browser happens upon the HTML code for the Google logo, which you can see in Figure 1.2. The HTML code looks something like this:

Click here to view code image

```
<img alt="Google" height="92" width="272" id="hplogo"
src="/images/branding/
googlelogo/2x/googlelogo_color_272x92dp.png">
```

The HTML code for the image is an `<img>` tag, and it also provides attributes that tell the browser the file source location (`src`), width (`width`), and height (`height`) necessary to display the logo. You'll learn more about attributes throughout later lessons.

4. The browser looks at the `src` attribute in the `<img>` tag to find the source location. In this case, the image `googlelogo_color_272x92dp.png` can be found in a subdirectory of the `images` directory at the same web address (www.google.com) from which the browser retrieved the HTML file.
5. The browser requests the file at the web address http://www.google.com/images/branding/googlelogo/2x/googlelogo_color_272x92dp.png.
6. The web server interprets that request, finds the file, and sends the contents of that file to the web browser that requested it.
7. The web browser displays the image on your monitor.

As you can see in the description of the web content delivery process, web browsers do more than simply act as picture frames through which you can view content. Browsers assemble the web content components and arrange those parts according to the HTML commands in the file.

You can also view web content locally, or on your own hard drive, without the need for a web server. The process of content retrieval and display is the same as the process listed in the previous steps, in that a browser looks for and interprets the codes and content of an HTML file, but the trip is shorter: The browser looks for files on your own computer's hard drive rather than on a remote machine. A web server would be needed to interpret any server-based programming language embedded in the files, but that is outside the scope of this book. In fact, you could work through all the HTML, CSS, and JavaScript lessons in this book without having a web server to call your own, but then nobody but you could view your masterpieces.

Selecting a Web Hosting Provider

Despite my just telling you that you can work through all the HTML, CSS, and JavaScript lessons in this book without having a web server, having a web server is the recommended method for continuing. Although the appendixes describe how to install a full-blown web server and database on your local machine for personal development, invariably you will want your static or dynamic websites to be visible to the public. Don't worry—obtaining a hosting provider is usually a quick, painless, and relatively inexpensive process. In fact, you can get your own domain name and a year of web hosting for just slightly more than the cost of the book you are reading now.

If you type **web hosting provider** in your search engine of choice, you will get millions of hits and an endless list of sponsored search results (also known as *ads*). Not this many web hosting providers exist in the world, although it might seem otherwise. Even if you are looking at a managed list of hosting providers, it can be overwhelming—especially if all you are looking for is a place to host a simple website for yourself or your company or organization.

You'll want to narrow your search when looking for a provider and choose one that best meets your needs. Some selection criteria for a web hosting provider follow:

- **Reliability/server "uptime"**—If you have an online presence, you want to make sure people can actually get there consistently.
- **Customer service**—Look for multiple methods for contacting customer service (phone, email, chat), as well as online documentation for common issues.
- **Server space**—Does the hosting package include enough server space to hold all the multimedia files (images, audio, video) you plan to include in your website (if any)?
- **Bandwidth**—Does the hosting package include enough bandwidth that all the people visiting your site and downloading files can do so without your having to pay extra?

- **Domain name purchase and management**—Does the package include a custom domain name, or must you purchase and maintain your domain name separately from your hosting account?
- **Price**—Do not overpay for hosting. If you see a wide range of prices offered, you should immediately wonder, "What's the difference?" Often the difference has little to do with the quality of the service and everything to do with company overhead and what the company thinks it can get away with charging people. A good rule of thumb is that if you are paying more than $75 per year for a basic hosting package and domain name, you are probably paying too much.

Here are three reliable web hosting providers whose basic packages contain plenty of server space and bandwidth (as well as domain names and extra benefits) at a relatively low cost. If you don't go with any of these web hosting providers, you can at least use their basic package descriptions as a guideline as you shop around.

NOTE

The author has used all these providers (and then some) over the years and has no problem recommending any of them; predominantly, she uses DailyRazor as a web hosting provider, especially for advanced development environments.

- **A Small Orange (http://www.asmallorange.com)**—The Tiny and Small hosting packages are perfect starting places for any new web content publisher.
- **DailyRazor (http://www.dailyrazor.com)**—Even its personal-sized hosting package is full-featured and reliable.
- **Lunarpages (http://www.lunarpages.com)**—The Starter hosting package is suitable for many personal and small business websites.

One feature of a good hosting provider is that it offers a "control panel" for you to manage aspects of your account. Figure 1.3 shows the control panel for my own hosting account at DailyRazor. Many web hosting providers offer this particular control panel software, or some control panel that is

similar in design—clearly labeled icons leading to tasks you can perform to configure and manage your account.

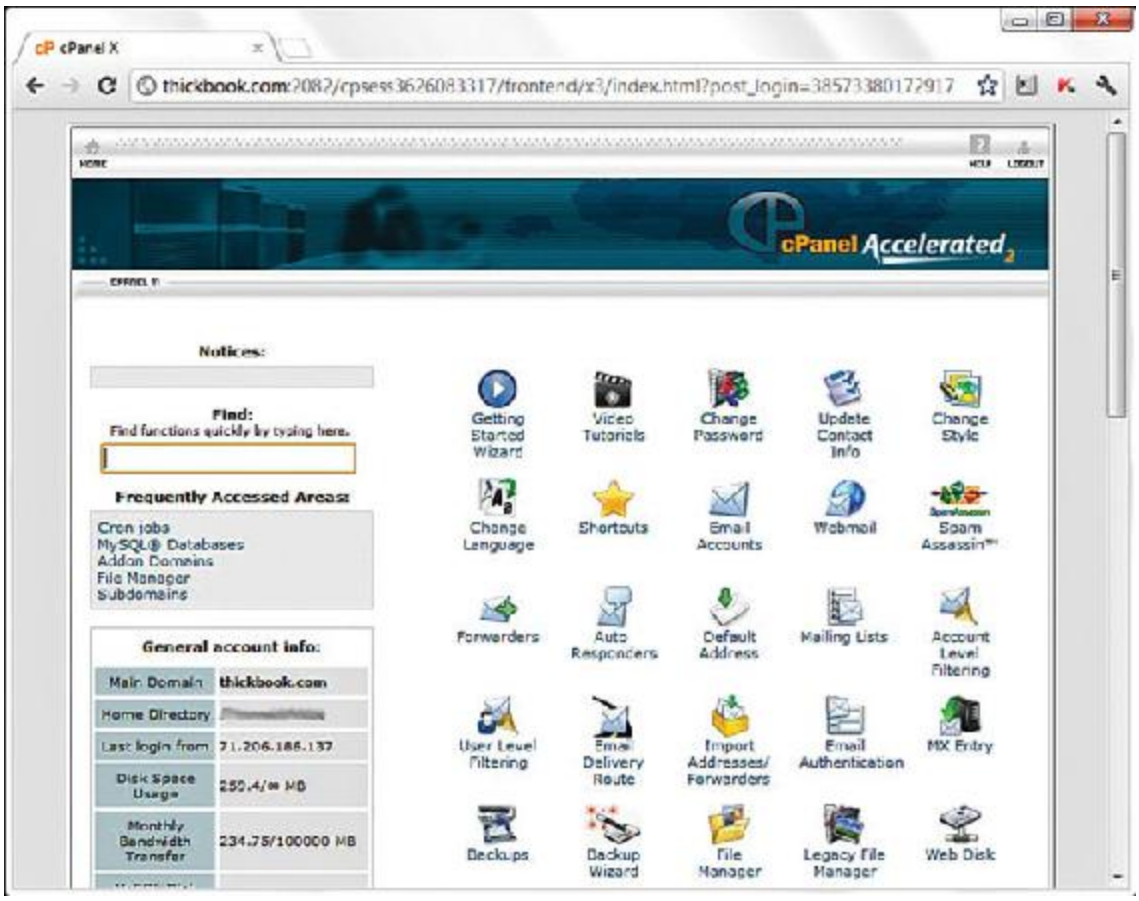

FIGURE 1.3
A sample control panel.

You might never need to use your control panel, but having it available to you simplifies the installation of databases and other software, the viewing of web statistics, and the addition of email addresses (among many other features). If you can follow instructions, you can manage your own web server—no special training required.

Testing with Multiple Web Browsers

Now that we've just discussed the process of web content delivery and the acquisition of a web server, it might seem a little strange to step back and talk about testing your websites with multiple web browsers. However,

before you go off and learn all about creating websites with HTML and CSS, do so with this very important statement in mind: Every visitor to your website will potentially use hardware and software configurations that are different from your own. From their device types (desktop, laptop, tablet, smartphone) to their screen resolutions, browser types, browser window sizes, and speed of connections—you cannot control any aspect of what your visitors use when they view your site. So just as you're setting up your web hosting environment and getting ready to work, think about downloading several web browsers so that you have a local test suite of tools available to you. Let me explain why this is important.

Although all web browsers process and handle information in the same general way, some specific differences among them result in things not always looking the same in different browsers. Even users of the same version of the same web browser can alter how a page appears by choosing different display options and/or changing the size of their viewing windows. All the major web browsers allow users to override the background and fonts the web page author specifies with those of their own choosing. Screen resolution, window size, and optional toolbars can also change how much of a page someone sees when it first appears on the screen. You can ensure only that you write standards-compliant HTML and CSS.

NOTE

In Chapter 3, "Understanding the CSS Box Model and Positioning," you'll learn a little bit about the concept of responsive web design, in which the design of a site shifts and changes automatically depending on the user's behavior and viewing environment (screen size, device, and so on).

Do not, under any circumstances, spend hours on end-designing something that looks perfect only on your own computer—unless you are willing to be disappointed when you look at it on your friend's computer, on a computer in the local library, or on your iPhone.

You should always test your websites with as many of these web browsers as possible, on standard, portable, and mobile devices:

- Apple Safari (http://www.apple.com/safari/) for Mac
- Google Chrome (http://www.google.com/chrome) for Mac, Windows, and Linux/UNIX
- Microsoft Internet Explorer (http://www.microsoft.com/ie) and Microsoft Edge (https://www.microsoft.com/microsoft-edge) for Windows
- Mozilla Firefox (http://www.mozilla.com/firefox/) for Mac, Windows, and Linux/UNIX

Now that you have a development environment set up, or at least some idea of the type you'd like to set up in the future, let's move on to creating a test file.

Creating a Sample File

Before we begin, take a look at Listing 1.1. This listing represents a simple piece of web content—a few lines of HTML that print "`Hello World! Welcome to My Web Server.`" in large, bold letters on two lines centered within the browser window. You'll learn more about the HTML and CSS used within this file as you move forward in this book.

LISTING 1.1 Our Sample HTML File

Click here to view code image

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello World!</title>
  </head>
  <body>
    <h1 style="text-align: center">Hello World!<br>
    Welcome to My Web Server.</h1>
  </body>
</html>
```

To make use of this content, open a text editor of your choice, such as Notepad (on Windows) or TextEdit (on a Mac). Do not use WordPad, Microsoft Word, or other full-featured word processing software because

those programs create different sorts of files from the plain-text files we use for web content.

Type the content that you see in Listing 1.1 and then save the file using `sample.html` as the filename. Be sure your editor does not change the extension you give it; the `.html` extension tells the web server that your file is, indeed, full of HTML. When the file contents are sent to the web browser that requests it, the browser will also know from it is HTML and will render it appropriately.

Now that you have a sample HTML file to use—and hopefully somewhere to put it, such as a web hosting account—let's get to publishing your web content.

Using FTP to Transfer Files

As you've learned so far, you have to put your web content on a web server to make it accessible to others. This process typically occurs by using the *File Transfer Protocol (FTP)*. To use FTP, you need an FTP client—a program used to transfer files from your computer to a web server.

FTP clients require three pieces of information to connect to your web server; this information will have been sent to you by your hosting provider after you set up your account:

- The hostname, or address, to which you will connect
- Your account username
- Your account password

When you have this information, you are ready to use an FTP client to transfer content to your web server.

Selecting an FTP Client

Regardless of the FTP client you use, FTP clients generally use the same type of interface. Figure 1.4 shows an example of FireFTP, which is an FTP

client used with the Firefox web browser. The directory listing of the local machine (your computer) appears on the left of your screen, and the directory listing of the remote machine (the web server) appears on the right. Typically, you will see right-arrow and left-arrow buttons, as shown in Figure 1.4. The right arrow sends selected files from your computer to your web server; the left arrow sends files from the web server to your computer. Many FTP clients also enable you to simply select files and then drag and drop those files to the target machines.

Many FTP clients are freely available to you, but you can also transfer files via the web-based file management tool that is likely part of your web server's control panel. However, that method of file transfer typically introduces more steps into the process and isn't nearly as streamlined (or simple) as the process of installing an FTP client on your own machine.

Here are some popular free FTP clients:

- Classic FTP (http://www.nchsoftware.com/classic/) for Mac and Windows
- Cyberduck (http://cyberduck.ch) for Mac
- Fetch (http://fetchsoftworks.com) for Mac
- FileZilla (http://filezilla-project.org) for all platforms
- FireFTP (http://fireftp.mozdev.org) Firefox extension for all platforms

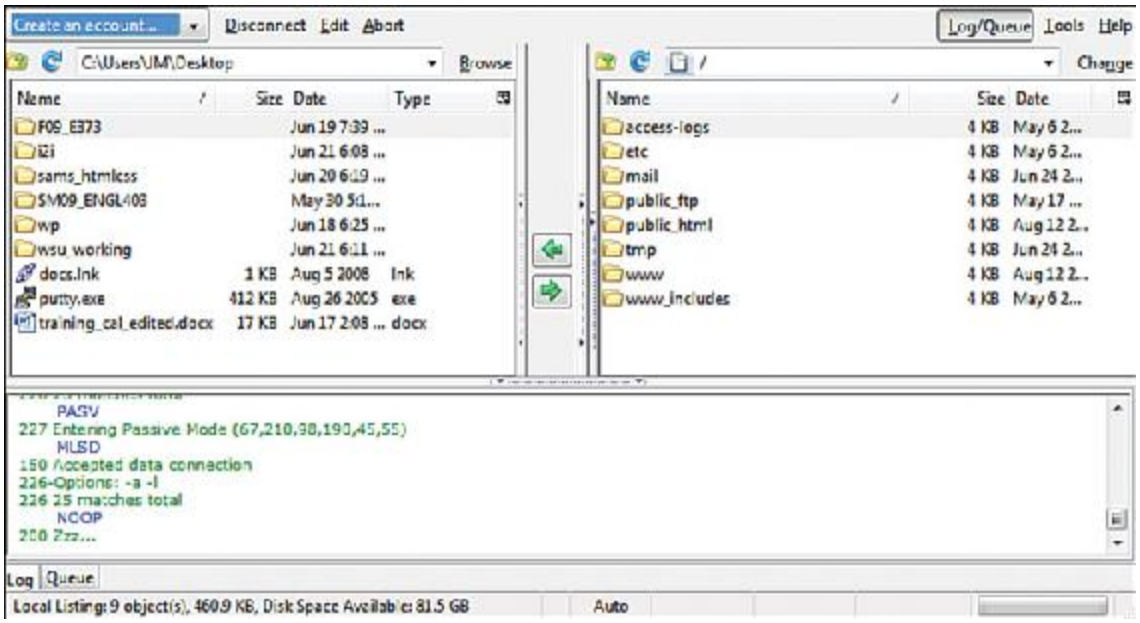

FIGURE 1.4
The FireFTP interface.

When you have selected an FTP client and installed it on your computer, you are ready to upload and download files from your web server. In the next section, you'll see how this process works using the sample file in Listing 1.1.

Using an FTP Client

The following steps show how to use Classic FTP to connect to your web server and transfer a file. However, all FTP clients use similar, if not identical, interfaces. If you understand the following steps, you should be able to use any FTP client. (Remember, you first need the hostname, the account username, and the account password.)

1. Start the Classic FTP program and click the Connect button. You are prompted to fill out information for the site to which you want to connect, as shown in Figure 1.5.
2. Fill in each of the items shown in Figure 1.5 as described here:
    - The FTP Server is the FTP address of the web server to which you need to send your content. Your hosting provider will have given

you this address. It probably is *yourdomain*.com, but check the information you received when you signed up for service.

- Complete the User Name field and the Password field using the information your hosting provider gave you.

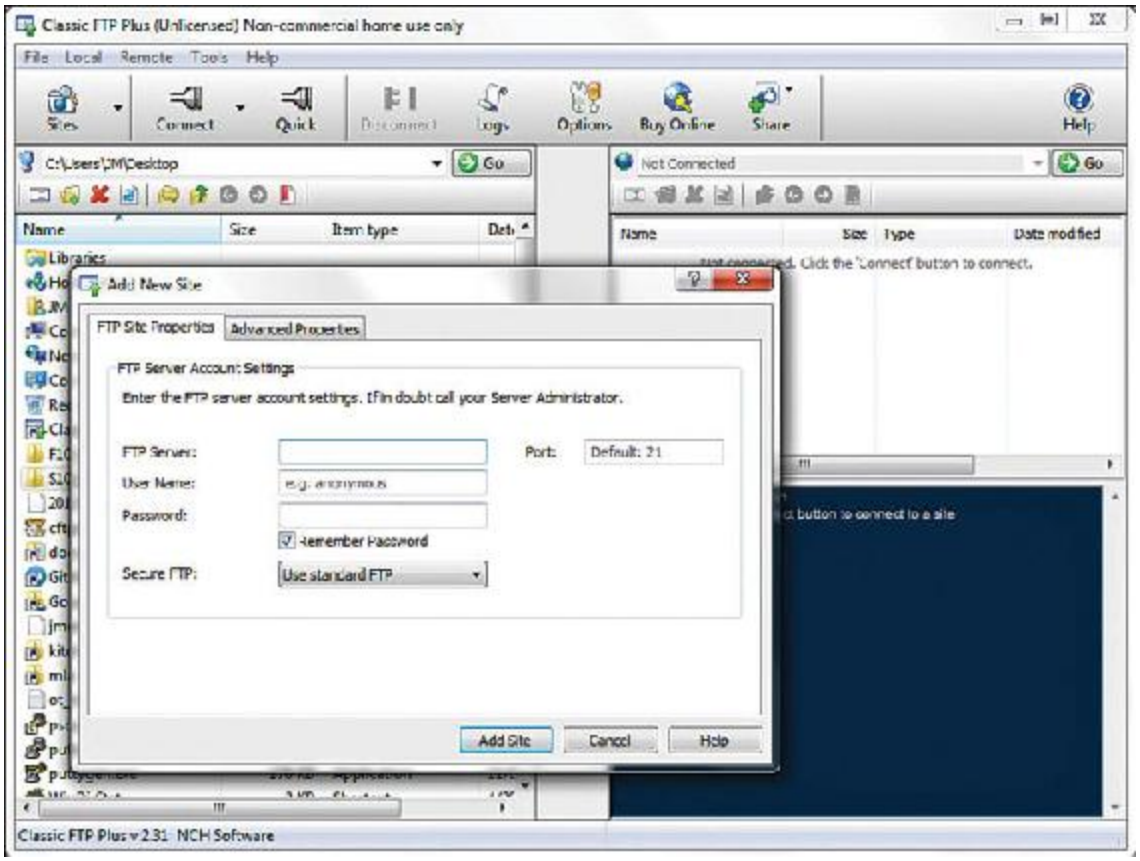

FIGURE 1.5
Connecting to a new site in Classic FTP.

3. You can switch to the Advanced tab and modify the following optional items, shown in Figure 1.6:

   - The Site Label is the name you'll use to refer to your own site. Nobody else will see this name, so enter whatever you want.
   - You can change the values for Initial Remote Directory on First Connection and Initial Local Directory on First Connection, but you might want to wait until you have become accustomed to using the client and have established a workflow.

4. When you're finished with the settings, click Add Site to save them. You can then click Connect to establish a connection with the web

server.

You will see a dialog box indicating that Classic FTP is attempting to connect to the web server. Upon a successful connection, you will see an interface like the one in Figure 1.7, showing the contents of the local directory on the left and the contents of your web server on the right.

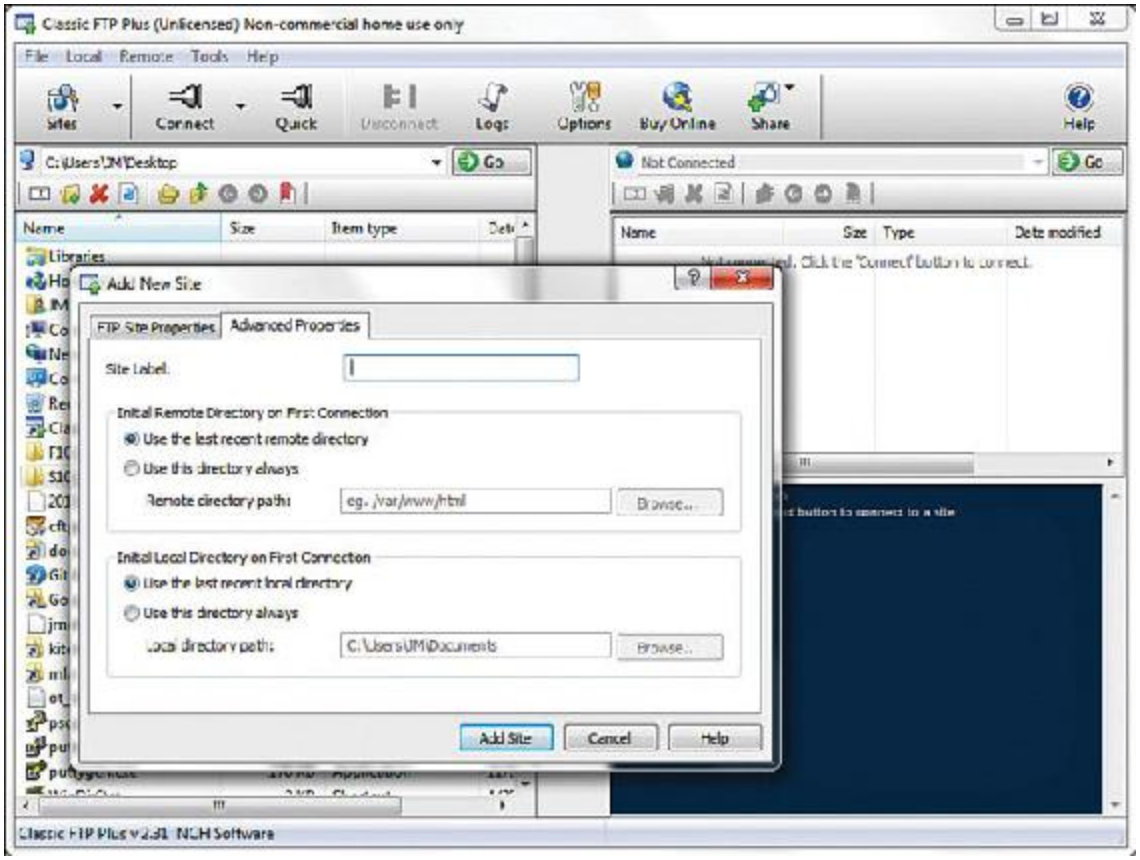

FIGURE 1.6
The advanced connection options in Classic FTP.

5. You are now *almost* ready to transfer files to your web server. All that remains is to change directories to what is called the *document root* of your web server. The document root of your web server is the directory that is designated as the top-level directory for your web content—the starting point of the directory structure, which you'll learn more about later in this chapter. Often, this directory is named `public_html`, `www` (because `www` has been created as an alias for `public_html`), or `htdocs`. You do not have to create this directory; your hosting provider will have created it for you.

Double-click the document root directory name to open it. The display shown on the right of the FTP client interface changes to show the contents of this directory (it will probably be empty at this point, unless your web hosting provider has put placeholder files in that directory on your behalf).

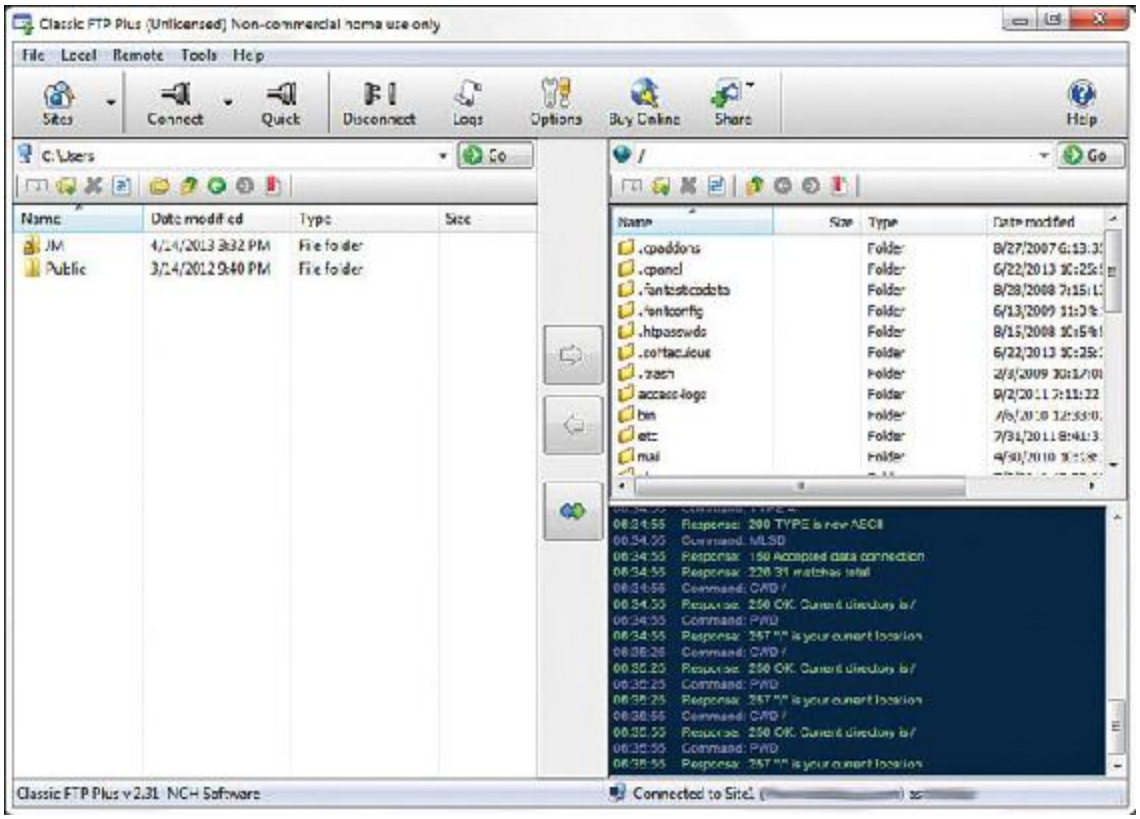

FIGURE 1.7
A successful connection to a remote web server via Classic FTP.

6. The goal is to transfer the `sample.html` file you created earlier from your computer to the web server. Find the file in the directory listing on the left of the FTP client interface (navigate if you have to) and click it once to highlight the filename.
7. Click the right-arrow button in the middle of the client interface to send the file to the web server. When the file transfer completes, the right side of the client interface refreshes to show you that the file has made it to its destination.
8. Click the Disconnect button to close the connection, and then exit the Classic FTP program.

These steps are conceptually similar to the steps you take anytime you want to send files to your web server via FTP. You can also use your FTP client to create subdirectories on the remote web server. To create a subdirectory using Classic FTP, click the Remote menu and then click New Folder. Different FTP clients have different interface options to achieve the same goal.

Understanding Where to Place Files on the Web Server

An important aspect of maintaining web content is determining how you will organize that content—not only for the user to find, but also for you to maintain on your server. Putting files in directories helps you manage those files.

Naming and organizing directories on your web server, and developing rules for file maintenance, is completely up to you. However, maintaining a well-organized server makes your management of its content more efficient in the long run.

Basic File Management

As you browse the Web, you might have noticed that URLs change as you navigate through websites. For instance, if you're looking at a company's website and you click on graphical navigation leading to the company's products or services, the URL probably changes from

http://www.*companyname*.com/

to

http://www.*companyname*.com/products/

or

http://www.*companyname*.com/services/

In the preceding section, I used the term *document root* without really explaining what that is all about. The document root of a web server is essentially the trailing slash in the full URL. For instance, if your domain is *yourdomain*.com and your URL is http://www.*yourdomain*.com/, the document root is the directory represented by the trailing slash (/). The document root is the starting point of the directory structure you create on your web server; it is the place where the web server begins looking for files requested by the web browser.

If you put the `sample.html` file in your document root as previously directed, you will be able to access it via a web browser at the following URL:

http://www.*yourdomain*.com/sample.html

If you entered this URL into your web browser, you would see the rendered `sample.html` file, as shown in Figure 1.8.

However, if you created a new directory within the document root and put the `sample.html` file in that directory, the file would be accessed at this URL:

http://www.*yourdomain*.com/newdirectory/sample.html

FIGURE 1.8
The `sample.html` file accessed via a web browser.

If you put the `sample.html` file in the directory you originally saw upon connecting to your server—that is, you did *not* change directories and place the file in the document root—the `sample.html` file would not be accessible from your web server at any URL. The file will still be on the machine that you know as your web server, but because the file is not in the document root—where the server software knows to start looking for files—it will never be accessible to anyone via a web browser.

The bottom line? Always navigate to the document root of your web server before you start transferring files.

This is especially true with graphics and other multimedia files. A common directory on web servers is called `images`, where, as you can imagine, all the image assets are placed for retrieval. Other popular directories include `css` for style sheet files (if you are using more than one) and `js` for JavaScript files. Alternatively, if you know that you will have an area on your website where visitors can download many types of files, you might simply call that directory `downloads`.

Whether it's a ZIP file containing your art portfolio or an Excel spreadsheet with sales numbers, it's often useful to publish a file on the Internet that isn't simply a web page. To make available on the Web a file that isn't an HTML file, just upload the file to your website as if it *were* an HTML file, following the instructions earlier in this chapter for uploading. After the file is uploaded to the web server, you can create a link to it (as you'll learn in Chapter 2, "Structuring HTML and Using Cascading Style Sheets"). In other words, your web server can serve much more than HTML.

Here's a sample of the HTML code you will learn more about later in this book. The following code would be used for a file named `artfolio.zip`, located in the `downloads` directory of your website, and with link text that reads `Download my art portfolio!`:

Click here to view code image

```
<a href="/downloads/artfolio.zip">Download my art portfolio!</a>
```

Using an Index Page

When you think of an index, you probably think of the section in the back of a book that tells you where to look for various keywords and topics. The index file in a web server directory can serve that purpose—if you design it that way. In fact, that's where the name originates.

The `index.html` file (or just *index file*, as it's usually referred to) is the name you give to the page you want people to see as the default file when they navigate to a specific directory in your website.

Another function of the index page is that users who visit a directory on your site that has an index page but who do not specify that page will still land on the main page for that section of your site—or for the site itself.

For instance, you can type either of the following URLs and land on Apple's iPhone informational page:

http://www.apple.com/iphone/

http://www.apple.com/iphone/index.html

Had there been no `index.html` page in the `iphone` directory, the results would depend on the configuration of the web server. If the server is configured to disallow directory browsing, the user would have seen a "Directory Listing Denied" message when attempting to access the URL without a specified page name. However, if the server is configured to allow directory browsing, the user would have seen a list of the files in that directory.

Your hosting provider will already have determined these server configuration options. If your hosting provider enables you to modify server settings via a control panel, you can change these settings so that your server responds to requests based on your own requirements.

Not only is the index file used in subdirectories, but it's used in the top-level directory (or document root) of your website as well. The first page of your website—or *home page* or *main page*, or however you like to refer to the web content you want users to see when they first visit your domain—should be named `index.html` and placed in the document root of your web server. This ensures that when users type **http://www.*yourdomain*.com/** into their web browsers, the server responds with the content you intended them to see (instead of "Directory Listing Denied" or some other unintended consequence).

Summary

This chapter introduced you to the concept of using HTML to mark up text files to produce web content. You also learned that there is more to web content than just the "page"—web content also includes image, audio, and video files. All this content lives on a web server—a remote machine often far from your own computer. On your computer or other device, you use a web browser to request, retrieve, and eventually display web content on your screen.

You learned the criteria to consider when determining whether a web hosting provider fits your needs. After you have selected a web hosting provider, you can begin to transfer files to your web server, which you also learned how to do, using an FTP client. You also learned a bit about web

server directory structures and file management, as well as the very important purpose of the `index.html` file in a given web server directory. In addition, you learned that you can distribute web content on removable media, and you learned how to go about structuring the files and directories to achieve the goal of viewing content without using a remote web server.

Finally, you learned the importance of testing your work in multiple browsers after you've placed it on a web server. Writing valid, standards-compliant HTML and CSS helps ensure that your site looks reasonably similar for all visitors, but you still shouldn't design without receiving input from potential users outside your development team—it is even *more* important to get input from others when you are a design team of one!

Q&A

Q. I've looked at the HTML source of some web sites on the Internet, and it looks frighteningly difficult to learn. Do I have to think like a computer programmer to learn this stuff?

A. Although complex HTML pages can indeed look daunting, learning HTML is much easier than learning actual programming languages—we're saving that for later on in this book, and you'll do just fine with it as well. HTML is a markup language rather than a programming language; you mark up text so that the browser can render the text a certain way. That's a completely different set of thought processes than developing a computer program. You really don't need any experience or skill as a computer programmer to be a successful web content author.

One of the reasons the HTML behind many commercial websites looks complicated is that it was likely created by a visual web design tool—a "what you see is what you get" (WYSIWYG) editor that uses whatever markup its software developer told it to use in certain circumstances—as opposed to being hand-coded (where you are completely in control of the resulting markup). In this book, you are taught fundamental coding from the ground up, which typically results in clean, easy-to-

read source code. Visual web design tools have a knack for making code difficult to read and for producing code that is convoluted and not standards compliant.

Q. Running all the tests you recommend would take longer than creating my pages! Can't I get away with less testing?

A. If your pages aren't intended to make money or provide an important service, it's probably not a big deal if they look funny to some users or produce errors once in a while. In that case, just test each page with a couple different browsers and call it a day. However, if you need to project a professional image, there is no substitute for rigorous testing.

Q. Seriously, who cares how I organize my web content?

A. Believe it or not, the organization of your web content does matter to search engines and potential visitors to your site. But overall, having an organized web server directory structure helps you keep track of content that you are likely to update frequently. For instance, if you have a dedicated directory for images or multimedia, you know exactly where to look for a file you want to update—no need to hunt through directories containing other content.

Workshop

The Workshop contains quiz questions and exercises to help you solidify your understanding of the material covered. Try to answer all questions before looking at the "Answers" section that follows.

Quiz

1. How many files would you need to store on a web server to produce a single web page with some text and two images on it?
2. What are some of the features to look for in a web hosting provider?
3. What three pieces of information do you need in order to connect to your web server via FTP?

4. What is the purpose of the `index.html` file?
5. Does your website have to include a directory structure?

Answers

1. You would need three: one for the web page itself, which includes the text and the HTML markup, and one for each of the two images.
2. Look for reliability, customer service, web space and bandwidth, domain name service, site-management extras, and price.
3. You need the hostname, your account username, and your account password.
4. The `index.html` file is typically the default file for a directory within a web server. It enables users to access http://www.*yourdomain*.com/*somedirectory*/ without using a trailing filename and still end up in the appropriate place.
5. No. Using a directory structure for file organization is completely up to you, although using one is highly recommended because it simplifies content maintenance.

Exercises

- Get your web hosting in order—are you going to move through the lessons in this book by viewing files locally on your own computer, or are you going to use a web hosting provider? Note that most web hosting providers will have you up and running the same day you purchase your hosting plan.
- If you are using an external hosting provider, then using your FTP client, create a subdirectory within the document root of your website. Paste the contents of the `sample.html` file into another file named `index.html`, change the text between the `<title>` and `</title>` tags to something new, and change the text between the `<h1>` and `</h1>` tags to something new. Save the file and upload it to the new subdirectory. Use your web browser to navigate to the new directory on your web server and see that the content in the

`index.html` file appears. Then, using your FTP client, delete the `index.html` file from the remote subdirectory. Return to that URL with your web browser, reload the page, and see how the server responds without the `index.html` file in place.

- Using the same set of files created in the preceding exercise, place these files on a removable media device such as a USB drive. Use your browser to navigate this local version of your sample website, and think about the instructions you would have to distribute with this removable media so that others could use it.

CHAPTER 2

Structuring HTML and Using Cascading Style Sheets

What You'll Learn in This Chapter:

- How to create a simple web page in HTML
- How to include the HTML tags that every web page must have
- How to use links within your web pages
- How to organize a page with paragraphs and line breaks
- How to organize your content with headings
- How to use the semantic elements of HTML5
- How to begin using basic CSS

In the first chapter, you got a basic idea of the process behind creating web content and viewing it online (or locally, if you do not yet have a web hosting provider). In this chapter, we get down to the business of explaining the various elements that must appear in an HTML file so that it is displayed appropriately in your web browser.

In general, this chapter provides an overview of HTML basics and gives some practical tips to help you make the most of your time as a web developer. You'll begin to dive a bit deeper into the theory behind it all as you learn about the HTML5 elements that enable you to enhance the

semantics—the meaning—of the information that you provide in your marked-up text. You'll take a closer look at six elements that are fundamental to solid semantic structuring of your documents: `<header>`, `<section>`, `<article>`, `<nav>`, `<aside>`, and `<footer>`. Finally, you'll learn the basics of fine-tuning the display of your web content using *Cascading Style Sheets (CSS)*, which enable you to set a number of formatting characteristics, including exact typeface controls, letter and line spacing, and margins and page borders, just to name a few.

Throughout the remainder of this book, you will see HTML tags and CSS styles used appropriately in the code samples, so this chapter makes sure that you have a good grasp of their meaning before we continue.

Getting Started with a Simple Web Page

In the first chapter, you learned that a web page is just a text file that is marked up by (or surrounded by) HTML code that provides guidelines to a browser for displaying the content. To create these text files, use a plain-text editor such as Notepad on Windows or TextEdit on a Mac—do not use WordPad, Microsoft Word, or other full-featured word-processing software because those create different sorts of files than the plain-text files used for web content.

CAUTION

I'll reiterate this point because it is very important to both the outcome and the learning process itself: Do not create your first HTML file with Microsoft Word or any other word processor, even if you can save your file as HTML, because most of these programs will write your HTML for you in strange ways, potentially leaving you totally confused.

Additionally, I recommend that you *not* use a graphical, what-you-see-is-what-you-get (WYSIWYG) editor such as Adobe Dreamweaver. You'll likely find it easier and more educational to start with a simple text editor that forces you to type the code yourself as you're learning HTML and CSS.

Before you begin working, you should start with some text that you want to put on a web page:

1. Find (or write) a few paragraphs of text about yourself, your family, your company, your pets, or some other subject that holds your interest.
2. Save this text as plain, standard ASCII text. Notepad (on Windows) and most simple text editors always save files as plain text, but if you're using another program, you might need to choose this file type as an option (after selecting File, Save As).

As you go through this chapter, you will add HTML markup (called *tags*) to the text file, thus turning it into content that is best viewed in a web browser.

When you save files containing HTML tags, always give them a name ending in `.html`. This is important—if you forget to type the `.html` at the end of the filename when you save the file, most text editors will give it some other extension (such as `.txt`). If that happens, you might not be able to find the file when you try to look at it with a web browser; if you find it, it certainly won't display properly. In other words, web browsers expect a web page file to have a file extension of `.html` and to be in plain-text format.

When visiting websites, you might also encounter pages with a file extension of `.htm`, which is another acceptable file extension to use. You might find other file extensions used on the Web, such as `.jsp` (Java Server Pages), `.aspx` (Microsoft Active Server Pages), and `.php` (PHP: Hypertext Preprocessor). These files also contain HTML in addition to the programming language—although the programming code in those files is executed on the server side and all you would see on the client side is the HTML output. If you looked at the source files, you would likely see some intricate weaving of programming and markup codes. You'll learn more about this process in later chapters as you learn to integrate PHP into your websites.

Listing 2.1 shows an example of text you can type and save to create a simple HTML page. If you opened this file with your web browser, you would see the page shown in Figure 2.1. Every web page you create must include a `<!DOCTYPE>` declaration, as well as `<html></html>`, `<head></head>`, `<title></title>`, and `<body></body>` tag pairs.

LISTING 2.1 The `<html>`, `<head>`, `<title>`, and `<body>` Tags

Click here to view code image

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>The First Web Page</title>
  </head>

  <body>
    <p>
      In the beginning, Tim created the HyperText Markup Language.
The
      Internet was without form and void, and text was upon the
face of
      the monitor and the Hands of Tim were moving over the face of
the
      keyboard. And Tim said, Let there be links; and there were
links.
      And Tim saw that the links were good; and Tim separated the
links
      from the text. Tim called the links Anchors, and the text He
      called Other Stuff. And the whole thing together was the
first
      Web Page.
    </p>
  </body>
</html>
```

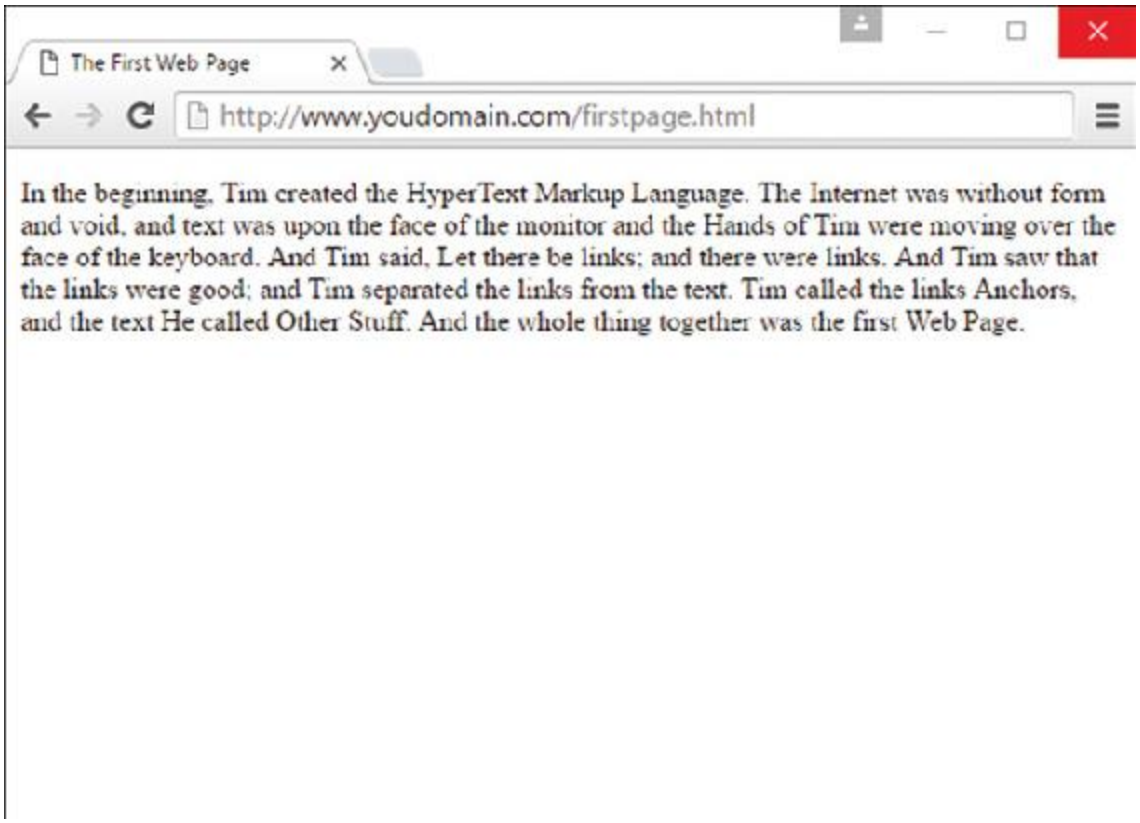

FIGURE 2.1
When you save the text in Listing 2.1 as an HTML file and view it with a web browser, only the actual title and body text are displayed.

In Listing 2.1, as in every HTML page, the words starting with < and ending with > are actually coded commands. These coded commands are called *HTML tags* because they "tag" pieces of text and tell the web browser what kind of text it is. This allows the web browser to display the text appropriately.

The first line in the document is the document type declaration; you are *declaring* that it is HTML (specifically, HTML5) because `html` is the value used to declare a document as HTML5 in the `<!DOCTYPE>` tag.

▼ TRY IT YOURSELF

Creating and Viewing a Basic Web Page

Before you learn the meaning of the HTML tags used in Listing 2.1, you might want to see exactly how I went about creating and viewing the document itself. Follow these steps:

1. Type all the text in Listing 2.1, including the HTML tags, in Windows Notepad (or use Macintosh TextEdit or another text editor of your choice).
2. Select File, Save As. Be sure to select plain text (or ASCII text) as the file type.
3. Name the file **`firstpage.html`**.
4. Choose the folder on your hard drive where you want to keep your web pages—and remember which folder you choose! Click the Save or OK button to save the file.
5. Now start your favorite web browser. (Leave Notepad running, too, so you can easily switch between viewing and editing your page.)

In Internet Explorer, select File, Open and click Browse. If you're using Firefox, select File, Open File. Navigate to the appropriate folder and select the `firstpage.html` file. Some browsers and operating systems also enable you to drag and drop the `firstpage.html` file onto the browser window to view it.

Voila! You should see the page shown in Figure 2.1.

If you have obtained a web hosting account, you could use FTP at this point to transfer the `firstpage.html` file to the web server. In fact, from this chapter forward, the instructions assume that you have a hosting provider and are comfortable sending files back and forth via FTP; if that is not the case, you should review the first chapter before moving on. Alternatively, if you are consciously choosing to work with files locally (without a web host), be prepared to adjust the instructions to suit your particular needs (such as ignoring the commands "transfer the files" and "type in the URL").

NOTE

You don’t need to be connected to the Internet to view a web page stored on your own computer. By default, your web browser probably tries to connect to the Internet every time you start it, which makes sense most of the time. However, this can be a hassle if you’re developing pages locally on your hard drive (offline) and you keep getting errors about a page not being found. If you have a full-time web connection via a cable modem, DSL, or Wi-Fi, this is a moot point because the browser will never complain about being offline. Otherwise, the appropriate action depends on your breed of browser; check the options under your browser’s Tools menu.

HTML Tags Every Web Page Must Have

The time has come for the secret language of HTML tags to be revealed to you. When you understand this language, you will have creative powers far beyond those of other humans. Don’t tell the other humans, but it’s really pretty easy.

The first line of code is the document type declaration; in HTML5, this is simply `<!DOCTYPE html>`.

This declaration identifies the document as being HTML5, which then ensures that web browsers know what to expect and can prepare to render content in HTML5.

Many HTML tags have two parts: an *opening tag*, which indicates where a piece of text begins, and a *closing tag*, which indicates where the piece of text ends. Closing tags start with a forward slash (/) just after the < symbol.

Another type of tag, or element, is the *empty element*, which is different in that it doesn’t include a pair of matching opening and closing tags. Instead, an empty element consists of a single tag that starts with the < symbol and ends with the > symbol. You may see some empty elements end with />, which is no longer required in HTML5 but did exist in previous versions of HTML.

Following is a quick summary of these three types of tags, just to make sure you understand the role each plays:

- An *opening tag* is an HTML tag that indicates the start of an HTML command; the text affected by the command appears after the opening tag. Opening tags always begin with < and end with >, as in `<html>`.
- A *closing tag* is an HTML tag that indicates the end of an HTML command; the text affected by the command appears before the closing tag. Closing tags always begin with `</` and end with >, as in `</html>`.
- An *empty tag* or *empty element* is an HTML tag that issues an HTML command without enclosing any text in the page. Examples include `<br>` for line breaks and `<img>` for images.

NOTE

You no doubt noticed in Listing 2.1 that there is some extra code associated with the `<html>` tag. This code consists of the language attribute (`lang`), which is used to specify additional information related to the tag. In this case, it specifies that the language of the text within the HTML is English. If you are writing in a different language, replace the `en` (for English) with the language identifier relevant to you.

For example, the `<body>` tag in Listing 2.1 tells the web browser where the actual body text of the page begins, and `</body>` indicates where it ends. Everything between the `<body>` and `</body>` tags appears in the main display area of the web browser window, as shown in Figure 2.1.

The very top of the browser window (refer to Figure 2.1) shows title text, which is any text that is located between `<title>` and `</title>`. The title text also identifies the page on the browser's Bookmarks or Favorites menu, depending on which browser you use. It's important to provide titles for your pages so that visitors to the page can properly bookmark them for future reference; search engines also use titles to provide a link to search results.

You will use the <body> and <title> tag pairs in every HTML page you create because every web page needs a title and body text. You will also use the <html> and <head> tag pairs, which are the other two tags shown in Listing 2.1. Putting <html> at the very beginning of a document simply indicates that the document is a web page. The </html> at the end indicates that the web page is over.

Within a page, there is a head section and a body section. Each section is identified by <head> and <body> tags. The idea is that information in the head of the page somehow describes the page but isn't actually displayed by a web browser. Information placed in the body, however, is displayed by a web browser. The <head> tag always appears near the beginning of the HTML code for a page, just after the opening <html> tag.

TIP

You might find it convenient to create and save a bare-bones page (also known as a *skeleton* page, or *template*) with just the DOCTYPE and opening and closing <html>, <head>, <title>, and <body> tags, similar to the document in Listing 2.1. You can then open that document as a starting point whenever you want to make a new web page and save yourself the trouble of typing all those obligatory tags every time.

The <title> tag pair used to identify the title of a page appears within the head of the page, which means it is placed after the opening <head> tag and before the closing </head> tag. In upcoming lessons, you'll learn about some other advanced header information that can go between <head> and </head>, such as style sheet rules for formatting the page.

The tag pair in Listing 2.1 encloses a paragraph of text. You should enclose your chunks of text in the appropriate container elements whenever possible; you'll learn more about container elements as you move forward in your lessons.

Using Hyperlinks in Web Pages

There is no rule that says you have to include links in your web content, but you would be hard-pressed to find a website that doesn't include at least one link either to another page on the same domain (for example, yourdomain.com), another domain, or even the same page. Links are all over the web, but it is important to understand a little bit of the "under the hood" details of links.

When files are part of the same domain, you can link to them by simply providing the name of the file in the `href` attribute of the `<a>` tag. An *attribute* is an extra piece of information associated with a tag that provides further details about the tag. For example, the `href` attribute of the `<a>` tag identifies the address of the page to which you are linking.

When you have more than a few pages, or when you start to have an organizational structure to the content in your site, you should put your files into directories (or *folders*, if you will) whose names reflect the content within them. For example, all your images could be in an `images` directory, company information could be in an `about` directory, and so on. Regardless of how you organize your documents within your own web server, you can use relative addresses, which include only enough information to find one page from another. A *relative address* describes the path from one web page to another, instead of a full (or *absolute*) Internet address which includes the full protocol (`http` or `https`) and the domain name (like www.yourdomain.com).

As you recall from Chapter 1, "Understanding How the Web Works," the document root of your web server is the directory designated as the top-level directory for your web content. In web addresses, that document root is represented by the forward slash (`/`). All subsequent levels of directories are separated by the same type of forward slash. Here's an example:

Click here to view code image

```
/directory/subdirectory/subsubdirectory/
```

CAUTION

The forward slash (/) is always used to separate directories in HTML. Don't use the backslash (\, which is normally used in Windows) to separate your directories. Remember, everything on the Web moves forward, so use forward slashes.

Suppose you are creating a page named `zoo.html` in your document root, and you want to include a link to pages named `african.html` and `asian.html` in the `elephants` subdirectory. The links would look like the following:

Click here to view code image

```
<a href="/elephants/african.html">Learn about African elephants.
</a>
<a href="/elephants/asian.html">Learn about Asian elephants.</a>
```

Linking Within a Page Using Anchors

The <a> tag—the tag responsible for hyperlinks on the Web—got its name from the word *anchor*, because a link serves as a designation for a spot in a web page. The <a> tag can be used to mark a spot on a page as an anchor, enabling you to create a link that points to that exact spot. For example, the top of a page could be marked as:

```
<a name="top"></a>
```

The <a> tag normally uses the `href` attribute to specify a hyperlinked target. The <a href> is what you click, and <a id> is where you go when you click there. In this example, the <a> tag is still specifying a target, but no actual link is created that you can see. Instead, the <a> tag gives a name to the specific point on the page where the tag occurs. The tag must be included and a unique name must be assigned to the `id` attribute, but no text between <a> and is necessary.

To link to this location, you would use the following:

Click here to view code image

```
<a href="#top">Go to Top of Page</a>
```

Linking to External Web Content

The only difference between linking to pages within your own site and linking to external web content is that when linking outside your site, you need to include the full address to that content. The full address includes the `http://` before the domain name and then the full pathname to the file (for example, an HTML file, an image file, or a multimedia file).

For example, to include a link to Google from within one of your own web pages, you would use this type of absolute addressing in your `<a>` link:

Click here to view code image

```
<a href="http://www.google.com/">Go to Google</a>
```

CAUTION

As you might know, you can leave out the `http://` at the front of any address when typing it into most web browsers. However, you *cannot* leave that part out when you type an Internet address into an `<a href>` link on a web page.

You can apply what you learned in previous sections to creating links to named anchors on other pages. Linked anchors are not limited to the same page. You can link to a named anchor on another page by including the address or filename followed by `#` and the anchor name. For example, the following link would take you to an anchor named `photos` within the african.html page inside the `elephants` directory on the (fictional) domain www.takeme2thezoo.com:

Click here to view code image

```
<a
href="http://www.takeme2thezoo.com/elephants/african.html#photos">
Check out the African Elephant Photos!</a>
```

If you are linking from another page already on the www.takeme2thezoo.com domain (because you are, in fact, the site maintainer), your link might simply be as follows:

Click here to view code image

```
<a href="/elephants/african.html#photos">Check out the
African Elephant Photos!</a>
```

The `http://` and the domain name would not be necessary in that instance, as you have already learned.

CAUTION

Be sure to include the `#` symbol only in `<a href>` link tags. Don't put the `#` symbol in the `<a id>` tag; links to that name won't work in that case.

Linking to an Email Address

In addition to linking between pages and between parts of a single page, the `<a>` tag enables you to link to email addresses. This is the simplest way to enable your web page visitors to talk back to you. Of course, you could just provide visitors with your email address and trust them to type it into whatever email programs they use, but that increases the likelihood for errors. By providing a clickable link to your email address, you make it almost completely effortless for them to send you messages and eliminate the chance for typos.

An HTML link to an email address looks like the following:

Click here to view code image

```
<a href="mailto:yourusername@yourdomain.com">Send me an
email message.</a>
```

The words `Send me an email message` will appear just like any other `<a>` link.

Having taken this brief foray into the world of hyperlinks, let's get back to content organization and display.

Organizing a Page with Paragraphs and Line Breaks

When a web browser displays HTML pages, it pays no attention to line endings or the number of spaces between words in the underlying text file itself. For example, the top version of the poem in Figure 2.2 appears with a single space between all words, even though that's not how it's shown in Listing 2.2. This is because extra whitespace in HTML code is automatically reduced to a single space when rendered by the web browser. Additionally, when the text reaches the edge of the browser window, it automatically wraps to the next line, no matter where the line breaks were in the original HTML file.

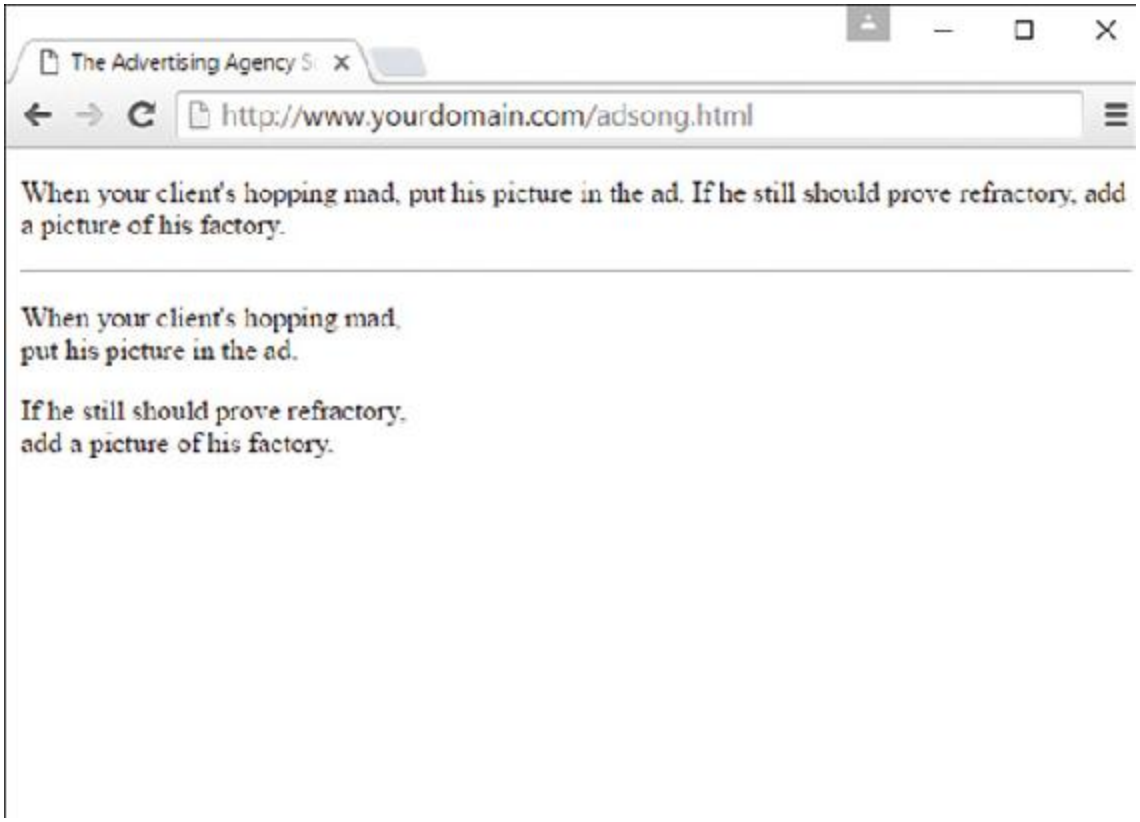

FIGURE 2.2
When the HTML in Listing 2.2 is viewed as a web page, line and paragraph breaks appear only where there are `<br>` and `<p>` tags.

LISTING 2.2 HTML Containing Paragraph and Line Breaks

Click here to view code image

```
<!DOCTYPE html>

<html lang="en">
  <head>
    <title>The Advertising Agency Song</title>
  </head>

  <body>
    <p>
      When your client's    hopping mad,
      put his picture in the ad.

      If he still should    prove refractory,
      add a picture of his factory.
    </p>
    <hr>
    <p>
      When your client's hopping mad,<br>
      put his picture in the ad.
    </p>
    <p>
      If he still should prove refractory,<br>
      add a picture of his factory.
    </p>
  </body>
</html>
```

You must use HTML tags if you want to control where line and paragraph breaks actually appear. When text is enclosed within the container tags, a line break is assumed after the closing tag. In later chapters, you'll learn to control the height of the line break using CSS. The
 tag forces a line break within a paragraph. Unlike the other tags you've seen so far,
 doesn't require a closing </br> tag—this is one of those empty elements discussed earlier.

The poem in Listing 2.2 and Figure 2.2 shows the
 and tags used to separate the lines and verses of an advertising agency song. You might have also noticed the <hr> tag in the listing, which causes a horizontal rule line to appear on the page (see Figure 2.2). Inserting a horizontal rule with the <hr> tag also causes a line break, even if you don't include a
 tag along with it. Like
, the <hr> horizontal rule tag is an empty element and, therefore, never gets a closing </hr> tag.

▼ TRY IT YOURSELF

Formatting Text in HTML

Try your hand at formatting a passage of text as proper HTML:

1. Add `<html><head><title>` *`My Title`* `</title></head><body>` to the beginning of the text (using your own title for your page instead of *My Title*). Also include the boilerplate code at the top of the page that takes care of meeting the requirements of standard HTML.
2. Add `</body></html>` to the very end of the text.
3. Add a `<p>` tag at the beginning of each paragraph and a `</p>` tag at the end of each paragraph.
4. Use `<br>` tags anywhere you want single-spaced line breaks.
5. Use `<hr>` to draw horizontal rules separating major sections of text, or wherever you'd like to see a line across the page.
6. Save the file as *`mypage`*`.html` (using your own filename instead of *`mypage`*).

CAUTION

If you are using a word processor to create the web page, be sure to save the HTML file in plain text or ASCII format.

7. Open the file in a web browser to see your web content. (Send the file via FTP to your web hosting account, if you have one.)
8. If something doesn't look right, go back to the text editor to make corrections and save the file again (and send it to your web hosting account, if applicable). You then need to click Reload/Refresh in the browser to see the changes you made.

Organizing Your Content with Headings

When you browse web pages on the Internet, you'll notice that many of them have a heading at the top that appears larger and bolder than the rest of the text. Listing 2.3 is sample code and text for a simple web page containing an example of a heading as compared to normal paragraph text. Any text between the `<h1>` and `</h1>` tags will appear as a large heading. Additionally, `<h2>` and `<h3>` make progressively smaller headings, all the way down to `<h6>`.

LISTING 2.3 Using Heading Tags

Click here to view code image

```
<!DOCTYPE html>

<html lang="en">
  <head>
    <title>My Widgets</title>
  </head>

  <body>
    <h1>My Widgets</h1>
    <p>My widgets are the best in the land. Continue reading to
    learn more about my widgets.</p>

    <h2>Widget Features</h2>
    <p>If I had any features to discuss, you can bet I'd do
    it here.</p>

    <h3>Pricing</h3>
    <p>Here, I would talk about my widget pricing.</p>

    <h3>Comparisons</h3>
    <p>Here, I would talk about how my widgets compare to my
    competitor's widgets.</p>
  </body>
</html>
```

NOTE

By now, you've probably caught on to the fact that HTML code is often indented by its author to reveal the relationship between different parts of

the HTML document, as well as for simple ease of reading. This indentation is entirely voluntary—you could just as easily run all the tags together with no spaces or line breaks, and they would still look fine when viewed in a browser. The indentations are for you so that you can quickly look at a page full of code and understand how it fits together. Indenting your code is another good web design habit and ultimately makes your pages easier to maintain, both for yourself and for anyone else who might pick up where you leave off.

As you can see in Figure 2.3, the HTML that creates headings couldn't be simpler. In this example, the phrase "My Widgets" is given the highest level of heading, and is prominently displayed, by surrounding it with the `<h1>` `</h1>` tag pair. For a slightly smaller (level 2) heading—for information that is of lesser importance than the title—use the `<h2>` and `</h2>` tags around your text. For content that should appear even less prominently than a level 2 heading, use the `<h3>` and `</h3>` tags around your text.

However, bear in mind that your headings should follow a content hierarchy; use only one level 1 heading, have one (or more) level 2 headings after the level 1 heading, use level 3 headings only after level 2 headings, and so on. Do not fall into the trap of assigning headings to content just to make that content display a certain way, such as by skipping headings. Instead, ensure that you are categorizing your content appropriately (as a main heading, a secondary heading, and so on) while using display styles to make that text render a particular way in a web browser.

You can also use `<h4>`, `<h5>`, and `<h6>` tags to make progressively less important headings. By default, web browsers seldom show a noticeable difference between these headings and the `<h3>` headings—although you can control that with your own CSS. Also, content usually isn't displayed in such a manner that you'd need six levels of headings to show the content hierarchy.

It's important to remember the difference between a *title* and a *heading*. These two words are often interchangeable in day-to-day English, but when you're talking HTML, `<title>` gives the entire page an identifying name

that isn't displayed on the page itself; it's displayed only on the browser window's title bar. The heading tags, on the other hand, cause some text on the page to be displayed with visual emphasis. There can be only one `<title>` per page, and it must appear within the `<head>` and `</head>` tags; on the other hand, you can have as many `<h1>`, `<h2>`, and `<h3>` headings as you want, in any order that suits your fancy. However, as I mentioned before, you should use the heading tags to keep tight control over content hierarchy (logic dictates only one `<h1>` heading); do not use headings as a way to achieve a particular look, because that's what CSS is for.

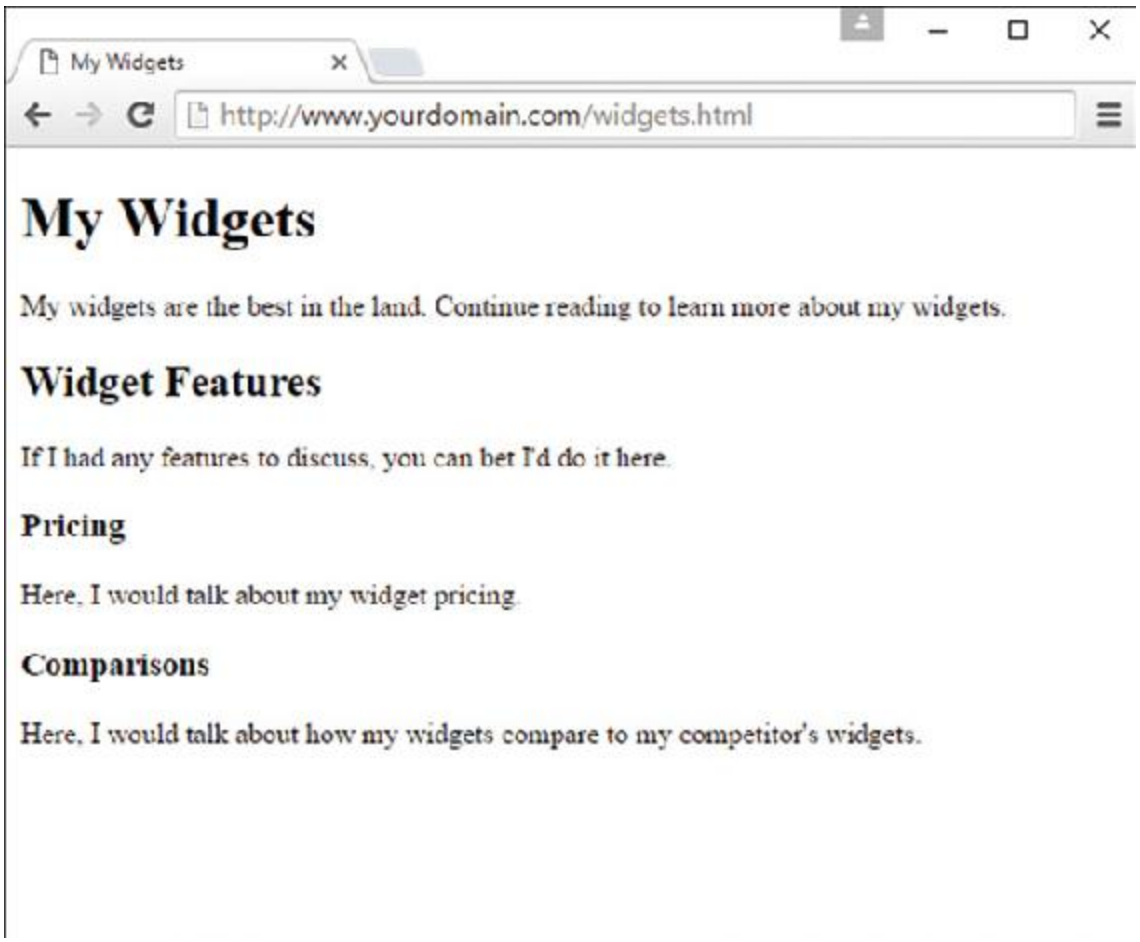

FIGURE 2.3
Using three levels of headings shows the hierarchy of content on this sample product page.

CAUTION

Don’t forget that anything placed in the head of a web page is not intended to be viewed on the page, whereas everything in the body of the page *is* intended for viewing.

Peeking at Other Designers’ Pages

Given the visual and sometimes audio pizzazz present in many popular web pages, you probably realize that the simple pages described in this lesson are only the tip of the HTML iceberg. Now that you know the basics, you might surprise yourself with how much of the rest you can pick up just by looking at other people’s pages on the Internet. You can see the HTML for any page by right-clicking and selecting View Source in any web browser.

Don’t worry if you aren’t yet able to decipher what some HTML tags do or exactly how to use them yourself. You’ll find out about all those things as you move forward in the book. However, sneaking a preview now will show you the tags that you do know in action and give you a taste of what you’ll soon be able to do with your web pages.

Understanding Semantic Elements

HTML5 includes tags that enable you to enhance the semantics—the meaning—of the information you provide in your marked-up text. Instead of simply using HTML as a presentation language, as was the practice in the very early days when `<b>` for bold and `<i>` for italics was the norm, modern HTML has as one of its goals the separation of presentation and meaning. While using CSS to provide guidelines for presentation, composers of HTML can provide meaningful names within their markup for individual elements, not only through the use of IDs and class names (which you’ll learn about later in this chapter), but also through the use of semantic elements.

Some of the semantic elements available in HTML5 follow:

- **<header></header>**—This might seem counterintuitive, but you can use multiple <header> tags within a single page. The <header> tag should be used as a container for introductory information, so it might be used only once in your page (likely at the top), but you also might use it several times if your page content is broken into sections. Any container element can have a <header> element; just make sure that you're using it to include introductory information about the element it is contained within.
- **<footer></footer>**—The <footer> tag is used to contain additional information about its containing element (page or section), such as copyright and author information or links to related resources.
- **<nav></nav>**—If your site has navigational elements, such as links to other sections within a site or even within the page itself, these links go in a <nav> tag. A <nav> tag typically is found in the first instance of a <header> tag, just because people tend to put navigation at the top and consider it introductory information—but that is not a requirement. You can put your <nav> element anywhere (as long as it includes navigation), and you can have as many on a page as you need (often no more than two, but you might feel otherwise).
- **<section></section>**—The <section> tag contains anything that relates thematically; it can also contain a <header> tag for introductory information and possibly a <footer> tag for other related information. You can think of a <section> as carrying more meaning than a standard <p> (paragraph) or (division) tag, which typically conveys no meaning at all; the use of <section> conveys a relationship between the content elements it contains.
- **<article></article>**—An <article> tag is like a <section> tag, in that it can contain a <header>, a <footer>, and other container elements such as paragraphs and divisions. But the additional meaning carried with the <article> tag is that it is, well, like an article in a newspaper or some other publication. Use this tag around blog posts, news articles, reviews, and other items that fit this description. One key difference between an <article> and a <section> is that an <article> is a standalone body of work, whereas a <section> is a thematic grouping of information.

- **<aside></aside>**—Use the <aside> tag to indicate secondary information; if the <aside> tag is within a <section> or an <article>, the relationship will be to those containers; otherwise, the secondary relationship will be to the overall page or site itself. It might make sense to think of the <aside> as a sidebar—either for all the content on the page or for an article or other thematic container of information.

These semantic elements will become clearer as you practice using them. In general, using semantic elements is a good idea because they provide additional meaning not only for you and other designers and programmers reading and working with your markup, but also for machines. Web browsers and screen readers will respond to your semantic elements by using them to determine the structure of your document; screen readers will report a deeper meaning to users, thus increasing the accessibility of your material.

One of the best ways to understand the HTML5 semantic elements is to see them in action, but that can be a little difficult when the primary purpose of these elements is to provide *meaning* rather than design. That's not to say that you can't add design to these elements—you most certainly can. But the "action" of the semantic elements is to hold content and provide meaning through doing so, as in Figure 2.4, which shows a common use of semantic elements for a basic web page.

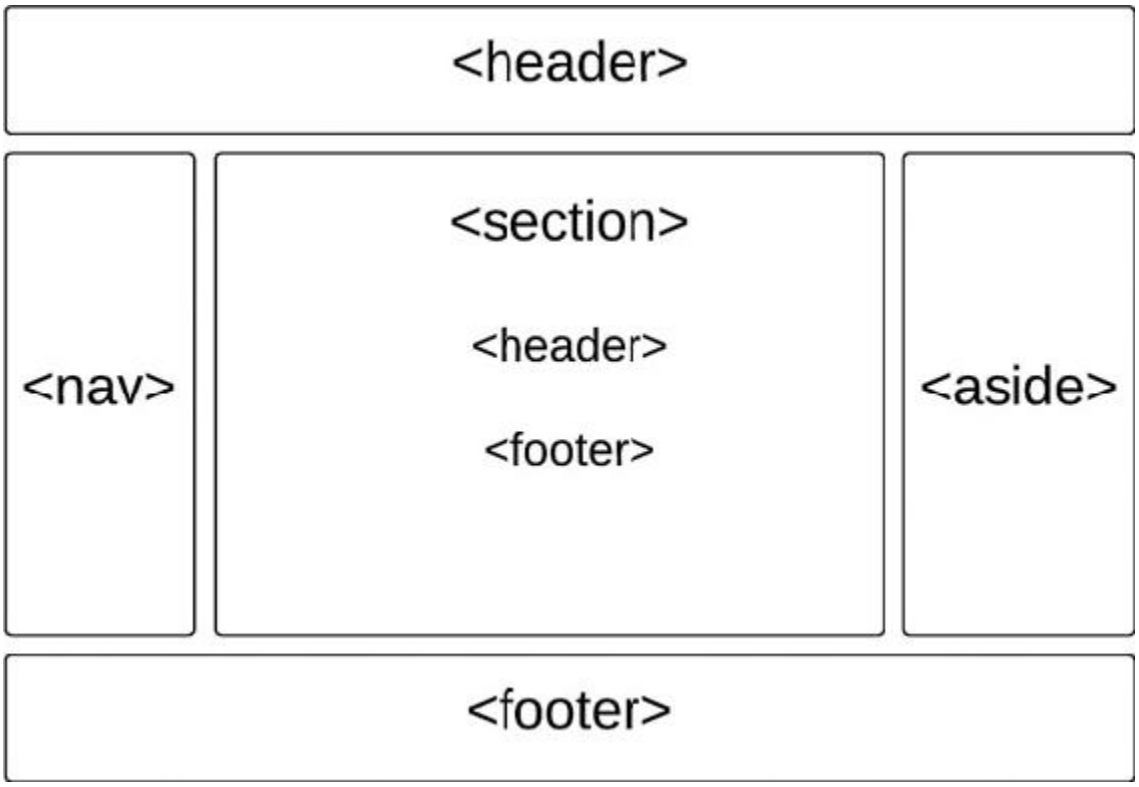

FIGURE 2.4
Showing basic semantic elements in a web page.

Initially, you might think, "Of *course*, that makes total sense, with the header at the top and the footer at the bottom," and feel quite good about yourself for understanding semantic elements at first glance—and you should! A second glance should then raise some questions: What if you want your navigation to be horizontal under your header? Does an aside have to be (literally) on the side? What if you don't want any asides? What's with the use of `<header>` and `<footer>` again within the main body section? And that's just to name a few! Something else you might wonder about is where the `<article>` element fits in; it isn't shown in this example but will be used later in this chapter.

This is the time when conceptualizing the page—and specifically the page *you* want to create—comes into play. If you understand the content you want to mark up and you understand that you can use any, all, or none of the semantic elements and still create a valid HTML document, then you can begin to organize the content of your page in the way that makes the most sense for it and for you (and, hopefully, for your readers).

NOTE

Although you do not need to use semantic elements to create a valid HTML document, even a minimal set is recommended so that web browsers and screen readers can determine the structure of your document. Screen readers are capable of reporting a deeper meaning to users, thus increasing the accessibility of your material.

(If this note were marked up in an HTML document, it would use the `<aside>` element.)

Let's take a look at the elements used in Figure 2.4 before moving on to a second example and then a deeper exploration of the individual elements themselves. In Figure 2.4, you see a `<header>` at the top of the page and a `<footer>` at the bottom—straightforward, as already mentioned. The use of a `<nav>` element on the left side of the page matches a common display area for navigation, and the `<aside>` element on the right side of the page matches a common display area for secondary notes, pull quotes, helper text, and "for more information" links about the content. In Figure 2.5, you'll see some of these elements shifted around, so don't worry—Figure 2.4 is not some immutable example of semantic markup.

Something you might be surprised to see in Figure 2.5 is the `<header>` and `<footer>` inside the `<section>` element. As you'll learn shortly, the role of the `<header>` element is to introduce a second example and then a deeper exploration of the individual elements themselves. In Figure 2.4, you see a `<header>` at the top of the page and a `<footer>` at the bottom—straightforward, as already mentioned. The use of a `<nav>` element on the left side of the page matches the content that comes after it, and the `<header>` element itself does not convey any level in a document outline. Therefore, you can use as many as you need to mark up your content appropriately; a `<header>` at the beginning of the page might contain introductory information about the page as a whole, and the `<header>` element within the `<section>` element might just as easily and appropriately contain introductory information about the content within it. The same is true for the multiple appearances of the `<footer>` element in this example.

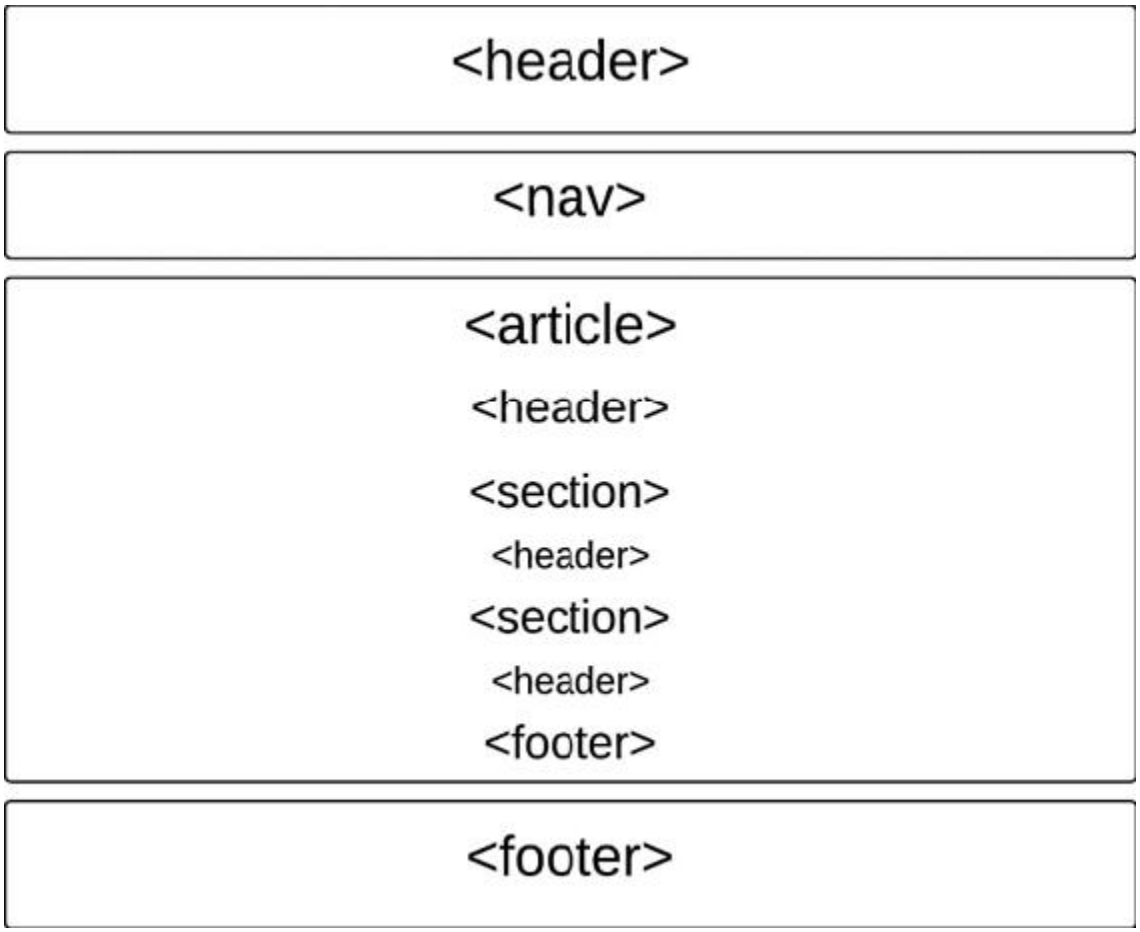

FIGURE 2.5
Using nested semantic elements to add more meaning to the content.

Let's move on to Figure 2.5, which shifts around the `<nav>` element and also introduces the use of the `<article>` element.

In Figure 2.5, the `<header>` and `<nav>` elements at the beginning of the page, and the `<footer>` element at the bottom of the page, should make perfect sense to you. And, although we haven't talked about the `<article>` element yet, if you think about it as a container element that has sections (`<section>`s, even!), with each of those sections having its own heading, then the chunk of semantic elements in the middle of the figure should make sense, too. As you can see, there's no single way to conceptualize a page; you should conceptualize content, and that content will be different on each page in a web site.

If you marked up some content in the structure shown in Figure 2.5, it might look like Listing 2.4.

LISTING 2.4 Semantic Markup of Basic Content

Click here to view code image

```
<!DOCTYPE html>

<html lang="en">
  <head>
    <title>Semantic Example</title>
  </head>
  <body>
    <header>
       <h1>SITE OR PAGE LOGO GOES HERE</h1>
    </header>
    <nav>
       SITE OR PAGE NAV GOES HERE.
    </nav>
    <article>
       <header>
          <h2>Article Heading</h2>
       </header>
       <section>
          <header>
             <h3>Section 1 Heading</h3>
          </header>
          <p>Section 1 content here.</p>
       </section>
       <section>
          <header>
             <h3>Section 2 Heading</h3>
          </header>
          <p>Section 2 content here.</p>
       </section>
       <footer>
          <p>Article footer goes here.</p>
       </footer>
    </article>
    <footer>
       SITE OR PAGE FOOTER HERE
    </footer>
  </body>
</html>
```

If you opened this HTML document in your web browser, you would see something like what's shown in Figure 2.6—a completely unstyled

document, but one that has semantic meaning (even if no one can “see” it).

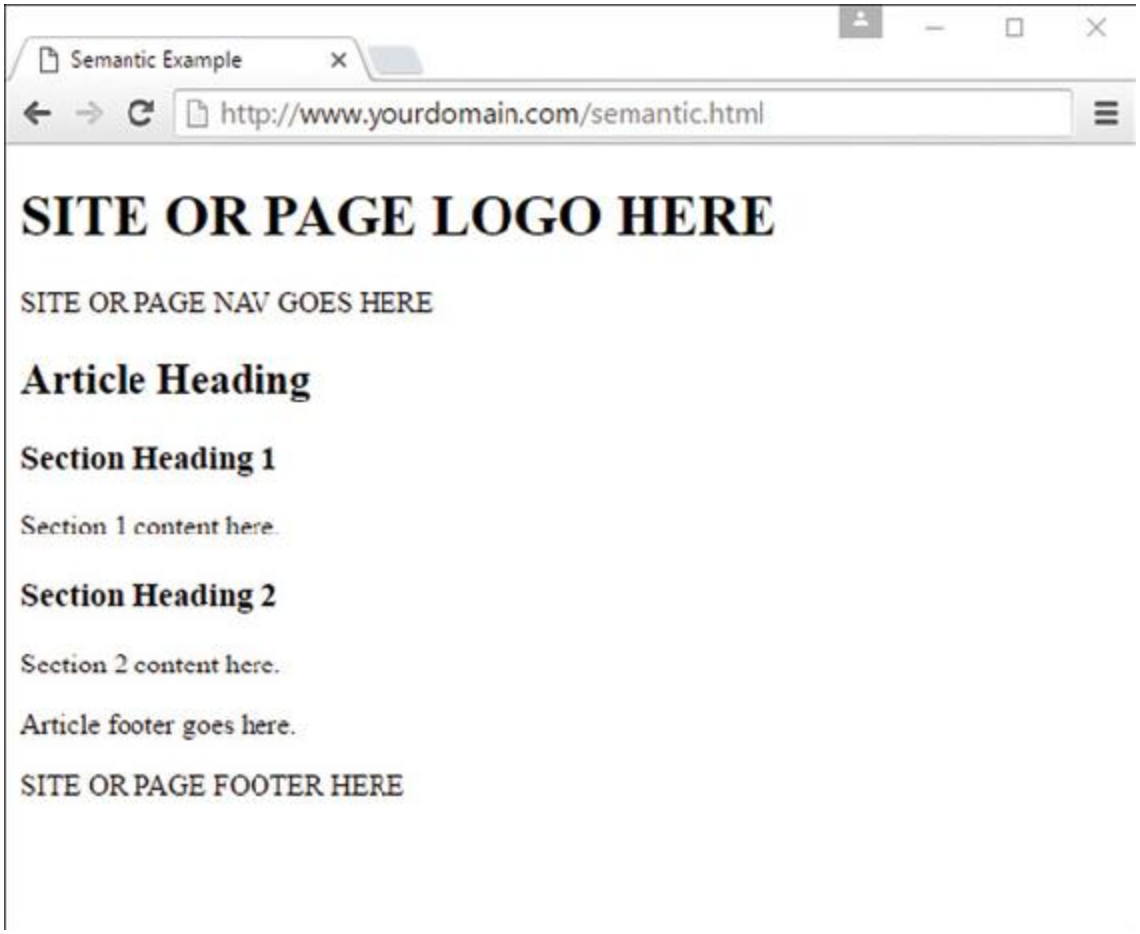

FIGURE 2.6
The output of Listing 2.4.

Just because there is no visible styling doesn’t mean the meaning is lost; as noted earlier in this section, machines can interpret the structure of the document as provided for through the semantic elements. You can see the outline of this basic document in Figure 2.7, which shows the output of this file after examination by the HTML5 Outline tool at http://gsnedders.html5.org/outliner/.

TIP

Using the HTML5 Outline tool is a good way to check that you’ve created your headers, footers, and sections; if you examine your document and see “untitled section” anywhere, and those untitled sections do not match up

with a <nav> or <aside> element (which has more relaxed guidelines about containing headers), then you have some additional work to do.

Now that you've seen some examples of conceptualizing the information represented in your documents, you're better prepared to start marking up those documents. The sections that follow take a look at the semantic elements individually.

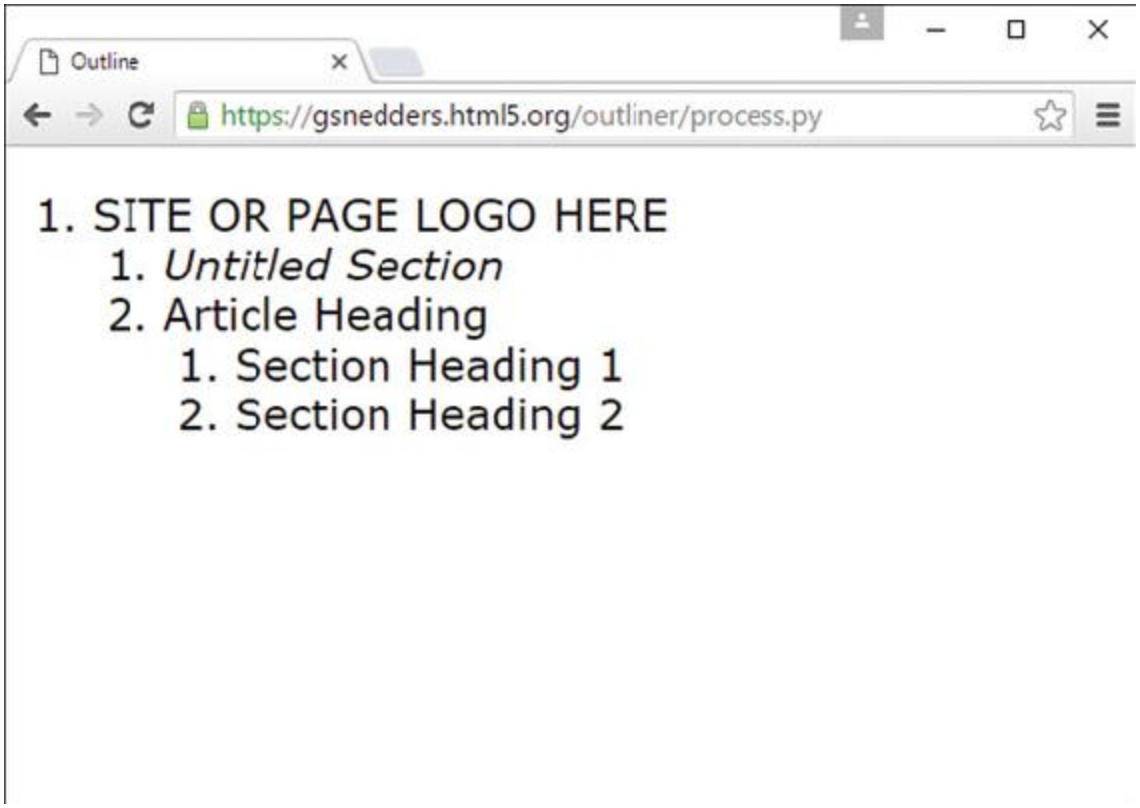

FIGURE 2.7
The outline of this document follows the semantic markup.

Using <header> in Multiple Ways

At the most basic level, the <header> element contains introductory information. That information might take the form of an actual <h1> (or other level) element, or it might simply be a logo image or text contained within a <p> or element. The *meaning* of the content should be introductory in nature, to warrant its inclusion within a <header> </header> tag pair.

As you've seen in the examples so far in this lesson, a common placement of a <header> element is at the beginning of a page. When it's used in this way, containing a logo or an <h1>-level title makes sense, such as here:

Click here to view code image

```
<header>
     <img src="acmewidgets.jpg" alt="ACME Widgets LLC">
</header>
```

Or even here:

Click here to view code image

```
<header>
     <img src="acmewidgets.jpg" alt="ACME Widgets LLC">
     <h1>The finest widgets are made here!</h1>
</header>
```

Both snippets are valid uses of <header> because the information contained within them is introductory to the page overall.

As you've also seen in this chapter, you are not limited to only one <header>. You can go crazy with your <header> elements, as long as they are acting as containers for introductory information—Listing 2.4 showed the use of <header> elements for several <section> elements within an <article>, and this is a perfectly valid use of the element:

Click here to view code image

```
<section>
   <header>
      <h3>Section 1 Heading</h3>
   </header>
   <p>Section 1 content here.</p>
</section>
<section>
   <header>
      <h3>Section 2 Heading</h3>
   </header>
   <p>Section 2 content here.</p>
</section>
```

The <header> element can contain any other element in the flow content category, of which it is also a member. This means that a <header> *could* contain a <section> element, if you wanted, and be perfectly valid markup. However, when you are conceptualizing your content, think about whether that sort of nesting makes sense before you go off and do it.

NOTE

In general, *flow content* elements are elements that contain text, images, or other multimedia embedded content; HTML elements fall into multiple categories.

If you want to learn more about the categorization of elements into content models, see http://www.w3.org/TR/2011/WD-html5-20110525/content-models.html.

The only exceptions to the permitted content within <header> are that the <header> element cannot contain *other* <header> elements and it cannot contain a <footer> element. Similarly, the <header> element cannot be contained within a <footer> element.

Understanding the <section> Element

The <section> element has a simple definition: It is a "generic section of a document" that is also a "thematic grouping of content, typically with a heading." That sounds pretty simple to me, and probably does to you as well. So you might be surprised to find that if you type "difference between section and article in HTML5" in your search engine of choice, you'll find tens of thousands of entries talking about the differences because the definitions trip people up all the time. We first discuss the <section> element and then cover the <article> element—and hopefully avoid any of the misunderstandings that seem to plague new web developers.

In Listing 2.4, you saw a straightforward example of using <section> within an <article> (repeated here). In this example, you can easily

imagine that the <section>s contain a "thematic grouping of content," which is supported by the fact that they each have a heading:

Click here to view code image

```
<article>
    <header>
       <h2>Article Heading</h2>
    </header>
    <section>
       <header>
          <h3>Section 1 Heading</h3>
       </header>
       <p>Section 1 content here.</p>
    </section>
    <section>
       <header>
          <h3>Section 2 Heading</h3>
       </header>
       <p>Section 2 content here.</p>
    </section>
    <footer>
       <p>Article footer goes here.</p>
    </footer>
</article>
```

But here's an example of a perfectly valid use of <section> with no <article> element in sight:

Click here to view code image

```
<section>
    <header>
       <h1>Super Heading</h1>
    </header>
    <p>Super content!</p>
</section>
```

So what's a developer to do? Let's say you have some generic content that you know you want to divide into sections with their own headings. In that case, use <section>. If you need to only *visually* delineate chunks of content (such as with paragraph breaks) that do not require additional headings, then <section> isn't for you—use <p> or instead.

Because the <section> element can contain any other flow content element, and can be contained within any other flow content element

(except the <address> element, discussed later in this chapter), it's easy to see why, without other limitations and with generic guidelines for use, the <section> element is sometimes misunderstood.

Using <article> Appropriately

Personally, I believe that a lot of the misunderstanding regarding the use of <section> versus <article> has to do with the name of the <article> element. When I think of an article, I think specifically about an article in a newspaper or a magazine. I don't naturally think "any standalone body of work," which is how the <article> element is commonly defined. The HTML5 recommended specification defines it as "a complete, or self-contained, composition in a document, page, application, or site and that is, in principle, independently distributable or reusable," such as "a forum post, a magazine or newspaper article, a blog entry, a user-submitted comment, an interactive widget or gadget, or any other independent item of content."

In other words, an <article> element could be used to contain the entire page of a website (whether or not it is an article in a publication), an actual article in a publication, a blog post anywhere and everywhere, part of a threaded discussion in a forum, a comment on a blog post, and as a container that displays the current weather in your city. It's no wonder there are tens of thousands of results for a search on "difference between section and article in HTML5."

A good rule of thumb when you're trying to figure out when to use <article> and when to use <section> is simply to answer the following question: Does this content make sense on its own? If so, then no matter what the content seems to be to you (for example, a static web page, not an article in the *New York Times*), start by using the <article> element. If you find yourself breaking it up, do so in <section>s. And if you find yourself thinking that your "article" is, in fact, part of a greater whole, then change the <article> tags to <section> tags, find the beginning of the document, and surround it from there with the more appropriately placed <article> tag at a higher level.

Implementing the <nav> Element

The <nav> element seems so simple (<nav> implies *navigation*), and it ultimately is—but it can also be used incorrectly. In this section, you'll learn some basic uses, and also some incorrect uses to avoid. If your site has any navigational elements at all, either sitewide or within a long page of content, you have a valid use for the <nav> element.

For that sitewide navigation, you typically find a <nav> element within the primary <header> element; you are not required to put it there, but if you want your navigational content to be introductory (and omnipresent in your template), you can easily make a case for your primary <nav> element to appear within the primary <header>. More important, that is valid HTML (as is <nav> outside a <header>) because a <nav> element can appear within any flow content, as well as contain any flow content.

The following code snippet shows the main navigational links of a website, placed within a <header> element:

Click here to view code image

```
<header>
   <img src="acmewidgets.jpg" alt="ACME Widgets LLC"/>
   <h1>The finest widgets are made here!</h1>
   <nav>
      <ul>
         <li><a href="#">About Us</a></li>
         <li><a href="#">Products</a></li>
         <li><a href="#">Support</a></li>
         <li><a href="#">Press</a></li>
      </ul>
   </nav>
</header>
```

You are not limited to a single <nav> element in your documents, which is good for site developers who create templates that include both primary *and* secondary navigation. For example, you might see horizontal primary navigation at the top of a page (often contained within a <header> element), and then vertical navigation in the left column of a page, representing the secondary pages within the main section. In that case, you simply use a second <nav> element, not contained within the <header>,

placed and styled differently to delineate the two types visually in addition to semantically.

Remember, the `<nav>` element is used for *major* navigational content—primary and secondary navigation both count, as does the inclusion of tables of contents within a page. For good and useful semantic use of the `<nav>` element, do not simply apply it to every link that allows a user to navigate anywhere. Note that I said "good and useful" semantic use, not necessarily "valid" use—it's true that you could apply `<nav>` to any list of links, and it would be valid according to the HTML specification because links are flow content. But it wouldn't be particularly *useful*—it wouldn't add meaning—to surround a list of links to social media sharing tools with the `<nav>` element.

When to Use `<aside>`

As you'll see by the number of tips and notes from me throughout this book, I'm a big fan of the type of content that is most appropriately marked up within the `<aside>` element. The `<aside>` element is meant to contain any content that is tangentially related to the content around it—additional explanation, links to related resources, pull quotes, helper text, and so on. You might think of the `<aside>` element as a sidebar, but be careful not to think of it only as a *visual* sidebar, or a column on the side of a page where you can stick anything and everything you want, whether or not it's related to the content or site at hand.

In Figure 2.8, you can see how content in an `<aside>` is used to create a *pull quote*, or a content excerpt that is specifically set aside to call attention to it. The `<aside>`, in this case, is used to highlight an important section of the text, but it could also have been used to define a term or link to related documents.

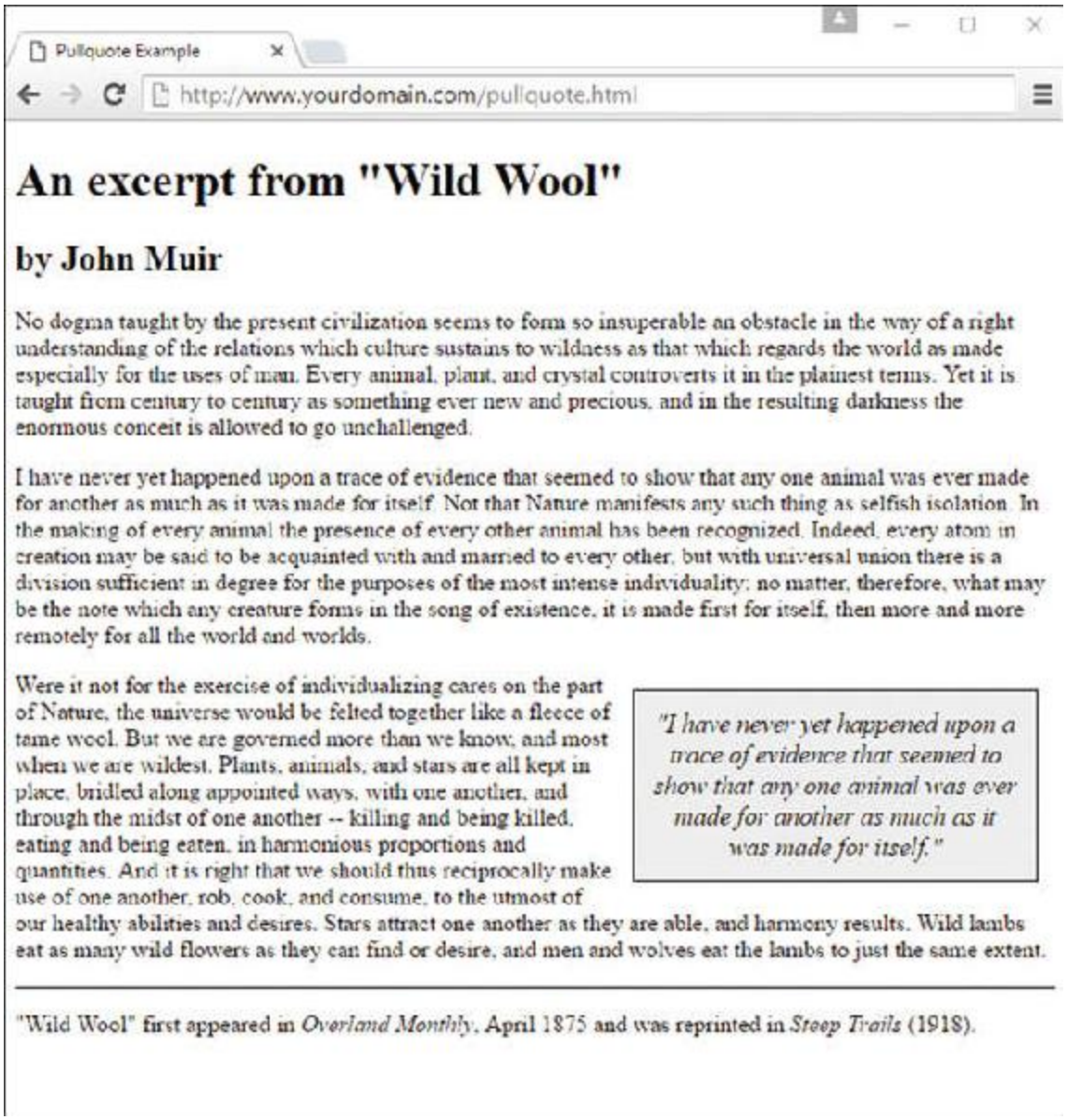

FIGURE 2.8
Using `<aside>` to create meaningful pull quotes.

When determining whether to use the `<aside>` element, think about the content you want to add. Is it related directly to the content in which the `<aside>` would be contained, such as a definition of terms used in an article or a list of related links for the article? If your answer is an easy yes, that's great! Use `<aside>` to your heart's content. If you're thinking of including an `<aside>` outside a containing element that is itself full of content, just make sure that the content of the `<aside>` is reasonably

related to your site overall and that you're not just using the <aside> element for visual effect.

Using <footer> Effectively

The counterpart to the <header> element, the <footer> element, contains additional information about its containing element. The most common use of the <footer> element is to contain copyright information at the bottom of a page, such as here:

Click here to view code image

```
<footer>
   <p>&copy; 2017 Acme Widgets, LLC. All Rights Reserved.</p>
</footer>
```

Similar to the <header> element, the <footer> element can contain any other element in the flow content category, of which it is also a member, with the exception of *other* <footer> or <header> elements. Additionally, a <footer> element cannot be contained within an <address> element, but a <footer> element can contain an <address> element—in fact, a <footer> element is a common location for an <address> element to reside in.

Placing useful <address> content within a <footer> element is one of the most effective uses of the <footer> element (not to mention the <address> element) because it provides specific contextual information about the page or section of the page to which it refers. The following snippet shows a use of <address> within <footer>:

Click here to view code image

```
<footer>
   <p>&copy; 2017 Acme Widgets, LLC. All Rights Reserved.</p>
   <p>Copyright Issues? Contact:</p>
      <address>
      Our Lawyer<br>
      123 Main Street<br>
      Somewhere, CA 95128<br>
      <a href="mailto:lawyer@example.com">lawyer@example.com</a>
      </address>
</footer>
```

As with the <header> element, you are not limited to only one <footer>. You can use as many <footer> elements as you need, as long as they are containers for additional information about the containing element—Listing 2.4 showed the use of <footer> elements for both a page and an <article>, both of which are valid.

How CSS Works

In the preceding sections, you learned the basics of HTML, including how to set up a skeletal HTML template for all your web content, use hyperlinks, and generally organize your content. In this section, you'll learn the basics of fine-tuning the visual display of your web content using *Cascading Style Sheets (CSS)*.

The concept behind style sheets is simple: You create a style sheet document that specifies the fonts, colors, spacing, and other characteristics that establish a unique look for a website. You then link every page that should have that look to the style sheet instead of specifying all those styles repeatedly in each separate document. Therefore, when you decide to change your official corporate typeface or color scheme, you can modify all your web pages at once just by changing one or two entries in your style sheet—you don't have to change them in all your static web files. So a *style sheet* is a grouping of formatting instructions that control the appearance of several HTML pages at once.

Style sheets enable you to set a great number of formatting characteristics, including exact typeface controls, letter and line spacing, and margins and page borders, just to name a few. Style sheets also enable you to specify sizes and other measurements in familiar units, such as inches, millimeters, points, and picas. In addition, you can use style sheets to precisely position graphics and text anywhere on a web page, either at specific coordinates or relative to other items on the page.

In short, style sheets bring a sophisticated level of display to the Web—and they do so, if you'll pardon the expression, with style.

NOTE

If you have three or more web pages that share (or should share) similar formatting and fonts, you might want to create a style sheet for them as you read this chapter. Even if you choose not to create a complete style sheet, you'll find it helpful to apply styles to individual HTML elements directly within a web page.

A *style rule* is a formatting instruction that can be applied to an element on a web page, such as a paragraph of text or a link. Style rules consist of one or more style properties and their associated values. An *internal style sheet* is placed directly within a web page, whereas an *external style sheet* exists in a separate document and is simply linked to a web page via a special tag —more on this tag in a moment.

The *cascading* part of the name Cascading Style Sheets refers to the manner in which style sheet rules are applied to elements in an HTML document. More specifically, styles in a CSS style sheet form a hierarchy in which more specific styles override more general styles. It is the responsibility of CSS to determine the precedence of style rules according to this hierarchy, which establishes a cascading effect. If that sounds a bit confusing, just think of the cascading mechanism in CSS as being similar to genetic inheritance, in which general traits are passed from parents to a child, but more specific traits are entirely unique to the child. Base-style rules are applied throughout a style sheet but can be overridden by more specific style rules.

NOTE

You might notice that I use the term *element* a fair amount in this chapter (and I do in the rest of the book, for that matter). An *element* is simply a piece of information (content) in a web page, such as an image, a paragraph, or a link. Tags are used to mark up elements, and you can think of an element as a tag, complete with descriptive information (attributes, text, images, and so on) within the tag.

A quick example should clear things up. Take a look at the following code to see whether you can tell what's going on with the color of the text:

Click here to view code image

```
<div style="color:green">
  This text is green.
  <p style="color:blue">This text is blue.</p>
  <p>This text is still green.</p>
</div>
```

In the preceding example, the color green is applied to the `<div>` tag via the `color` style property. Therefore, the text in the `<div>` tag is colored green. Because both `<p>` tags are children of the `<div>` tag, the green text style cascades down to them. However, the first `<p>` tag overrides the color style and changes it to blue. The end result is that the first line (not surrounded by a paragraph tag) is green, the first official paragraph is blue, and the second official paragraph retains the cascaded green color.

If you made it through that description on your own and came out on the other end unscathed, congratulations—that's half the battle. Understanding CSS isn't like understanding rocket science—the more you practice, the more it will become clear. The real trick is developing the aesthetic design sense that you can then apply to your online presence through CSS.

Like many web technologies, CSS has evolved over the years. The original version of CSS, known as *Cascading Style Sheets Level 1* (*CSS1*), was created in 1996. The later CSS2 standard was created in 1998, and CSS2 is still in use today; all modern web browsers support CSS2. The latest version of CSS is CSS3, which builds on the strong foundation laid by its predecessors but adds advanced functionality to enhance the online experience. In the following sections you'll learn core CSS, and throughout the rest of the book when CSS is discussed or used, I'll refer to CSS3.

The rest of this chapter explains the basics of putting CSS to good use, but it's not a reference for all things CSS. Nor is the rest of this book, which will show plenty of examples of basic CSS in use as you learn to build dynamic web applications. However, you can find a developer-oriented guide to CSS at https://developer.mozilla.org/en-US/docs/Web/CSS that gets into excruciating detail regarding everything you can do with CSS.

This guide can be an invaluable reference to you as you continue on your web development journey.

A Basic Style Sheet

Despite their power, style sheets are simple to create. Consider the web pages shown in Figures 2.9 and 2.10. These pages share several visual properties that can be put into a common style sheet:

- They use a large, bold Verdana font for the headings and a normal-size and normal-weight Verdana font for the body text.
- They use an image named `logo.gif` floating within the content and on the right side of the page.
- All text is black except for subheadings, which are purple.
- They have margins on the left side and at the top.
- They include vertical space between lines of text.
- They include a footer that is centered and in small print.

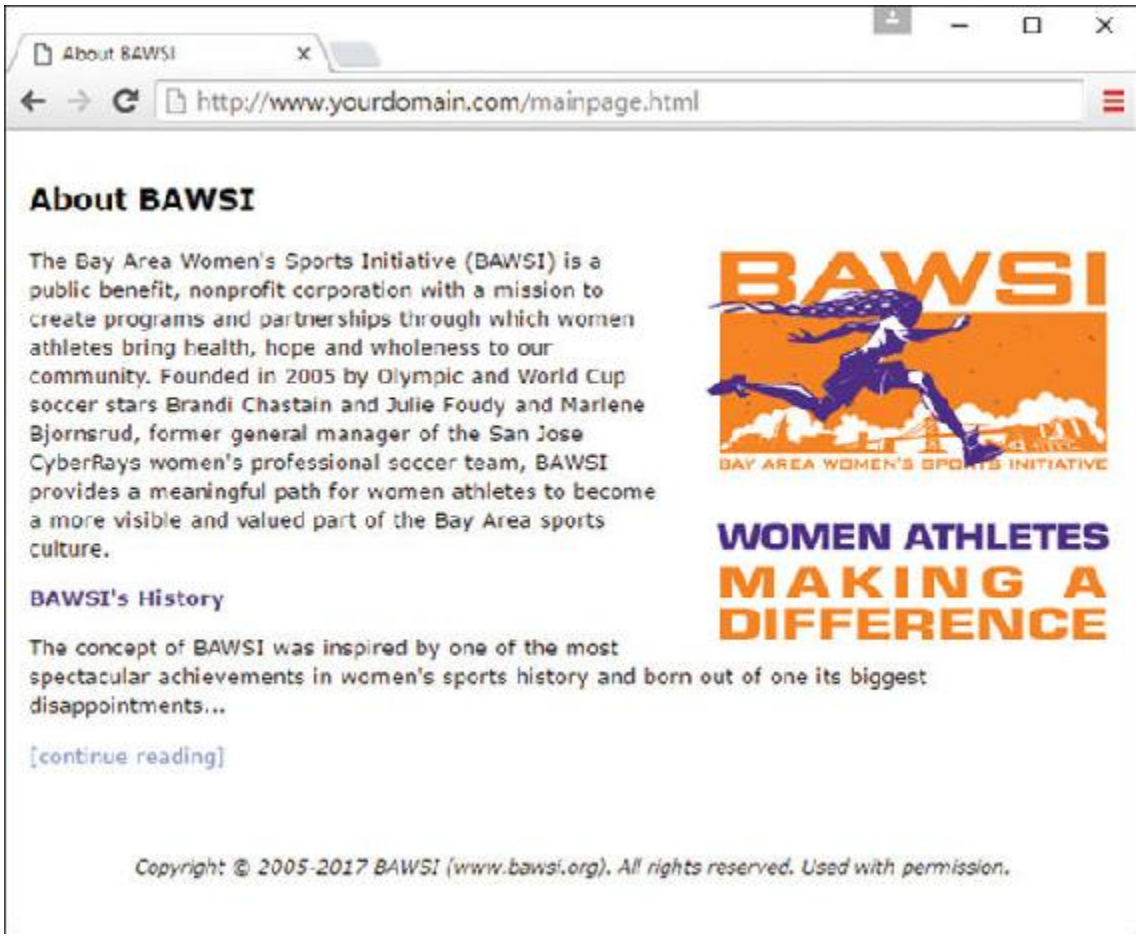

FIGURE 2.9
This page uses a style sheet to fine-tune the appearance and spacing of the text and images.

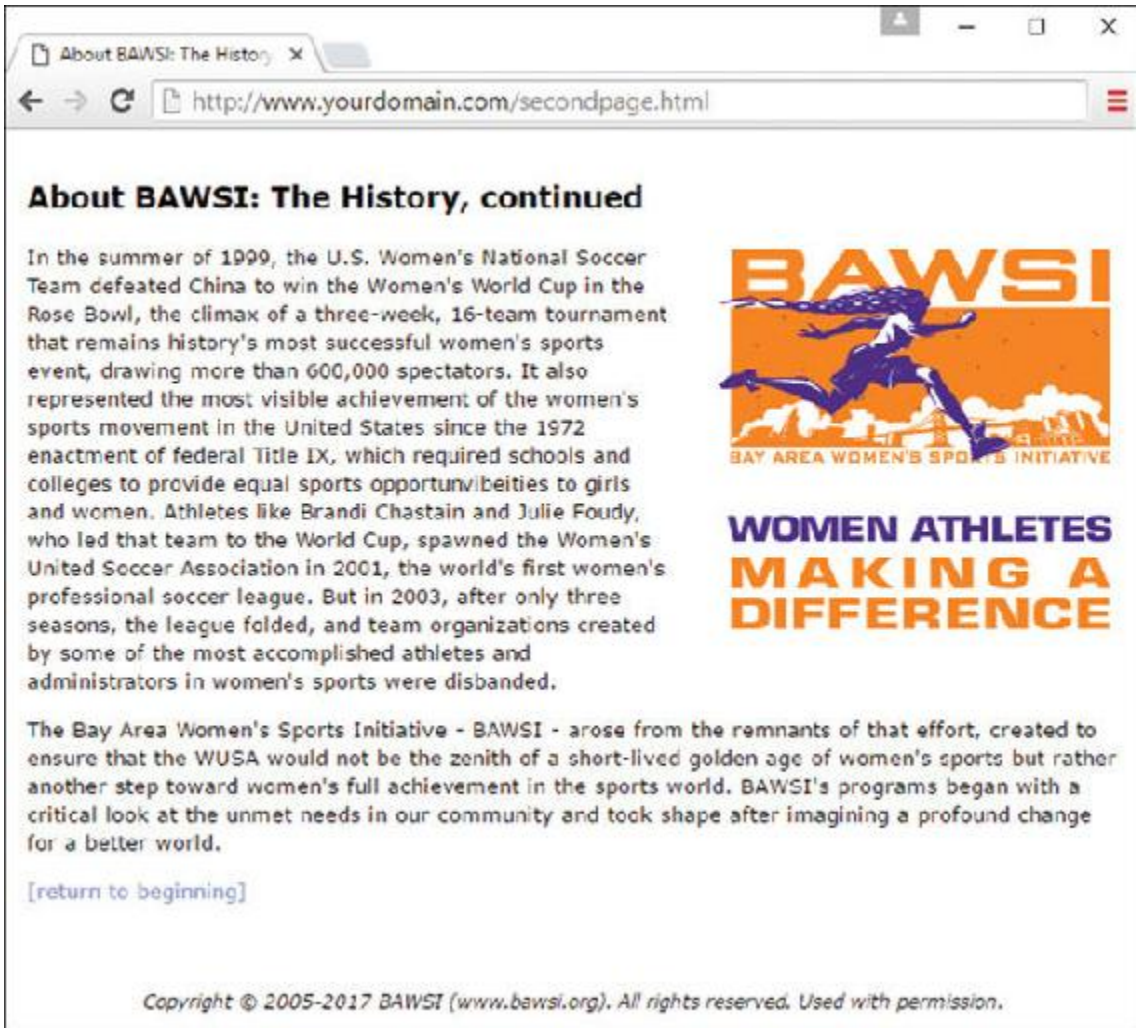

FIGURE 2.10
This page uses the same style sheet as the one in Figure 2.9, thus maintaining a consistent look and feel.

Listing 2.5 shows the CSS used in a style sheet to specify these properties.

LISTING 2.5 A Single External Style Sheet

Click here to view code image

```
body {
  font-size: 10pt;
  font-family: Verdana, Geneva, Arial, Helvetica, sans-serif;
  color: black;
  line-height: 14pt;
  padding-left: 5pt;
  padding-right: 5pt;
  padding-top: 5pt;
```

```
}

h1 {
  font: 14pt Verdana, Geneva, Arial, Helvetica, sans-serif;
  font-weight: bold;
  line-height: 20pt;
}

p.subheader {
  font-weight: bold;
  color: #593d87;
}

img {
  padding: 3pt;
  float: right;
}

a {
  text-decoration: none;
}

a:link, a:visited {
  color: #8094d6;
}

a:hover, a:active {
  color: #FF9933;
}

footer {
  font-size: 9pt;
  font-style: italic;
  line-height: 12pt;
  text-align: center;
  padding-top: 30pt;
}
```

This might initially appear to be a lot of code, but if you look closely, you'll see that there isn't a lot of information on each line of code. It's fairly standard to place individual style rules on their own line, to help make style sheets more readable, but that is a personal preference; you could put all the rules on one line as long as you kept using the semicolon to separate each rule (more on that in a bit). Speaking of code readability, perhaps the first thing you noticed about this style sheet code is that it doesn't look anything

like normal HTML code. CSS uses a syntax all its own to specify style sheets.

Of course, the listing includes some familiar HTML tags (although not all tags require an entry in the style sheet). As you might guess, `body`, `h1`, `p`, `img`, `a`, and `footer` in the style sheet refer to the corresponding tags in the HTML documents to which the style sheet will be applied. The curly braces after each tag name describe how all content within that tag should appear.

In this case, the style sheet says that all body text should be rendered at a size of 10 points, in the Verdana font (if possible), and with the color black, with 14 points between lines. If the user does not have the Verdana font installed, the list of fonts in the style sheet represents the order in which the browser should search for fonts to use: Geneva, then Arial, and then Helvetica. If the user has none of those fonts, the browser uses whatever default sans-serif font is available. Additionally, the page should have left, right, and top padding of 5 points each.

Any text within an `<h1>` tag should be rendered in boldface Verdana at a size of 14 points. Moving on, any paragraph that uses only the `<p>` tag inherits all the styles indicated by the body element. However, if the `<p>` tag uses a special class named `subheader`, the text appears bold and in the color `#593d87` (a purple color).

The `pt` after each measurement in Listing 2.5 means *points* (there are 72 points in an inch). If you prefer, you can specify any style sheet measurement in inches (`in`), centimeters (`cm`), pixels (`px`), or "widths of a letter *m*," which are called ems (`em`).

You might have noticed that each style rule in the listing ends with a semicolon (`;`). Semicolons are used to separate style rules from each other. It is therefore customary to end each style rule with a semicolon so that you can easily add another style rule after it. Review the remainder of the style sheet in Listing 2.5 to see the presentation formatting applied to additional tags.

NOTE

You can specify font sizes as large as you like with style sheets, although some display devices and printers do not correctly handle fonts larger than 200 points.

To link this style sheet to HTML documents, include a `<link>` tag in the `<head>` section of each document. Listing 2.6 shows the HTML code for the page shown in Figure 2.9. It contains the following `<link>` tag:

Click here to view code image

```
<link rel="stylesheet" type="text/css" href="styles.css">
```

This assumes that the style sheet is stored under the name `styles.css` in the same folder as the HTML document. As long as the web browser supports style sheets—and all modern browsers do—the properties specified in the style sheet will apply to the content in the page without the need for any special HTML formatting code. This meets one of the goals of HTML, which is to provide a separation between the content in a web page and the specific formatting required to display that content.

LISTING 2.6 HTML Code for the Page Shown in Figure 2.9

Click here to view code image

```
<!DOCTYPE html>

<html lang="en">
  <head>
    <title>About BAWSI</title>
    <link rel="stylesheet" type="text/css" href="styles.css">
  </head>
  <body>
     <section>

     <header>
     <h1>About BAWSI</h1>
     </header>

     <p><img src="logo.gif" alt="BAWSI logo">The Bay Area Women's
     Sports Initiative (BAWSI) is a public benefit, nonprofit
     corporation with a mission to create programs and partnerships
     through which women athletes bring health, hope and wholeness
```

```
to
      our community. Founded in 2005 by Olympic and World Cup soccer
      stars Brandi Chastain and Julie Foudy and Marlene Bjornsrud,
      former general manager of the San Jose CyberRays women's
      professional soccer team, BAWSI provides a meaningful path for
      women athletes to become a more visible and valued part of the
      Bay Area sports culture.</p>

      <p class="subheader">BAWSI's History</p>

      <p>The concept of BAWSI was inspired by one of the most
      spectacular achievements in women's sports history and born
out
      of one its biggest disappointments… </p>

      <p><a href="secondpage.html">[continue reading]</a></p>
      </section>


      <footer>
      Copyright &copy; 2005-2017 BAWSI (www.bawsi.org).
      All rights reserved.  Used with permission.
      </footer>
  </body>
</html>
```

TIP

In most web browsers, you can view the style rules in a style sheet by opening the `.css` file and choosing Notepad or another text editor as the helper application to view the file. (To determine the name of the `.css` file, look at the HTML source of any web page that links to it.) To edit your own style sheets, just use a text editor.

The code in Listing 2.6 is interesting because it contains no formatting of any kind. In other words, nothing in the HTML code dictates how the text and images are to be displayed—no colors, no fonts, nothing. Yet the page is carefully formatted and rendered to the screen, thanks to the link to the external style sheet, `styles.css`. The real benefit to this approach is that you can easily create a site with multiple pages that maintains a consistent look and feel. And you have the benefit of isolating the visual style of the

page to a single document (the style sheet) so that one change impacts all pages.

NOTE

Not every browser's support of CSS is flawless. To find out how major browsers compare to each other in terms of CSS support, take a look at these websites: http://www.quirksmode.org/css/contents.html and http://caniuse.com.

▼ TRY IT YOURSELF

Create a Style Sheet of Your Own

Starting from scratch, create a new text document called `mystyles.css` and add some style rules for the following basic HTML tags: `<body>`, `<p>`, `<h1>`, and `<h2>`. After creating your style sheet, make a new HTML file that contains these basic tags. Play around with different style rules and see for yourself how simple it is to change entire blocks of text in paragraphs with one simple change in a style sheet file.

A CSS Style Primer

You now have a basic knowledge of CSS style sheets and how they are based on style rules that describe the appearance of information in web pages. The next few sections of this chapter provide a quick overview of some of the most important style properties and enable you to get started using CSS in your own style sheets.

CSS includes various style properties that are used to control fonts, colors, alignment, and margins, to name just a few. The style properties in CSS can be generally grouped into two major categories:

- Layout properties, which consist of properties that affect the positioning of elements on a web page, such as margins, padding, and alignment
- Formatting properties, which consist of properties that affect the visual display of elements within a website, such as the font type, size, and color

Basic Layout Properties

CSS layout properties determine how content is placed on a web page. One of the most important layout properties is the `display` property, which describes how an element is displayed with respect to other elements. The `display` property has four basic values:

- **`block`**—The element is displayed on a new line, as in a new paragraph.
- **`list-item`**—The element is displayed on a new line with a list-item mark (bullet) next to it.
- **`inline`**—The element is displayed inline with the current paragraph.
- **`none`**—The element is not displayed; it is hidden.

NOTE

The `display` property relies on a concept known as *relative positioning*, which means that elements are positioned relative to the location of other elements on a page. CSS also supports *absolute positioning*, which enables you to place an element at an exact location on a page, independent of other elements. You'll learn more about both of these types of positioning in Chapter 3, "Understanding the CSS Box Model and Positioning."

Understanding the `display` property is easier if you visualize each element on a web page occupying a rectangular area when displayed—the `display` property controls the manner in which this rectangular area is displayed. For example, the `block` value results in the element being placed on a new line by itself, whereas the `inline` value places the

element next to the content just before it. The `display` property is one of the few style properties that can be applied in most style rules. Following is an example of how to set the `display` property:

```
display: block;
```

You control the size of the rectangular area for an element with the `width` and `height` properties. As with many size-related CSS properties, `width` and `height` property values can be specified in several different units of measurement:

- **`in`**—Inches
- **`cm`**—Centimeters
- **`mm`**—Millimeters
- **`%`**—Percentage
- **`px`**—Pixels
- **`pt`**—Points

You can mix and match units however you choose within a style sheet, but it's generally a good idea to be consistent across a set of similar style properties. For example, you might want to stick with points for font properties and pixels for dimensions. Following is an example of setting the width of an element using pixel units:

```
width: 200px;
```

Basic Formatting Properties

CSS formatting properties are used to control the appearance of content on a web page, as opposed to controlling the physical positioning of the content. One of the most popular formatting properties is the `border` property, which establishes a visible boundary around an element with a box or partial box. Note that a border is always present in that space is always left for it, but the border does not appear in a way that you can see unless you give it properties that make it visible (like a color). The

following border properties provide a means of describing the borders of an element:

- **`border-width`**—The width of the border edge
- **`border-color`**—The color of the border edge
- **`border-style`**—The style of the border edge
- **`border-left`**—The left side of the border
- **`border-right`**—The right side of the border
- **`border-top`**—The top of the border
- **`border-bottom`**—The bottom of the border
- **`border`**—All the border sides

The `border-width` property establishes the width of the border edge. It is often expressed in pixels, as the following code demonstrates:

```
border-width: 5px;
```

Not surprisingly, the `border-color` and `border-style` properties set the border color and style. Following is an example of how these two properties are set:

```
border-color: blue;
border-style: dotted;
```

The `border-style` property can be set to any of the following basic values:

- **`solid`**—A single-line border
- **`double`**—A double-line border
- **`dashed`**—A dashed border
- **`dotted`**—A dotted border
- **`groove`**—A border with a groove appearance
- **`ridge`**—A border with a ridge appearance
- **`inset`**—A border with an inset appearance

- **`outset`**—A border with an outset appearance
- **`none`**—No border
- **`hidden`**—Effectively the same as `none` in that no border is displayed, but if two elements are next to each other with collapsed space between them, `hidden` ensures that a collapsed visible border does not show within the area of the element with a hidden border.

The default value of the `border-style` property is `none`, which is why elements don't have a border *unless* you set the `border` property to a different style. Although `solid` is the most common border style, you will also see the other styles in use.

The `border-left`, `border-right`, `border-top`, and `border-bottom` properties enable you to set the border for each side of an element individually. If you want a border to appear the same on all four sides, you can use the single `border` property by itself, which expects the following styles separated by a space: `border-width`, `border-style`, and `border-color`. Following is an example of using the `border` property to set a border that consists of two (double) red lines that are a total of 10 pixels in width:

```
border: 10px double red;
```

Whereas the color of an element's border is set with the `border-color` property, the color of the inner region of an element is set using the `color` and `background-color` properties. The `color` property sets the color of text in an element (foreground), and the `background-color` property sets the color of the background behind the text. Following is an example of setting both color properties to predefined colors:

```
color: black;
background-color: orange;
```

You can also assign custom colors to these properties by specifying the colors in hexadecimal or as RGB (Red, Green, Blue) decimal values:

```
background-color: #999999;
color: rgb(0,0,255);
```

You can also control the alignment and indentation of web page content without too much trouble. This is accomplished with the `text-align` and `text-indent` properties, as the following code demonstrates:

```
text-align: center;
text-indent: 12px;
```

When you have an element properly aligned and indented, you might be interested in setting its font. The following basic font properties set the various parameters associated with fonts:

- **`font-family`**—The family of the font
- **`font-size`**—The size of the font
- **`font-style`**—The style of the font (normal or italic)
- **`font-weight`**—The weight of the font (normal, lighter, bold, bolder, and so on)

The `font-family` property specifies a prioritized list of font family names. A prioritized list is used instead of a single value to provide alternatives in case a font isn't available on a given system. The `font-size` property specifies the size of the font using a unit of measurement, often in points. Finally, the `font-style` property sets the style of the font, and the `font-weight` property sets the weight of the font. Following is an example of setting these font properties:

Click here to view code image

```
font-family: Arial, sans-serif;
font-size: 36pt;
font-style: italic;
font-weight: normal;
```

Now that you know a whole lot more about style properties and how they work, refer to Listing 2.5 and see whether it makes a bit more sense. Here's a recap of the style properties used in that style sheet, which you can use as a guide for understanding how it works:

- **`font`**—Lets you set many font properties at once. You can specify a list of font names separated by commas; if the first is not available, the

next is tried, and so on. You can also include the words bold and/or italic and a font size. Alternatively, you can set each of these `font` properties separately with `font-family`, `font-size`, `font-weight`, and `font-style`.

- **`line-height`**—Also known in the publishing world as *leading*. This sets the height of each line of text, usually in points.
- **`color`**—Sets the text color using the standard color names or hexadecimal color codes.
- **`text-decoration`**—Is useful for turning off link underlining; simply set it to `none`. The values of `underline`, `italic`, and `line-through` are also supported.
- **`text-align`**—Aligns text to the left, right, or center, along with justifying the text with a value of `justify`.
- **`padding`**—Adds padding to the left, right, top, and bottom of an element; this padding can be in measurement units or a percentage of the page width. Use `padding-left` and `padding-right` if you want to add padding to the left and right of the element independently. Use `padding-top` or `padding-bottom` to add padding to the top or bottom of the element, as appropriate. You'll learn a bit more about these style properties in Chapter 3.

Using Style Classes

Whenever you want some of the text on your pages to look different from the other text, you can create what amounts to a custom-built HTML tag. Each type of specially formatted text you define is called a *style class*, which is a custom set of formatting specifications that can be applied to any element in a web page.

Before showing you a style class, I need to take a quick step back and clarify some CSS terminology. First off, a CSS *style property* is a specific style that you can assign a value, such as `color` or `font-size`. You associate a style property and its respective value with elements on a web page by using a selector. A *selector* is used to identify tags on a page to

which you apply styles. Following is an example of a selector, a property, and a value all included in a basic style rule:

```
h1 { font: 36pt Courier; }
```

In this code, `h1` is the selector, `font` is the style property, and `36pt Courier` is the value. The selector is important because it means that the font setting will be applied to all `h1` elements in the web page. But maybe you want to differentiate between some of the `h1` elements—what then? The answer lies in style classes.

Suppose you want two different kinds of `<h1>` headings for use in your documents. You create a style class for each one by putting the following CSS code in a style sheet:

Click here to view code image

```
h1.silly { font: 36pt 'Comic Sans'; }
h1.serious { font: 36pt Arial; }
```

Notice that these selectors include a period (`.`) after `h1`, followed by a descriptive class name. To choose between the two style classes, use the `class` attribute, like this:

Click here to view code image

```
<h1 class="silly">Marvin's Munchies Inc. </h1>
<p>Text about Marvin's Munchies goes here. </p>
```

Or you could use this:

Click here to view code image

```
<h1 class="serious">MMI Investor Information</h1>
<p>Text for business investors goes here.</p>
```

When referencing a style class in HTML code, simply specify the class name in the `class` attribute of an element. In the preceding example, the words `Marvin's Munchies Inc.` would appear in a 36-point Comic Sans font, assuming that you included a `<link>` to the style sheet at the top of the web page and that the user has the Comic Sans font installed. The words `MMI Investor Information` would appear in the 36-point

Arial font instead. You can see another example of classes in action in Listing 2.5; look for the `specialtext` `<p>` class.

What if you want to create a style class that can be applied to any element instead of just headings or some other particular tag? In your CSS, simply use a period (`.`) followed by any style class name you make up and any style specifications you choose. That class can specify any number of font, spacing, and margin settings all at once. Wherever you want to apply your custom tag in a page, just use an HTML tag plus the `class` attribute, followed by the class name you created.

For example, the style sheet in Listing 2.5 includes the following style class specification:

```
p.specialtext {
  font-weight: bold;
  color: #593d87;
}
```

This style class is applied in Listing 2.6 with the following tag:

```
<p class="specialtext">
```

TIP

You might have noticed a change in the coding style when a style rule includes multiple properties. For style rules with a single style, you'll commonly see the property placed on the same line as the rule, like this:

Click here to view code image

```
p.specialtext { font-weight: bold; }
```

However, when a style rule contains multiple style properties, it's much easier to read and understand the code if you list the properties one per line, like this:

```
p.specialtext {
  font-weight: bold;
  color: #593d87;
}
```

Everything between that tag and the closing tag in Listing 2.6 appears in bold purple text.

If no element selector were present in your style sheet, meaning that the rule looked like this:

```
.specialtext {
  font-weight: bold;
  color: #593d87;
}
```

Then any element could refer to `specialtext` and have the text rendered as bold purple, not just a element.

What makes style classes so valuable is how they isolate style code from web pages, effectively enabling you to focus your HTML code on the actual content in a page, not on how it is going to appear on the screen. Then you can focus on how the content is rendered to the screen by fine-tuning the style sheet. You might be surprised by how a relatively small amount of code in a style sheet can have significant effects across an entire website. This makes your pages much easier to maintain and manipulate.

Using Style IDs

When you create custom style classes, you can use those classes as many times as you like—they are not unique. However, in some instances, you want precise control over unique elements for layout or formatting purposes (or both). In such instances, look to ID*s* instead of classes.

A *style ID* is a custom set of formatting specifications that can be applied to only one element in a web page. You can use IDs across a set of pages, but only once per time within each page.

For example, suppose you have a title within the body of all your pages. Each page has only one title, but all the pages themselves include one instance of that title. Following is an example of a selector with an ID indicated, plus a property and a value:

Click here to view code image

```
p#title {font: 24pt Verdana, Geneva, Arial, sans-serif}
```

Notice that this selector includes a hash mark, or pound sign (#), after p, followed by a descriptive ID name. When referencing a style ID in HTML code, simply specify the ID name in the id attribute of an element, like so:

Click here to view code image

```
<p id="title">Some Title Goes Here</p>
```

Everything between the opening and closing <p> tags will appear in 24-point Verdana text—but only once on any given page. You often see style IDs used to define specific parts of a page for layout purposes, such as a header area, footer area, main body area, and so on. These types of areas in a page appear only once per page, so using an ID rather than a class is the appropriate choice.

Internal Style Sheets and Inline Styles

In some situations, you want to specify styles that will be used in only one web page. You can enclose a style sheet between <style> and </style> tags and include it directly in an HTML document. Style sheets used in this manner must appear in the <head> of an HTML document. No <link> tag is needed, and you cannot refer to that style sheet from any other page (unless you copy it into the beginning of that document too). This kind of style sheet is known as an internal style sheet, as you learned earlier in the chapter.

Listing 2.7 shows an example of how you might specify an internal style sheet.

LISTING 2.7 A Web Page with an Internal Style Sheet

Click here to view code image

```
<!DOCTYPE html>

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

```
    <title>Some Page</title>

    <style type="text/css">
      footer {
        font-size: 9pt;
        line-height: 12pt;
        text-align: center;
      }
    </style>
  </head>
  <body>
  …
    <footer>
    Copyright 2017 Acme Products, Inc.
    </footer>
  </body>
</html>
```

In the listing code, the `footer` style class is specified in an internal style sheet that appears in the head of the page. The style class is now available for use within the body of this page, and only within this page. In fact, it is used in the body of the page to style the copyright notice.

Internal style sheets are handy if you want to create a style rule that is used multiple times within a single page. However, in some instances, you might need to apply a unique style to one particular element. This calls for an inline style rule, which enables you to specify a style for only a small part of a page, such as an individual element. For example, you can create and apply a style rule within a `<p>`, `<div>`, or `<span>` tag via the `style` attribute. This type of style is known as an *inline style* because it is specified right there in the middle of the HTML code.

NOTE

The `<span>` and `</span>` tags are dummy tags that do nothing in and of themselves except specify a range of content to apply any `style` attributes that you add. The only difference between `<div>` and `<span>` is that `<div>` is a block element and, therefore, forces a line break, whereas `<span>` is an inline element and doesn’t force a break. Therefore, you

should use to modify the style of any portion of text that is to appear in the middle of a sentence or paragraph without any line break.

Here's how a sample style attribute might look:

Click here to view code image

```
<p style="color:green">
  This text is green, but <span style="color:red">this text is
  red.</span>
  Back to green again, but…
</p>
<p>
  …now the green is over, and we're back to the default color
  for this page.
</p>
```

This code makes use of the tag to show how to apply the color style property in an inline style rule. In fact, both the <p> tag and the tag in this example use the color property as an inline style. What's important to understand is that the color:red style property overrides the color:green style property for the text between the and tags. Then, in the second paragraph, neither of the color styles applies because it is a completely new paragraph that adheres to the default color of the entire page.

CAUTION

Using inline styles isn't considered a best practice when used beyond page-level debugging or beyond trying out new things in a controlled setting. The best practice of all is having your pages link to a centrally maintained style sheet so that changes are immediately reflected in all pages that use it.

Validate Your Style Sheets

Just as it is important to validate your HTML markup, it is important to validate your style sheet. You can find a specific validation tool for CSS at http://jigsaw.w3.org/css-validator/. You can point the tool to a web address,

upload a file, or paste content into the form field provided. The ultimate goal is a result like the one in Figure 2.11: `valid`!

FIGURE 2.11
The W3C CSS Validator shows there are no errors in the style sheet contents of Listing 2.5.

Summary

This chapter introduced the basics of what web pages are and how they work. You learned that coded HTML commands are included in a text file, and you saw that typing HTML text yourself is better than using a graphical editor to create HTML commands for you—especially when you're learning HTML.

You were introduced to the most basic and important HTML tags. By adding these coded commands to any plain-text document, you can quickly transform it into a bona fide web page. You learned that the first step in creating a web page is to put a few obligatory HTML tags at the beginning and end, including adding a title for the page. You can then mark where paragraphs and lines end and add horizontal rules and headings, if you want them. You also got a taste of some of the semantic tags in HTML5, which are used to provide additional meaning by delineating the types of content your pages contain (not just the content itself). Table 2.1 summarizes the basic HTML tags introduced in this chapter.

Beyond HTML, you learned that a style sheet can control the appearance of many HTML pages at once. It can also give you extremely precise control over the typography, spacing, and positioning of HTML elements. You also learned that, by adding a `style` attribute to almost any HTML tag, you can control the style of any part of an HTML page without referring to a separate style sheet document.

You learned about three main approaches to including style sheets in your website: a separate style sheet file with the extension `.css` that is linked to in the `<head>` of your documents, a collection of style rules placed in the head of the document within the `<style>` tag, and rules placed directly in an HTML tag via the `style` attribute (although the latter is not a best practice for long-term use). Table 2.2 summarizes tags with attributes discussed in this chapter.

TABLE 2.1 HTML Tags Covered in this Chapter

Tag	Function
`<html>...</html>`	Encloses the entire HTML document.
`<head>...</head>`	Encloses the head of the HTML document. Used within the `<html>` tag pair.
`<title>...</title>`	Indicates the title of the document. Used within the `<head>` tag pair.

`<body>…</body>`	Encloses the body of the HTML document. Used within the `<html>` tag pair.
`<p>…</p>`	Encloses a paragraph; skips a line between paragraphs.
` `	Indicates a line break.
`<hr>`	Displays a horizontal rule line.
`<h1>…</h1>`	Encloses a first-level heading.
`<h2>…</h2>`	Encloses a second-level heading.
`<h3>…</h3>`	Encloses a third-level heading.
`<h4>…</h4>`	Encloses a fourth-level heading.
`<h5>…</h5>`	Encloses a fifth-level heading.
`<h6>…</h6>`	Encloses a sixth-level heading.
`<header>…</header>`	Contains introductory information.
`<footer>…</footer>`	Contains supplementary material for its containing element (commonly a copyright notice or author information).
`<nav>…</nav>`	Contains navigational elements.
`<section>…</section>`	Contains thematically similar content, such as a chapter of a book or a section of a page.
`<article>…</article>`	Contains content that is a standalone body of work, such as a news article.
`<aside>…</aside>`	Contains secondary information for its containing element.
`<address>…</address>`	Contains address information related to its nearest `<article>` or `<body>` element, often contained within a `<footer>` element.

TABLE 2.2 HTML Tags with Attributes Covered in This Chapter

Tag/Attributes	Function
Tag	
`<a>`	Indicates a hyperlink to a position in the current document or to another document.
Attributes	
`href="url"`	The address of the linked content.
Tag	
`<style>…</style>`	Allows an internal style sheet to be included within a document. Used between `<head>` and `</head>`.
Attribute	
`type="contenttype"`	The Internet content type. (Always `"text/css"` for a CSS style sheet.)
Tag	
`<link>`	Links to an external style sheet (or other document type). Used in the `<head>` section of the document.
Attributes	
`href="url"`	The address of the style sheet.
`type="contenttype"`	The Internet content type. (Always `"text/css"` for a CSS style sheet.)
`rel="stylesheet"`	The relationship to a referenced document. (Always `"stylesheet"` for style sheets.)
Tag	
`<span>…</span>`	Does nothing but provide a place to put `style` or other attributes. (Similar to `<div>…</div>`, but does not cause a line break.)
Attribute	

`style="style"`	Includes inline style specifications. (Can be used in `<span>`, `<div>`, `<body>`, and most other HTML tags.)

Q&A

Q. I've created a web page, but when I open the file in my web browser, I see all the text, including the HTML tags. Sometimes I even see weird gobbledygook characters at the top of the page. What did I do wrong?

A. You didn't save the file as plain text. Try saving the file again, being careful to save it as Text Only or ASCII Text. If you can't quite figure out how to get your word processor to do that, don't stress. Just type your HTML files in Notepad or TextEdit instead, and everything should work just fine. (Also, always make sure that the filename of your web page ends in `.html` or `.htm`.)

Q. I've seen web pages on the Internet that don't have `<!DOCTYPE>` or `<html>` tags at the beginning. You said pages always have to start with these tags. What's the deal?

A. Many web browsers will forgive you if you forget to include the `<!DOCTYPE>` or `<html>` tag and will display the page correctly anyway. However, it's a very good idea to include it because some software does need it to identify the page as valid HTML. Besides, you want your pages to be bona fide HTML pages so that they conform to the latest web standards.

Q. Do I have to use semantic markup at all? Didn't you say throughout this lesson that pages are valid with or without it?

A. True, none of these elements is required for a valid HTML document. You don't have to use any of them, but I urge you to think beyond the use of markup for visual display only and think about it for semantic meaning as well. Visual display is meaningless to screen readers, but

semantic elements convey a ton of information through these machines.

Q. Say I link a style sheet to my page that says all text should be blue, but there's a `<span style="font-color:red">` tag in the page somewhere. Will that text display as blue or red?

A. Red. Local inline styles always take precedence over external style sheets. Any style specifications you put between `<style>` and `</style>` tags at the top of a page also take precedence over external style sheets (but not over inline styles later in the same page). This is the cascading effect of style sheets that I mentioned earlier in the chapter. You can think of cascading style effects as starting with an external style sheet, which is overridden by an internal style sheet, which is overridden by inline styles.

Q. Can I link more than one style sheet to a single page?

A. Sure. For example, you might have a sheet for formatting (text, fonts, colors, and so on) and another one for layout (margins, padding, alignment, and so on). Just be sure to include a `<link>` for both. Technically, the CSS standard requires web browsers to give the user the option to choose between style sheets when multiple sheets are presented via multiple `<link>` tags. However, in practice, all major web browsers simply include every style sheet unless it has a `rel="alternate"` attribute. The preferred technique for linking in multiple style sheets involves using the special `@import` command. The following is an example of importing multiple style sheets with `@import`:

```
@import url(styles1.css);
@import url(styles2.css);
```

Similar to the `<link>` tag, the `@import` command must be placed in the head of a web page.

Workshop

The Workshop contains quiz questions and exercises to help you solidify your understanding of the material covered. Try to answer all questions before looking at the "Answers" section that follows.

Quiz

1. Which five tags does every HTML5 page require?
2. Which of the semantic elements discussed in this chapter is appropriate for containing the definition of a word used in an article?
3. Do you have to use an `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>`, or `<h6>` element within a `<header>` element?
4. How many different `<nav>` elements can you have in a single page?
5. How many different ways are there to ensure that style rules can be applied to your content?

Answers

1. Every HTML page requires `<html>`, `<head>`, `<title>`, and `<body>` (along with their closing tags, `</html>`, `</head>`, `</title>`, and `</body>`, respectively), plus `<!DOCTYPE html>` on the very first line.
2. The `<aside>` element is appropriate for this situation.
3. No. The `<header>` element can contain any other flow content besides another `<header>` element or a `<footer>` element. However, a heading element (`<h1>` through `<h6>`) is not required in a `<header>` element.
4. You can have as many `<nav>` elements as you need. The trick is to "need" only a few (perhaps for primary and secondary navigation only); otherwise, the meaning is lost.
5. Three: externally, internally, and inline.

Exercises

- Even if your main goal in reading this book is to create web content for your business, you might want to make a personal web page just for practice. Type a few paragraphs to introduce yourself to the world, and use the HTML tags you learned in this chapter to make them into a web page.
- Throughout the book, you'll be following along with the code examples and making pages of your own. Take a moment now to set up a basic document template containing the document type declaration and tags for the core HTML document structure. That way, you can be ready to copy and paste that information whenever you need it.
- Develop a standard style sheet for your website, and link it into all your pages. (Use internal style sheets and/or inline styles for pages that need to deviate from it.) If you work for a corporation, chances are it has already developed font and style specifications for printed materials. Get a copy of those specifications and follow them for company web pages too.

CHAPTER 3

Understanding the CSS Box Model and Positioning

What You'll Learn in This Chapter:

- How to conceptualize the CSS box model
- How to position your elements
- How to control the way elements stack up
- How to manage the flow of text
- How fixed layouts work
- How fluid layouts work
- How to create a fixed/fluid hybrid layout
- How to think about and begin to implement a responsive design

In the preceding chapter, you learned a lot about the basic structure and syntax of HTML and CSS. In this chapter, you'll take that knowledge one step further and learn about the CSS box model, which is the guiding force behind the layout of elements on your screen—be it a desktop or mobile display.

It's important to spend some time focusing on and practicing working with the box model, because if you have a good handle on how the box model

works, you won't tear your hair out when you create a design and then realize that the elements don't line up or that they seem a little "off." You'll know that, in almost all cases, something—the margin, the padding, the border—just needs a little tweaking.

You'll also learn more about CSS positioning, including stacking elements on top of each other in a three-dimensional way (instead of a vertical way), as well as controlling the flow of text around elements using the `float` property. You'll then build on this information and learn about the types of overall page layouts: fixed and fluid (also referred to as liquid). But it's also possible to use a combination of the two, with some elements fixed and others fluid. We end the chapter with a brief mention of responsive web design, which is an important topic upon which entire books have been written.

The CSS Box Model

Every element in HTML is considered a "box," whether it is a paragraph, a `<div>`, an image, or anything else. Boxes have consistent properties, whether we see them or not, and whether the style sheet specifies them or not. They're always present, and as designers, we have to keep their presence in mind when creating a layout.

Figure 3.1 is a diagram of the box model. The box model describes the way in which every HTML block-level element has the potential for a border, padding, and margin and, specifically, how the border, padding, and margin are applied. In other words, all elements have some padding between the content and the border of the element. Additionally, the border might or might not be visible, but there is space for it, just as there is a margin between the border of the element and any other content outside the element.

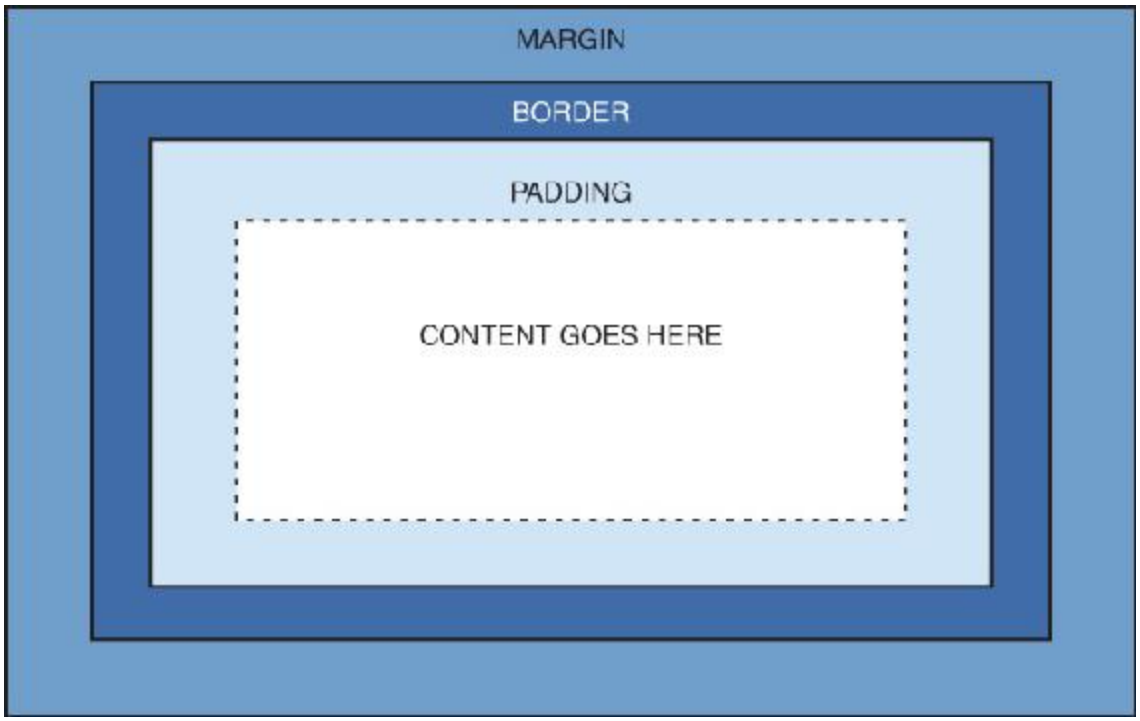

FIGURE 3.1
Every element in HTML is represented by the CSS box model.

Here's yet another explanation of the box model, going from the outside inward:

- The *margin* is the area outside the element. It never has color; it is always transparent.
- The *border* extends around the element, on the outer edge of any padding. The border can be of several types, widths, and colors.
- The *padding* exists around the content and inherits the background color of the content area.
- The *content* is surrounded by padding.

Here's where the tricky part comes in: To know the true height and width of an element, you have to take all the elements of the box model into account. Think back to the example from the preceding chapter: Despite the specific indication that a `<div>` should be 250 pixels wide and 100 pixels high, that `<div>` had to grow larger to accommodate the padding in use.

You already know how to set the width and height of an element using the `width` and `height` properties. The following example shows how to define a `<div>` that is 250 pixels wide and 100 pixels high, with a red background and a black single-pixel border:

```
div {
  width: 250px;
  height: 100px;
  background-color: #ff0000;
  border: 1px solid #000000;
}
```

Figure 3.2 shows this simple `<div>`.

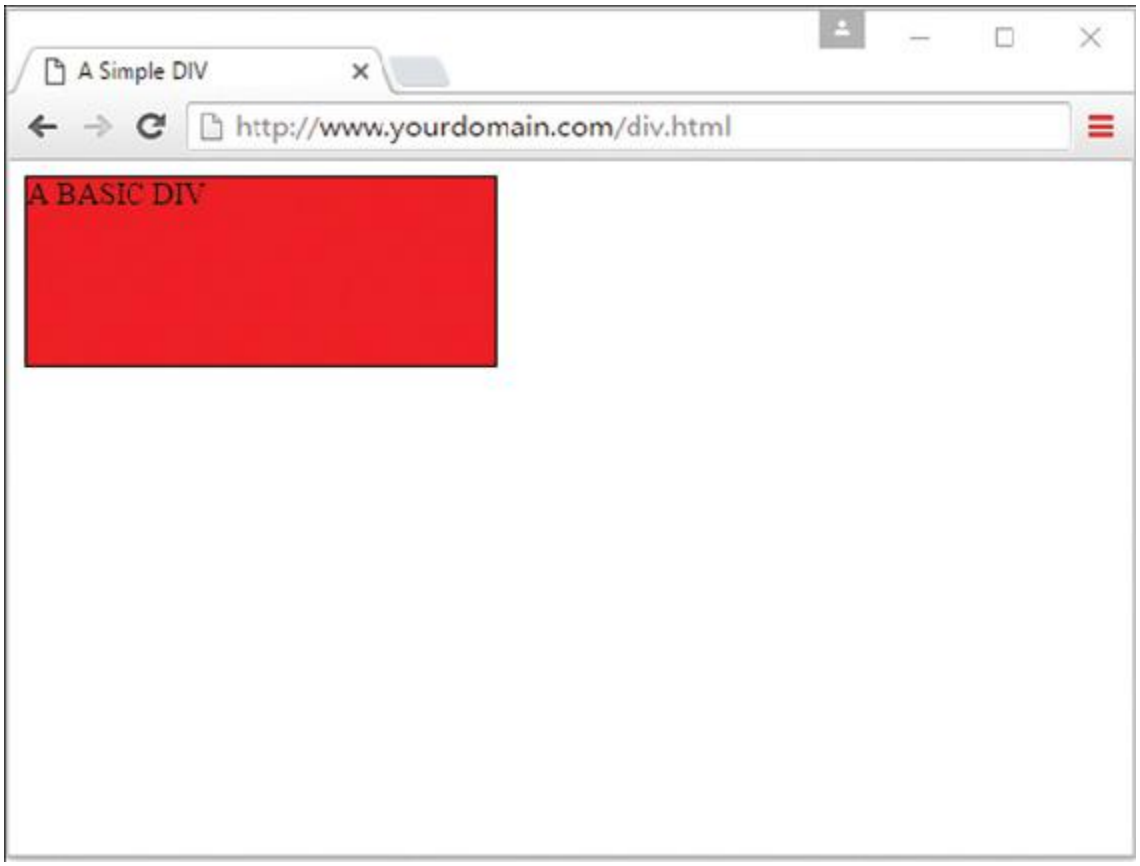

FIGURE 3.2
This is a simple styled `<div>`.

If we define a second element with these same properties, but also add `margin` and `padding` properties of a certain size, we begin to see how the size of the element changes. This is because of the box model.

The second is defined as follows, just adding 10 pixels of margin and 10 pixels of padding to the element:

```
div#d2 {
  width: 250px;
  height: 100px;
  background-color: #ff0000;
  border: 5px solid #000000;
  margin: 10px;
  padding: 10px;
}
```

The second , shown in Figure 3.3, is defined as the same height and width as the first one, but the overall height and width of the entire box surrounding the element itself is much larger when margins and padding are put in play.

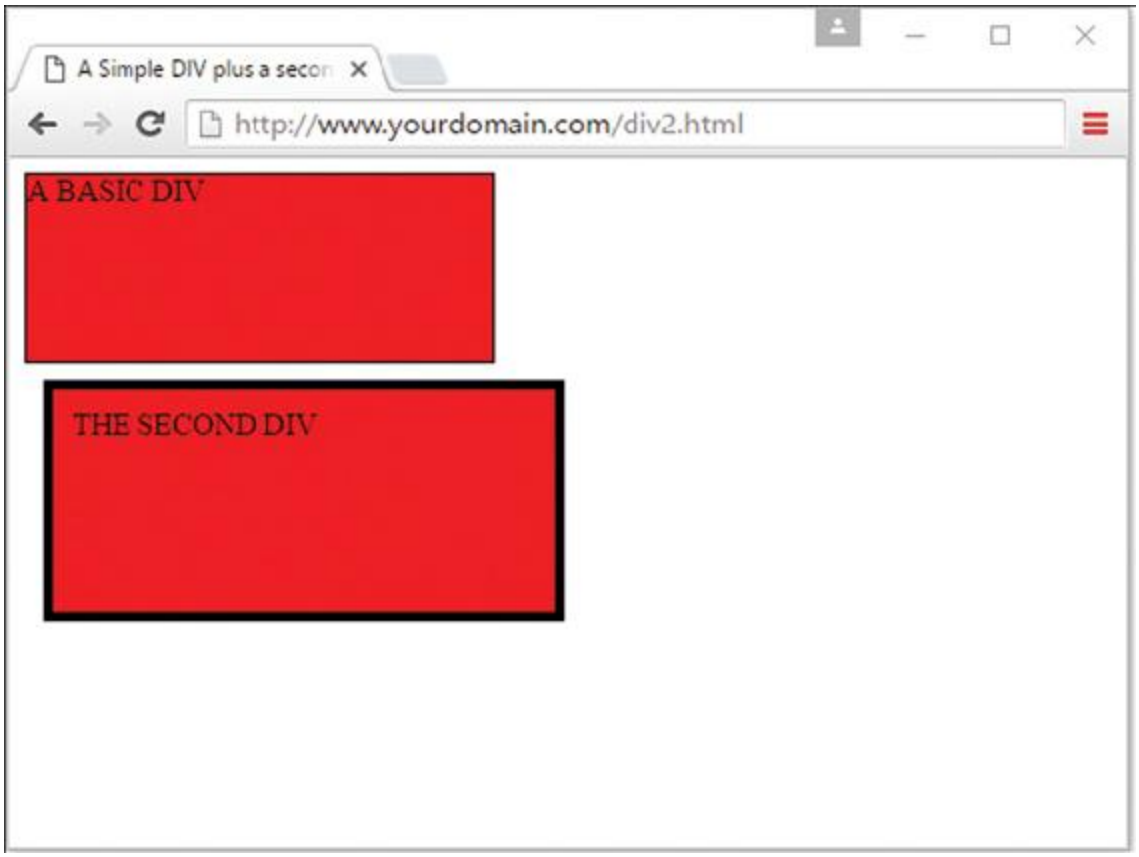

FIGURE 3.3
This is supposed to be another simple , but the box model affects the size of the second .

The total *width* of an element is the sum of the following:

Click here to view code image

```
width + padding-left + padding-right + border-left + border-right +
margin-left + margin-right
```

The total *height* of an element is the sum of the following:

Click here to view code image

```
height + padding-top + padding-bottom + border-top + border-bottom
+
margin-top + margin-bottom
```

Therefore, the second `<div>` has an actual width of 300 (250 + 10 + 10 + 5 + 5 + 10 + 10) and an actual height of 150 (100 + 10 + 10 + 5 + 5 + 10 + 10).

By now, you can begin to see how the box model affects your design. Let's say that you have only 250 pixels of horizontal space, but you would like 10 pixels of margin, 10 pixels of padding, and 5 pixels of border on all sides. To accommodate what you would like with what you have room to display, you must specify the `width` of your `<div>` as only 200 pixels so that 200 + 10 + 10 + 5 + 5 + 10 + 10 adds up to that 250 pixels of available horizontal space.

The mathematics of the model are important as well. In dynamically driven sites or sites in which user interactions drive the client-side display (such as through JavaScript events), your server-side or client-side code could draw and redraw container elements on the fly. In other words, your code will produce the numbers, but you have to provide the boundaries.

Now that you've been introduced to the way of the box model, keep it in mind throughout the rest of the work you do in this book and in your web design. Among other things, it will affect element positioning and content flow, which are the two topics we tackle next.

The Whole Scoop on Positioning

Relative positioning is the default type of positioning HTML uses. You can think of relative positioning as being akin to laying out checkers on a checkerboard: The checkers are arranged from left to right, and when you get to the edge of the board, you move on to the next row. Elements that are styled with the `block` value for the `display` style property are automatically placed on a new row, whereas `inline` elements are placed on the same row immediately next to the element preceding them. As an example, `<p>` and `<div>` tags are considered block elements, whereas the `<span>` tag is considered an inline element.

The other type of positioning CSS supports is known as *absolute positioning* because it enables you to set the exact position of HTML content on a page. Although absolute positioning gives you the freedom to spell out exactly where an element is to appear, the position is still relative to any parent elements that appear on the page. In other words, absolute positioning enables you to specify the exact location of an element's rectangular area with respect to its parent's area, which is very different from relative positioning.

With the freedom of placing elements anywhere you want on a page, you can run into the problem of overlap, when an element takes up space another element is using. Nothing is stopping you from specifying the absolute locations of elements so that they overlap. In this case, CSS relies on the z-index of each element to determine which element is on the top and which is on the bottom. You'll learn more about the z-index of elements later in this lesson. For now, let's look at exactly how you control whether a style rule uses relative or absolute positioning.

The type of positioning (relative or absolute) a particular style rule uses is determined by the `position` property, which is capable of having one of the following four values:

- **`static`**—The default positioning according to the normal flow of the content
- **`relative`**—The element is positioned relative to its normal position, using offset properties discussed below

- **absolute**—The element is positioned relative to its nearest ancestor element, or according to the normal flow of the page if no ancestor is present
- **fixed**—The element is fixed relative to the viewport—this type of positioning is used for images that scroll along with the page (as an example)

After specifying the type of positioning, you provide the specific position using the following properties:

- **left**—The left position offset
- **right**—The right position offset
- **top**—The top position offset
- **bottom**—The bottom position offset

You might think that these position properties make sense only for absolute positioning, but they actually apply to relative and fixed positioning as well. For example, under relative positioning, the position of an element is specified as an offset relative to the original position of the element. So if you set the `left` property of an element to `25px`, the left side of the element shifts over 25 pixels from its original (relative) position. An absolute position, on the other hand, is specified relative to the ancestor element to which the style is applied. So if you set the `left` property of an element to `25px` under absolute positioning, the left side of the element appears 25 pixels to the right of the ancestor element's left edge. On the other hand, using the `right` property with the same value positions the element so that its *right* side is 25 pixels to the right of the ancestor's *right* edge.

Let's return to the color-blocks example to see how positioning works. In Listing 3.1, the four color blocks have relative positioning specified. As you can see in Figure 3.4, the blocks are positioned vertically.

LISTING 3.1 Showing Relative Positioning with Four Color Blocks

Click here to view code image

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title>Positioning the Color Blocks</title>
   <style type="text/css">
   div {
     position: relative;
     width: 250px;
     height: 100px;
     border: 5px solid #000;
     color: black;
     font-weight: bold;
     text-align: center;
    }
    div#d1 {
     background-color: #ff0000;
    }
    div#d2 {
     background-color: #00ff00;
    }
    div#d3 {
     background-color: #0000ff;
    }
    div#d4 {
     background-color: #ffff00;
    }
   </style>
</head>
<body>
  <div id="d1">DIV #1</div>
  <div id="d2">DIV #2</div>
  <div id="d3">DIV #3</div>
  <div id="d4">DIV #4</div>
</body>
</html>
```

The style sheet entry for the `<div>` element itself sets the `position` style property for the `<div>` element to `relative`. Because the remaining style rules are inherited from the `<div>` style rule, they inherit its relative positioning. In fact, the only difference between the other style rules is that they have different background colors.

Notice in Figure 3.4 that the `<div>` elements are displayed one after the next, which is what you would expect with relative positioning. But to

make things more interesting, which is what we're here to do, you can change the positioning to absolute and explicitly specify the placement of the colors. In Listing 3.2, the style sheet entries are changed to use absolute positioning to arrange the color blocks.

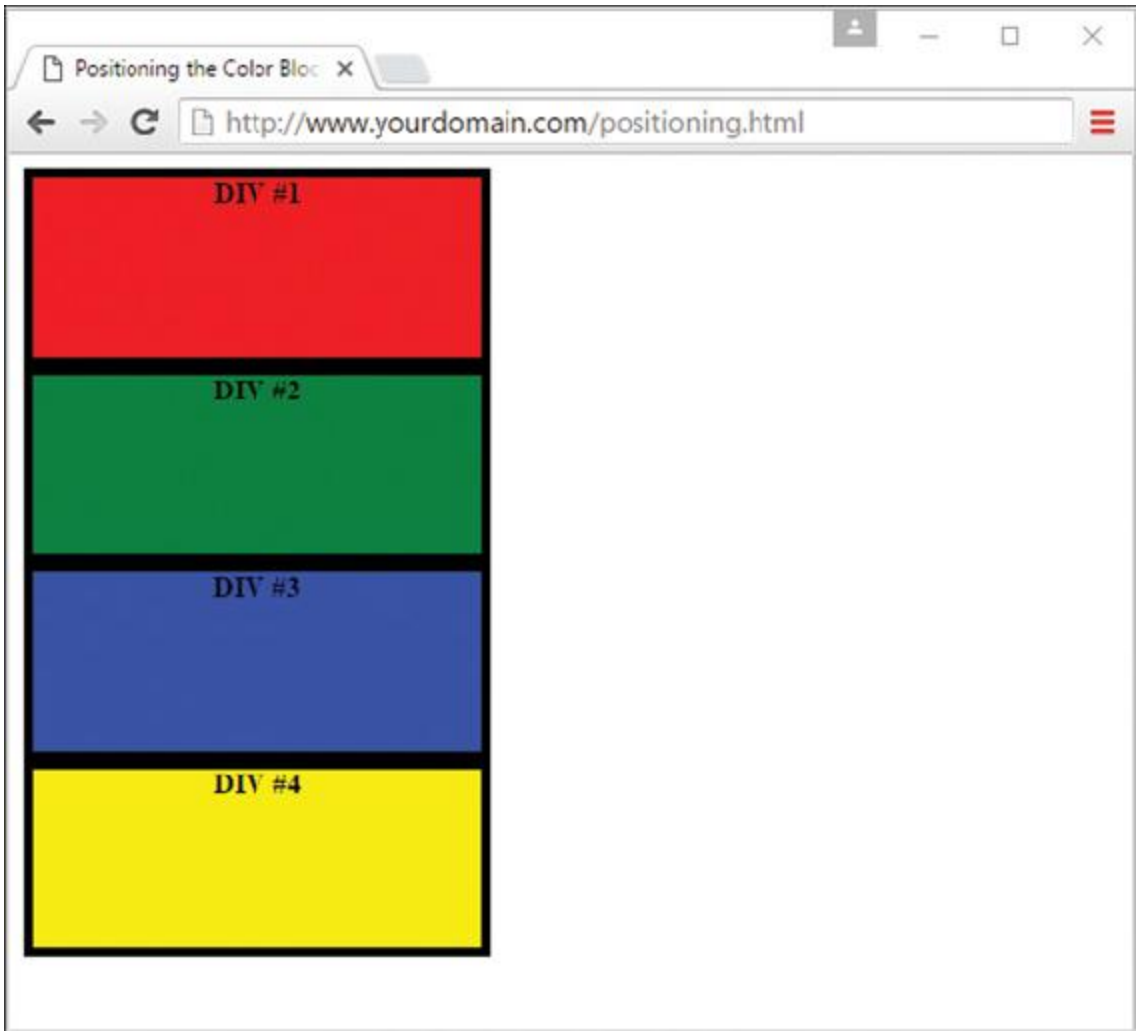

FIGURE 3.4
The color blocks are positioned vertically, with one on top of the other.

LISTING 3.2 Using Absolute Positioning of the Color Blocks

Click here to view code image

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title>Positioning the Color Blocks</title>
```

```
    <style type="text/css">
    div {
     position: absolute;
     width: 250px;
     height: 100px;
     border: 5px solid #000;
     color: black;
     font-weight: bold;
     text-align: center;
    }
    div#d1 {
     background-color: #ff0000;
     left: 0px;
     top: 0px;
    }
    div#d2 {
     background-color: #00ff00;
     left: 75px;
     top: 25px;
    }
    div#d3 {
     background-color: #0000ff;
     left: 150px;
     top: 50px;
    }
    div#d4 {
     background-color: #ffff00;
     left: 225px;
     top: 75px;
    }
    </style>
</head>
<body>
     <div id="d1">DIV #1</div>
     <div id="d2">DIV #2</div>
     <div id="d3">DIV #3</div>
     <div id="d4">DIV #4</div>
</body>
</html>
```

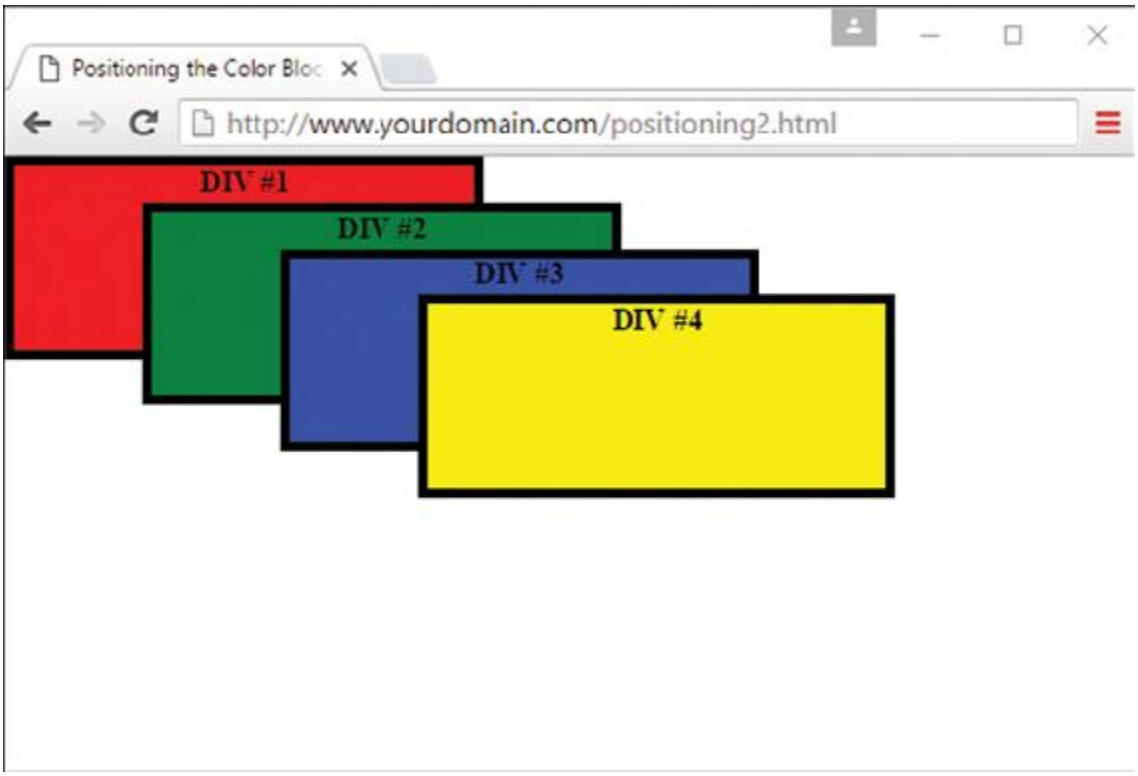

FIGURE 3.5
The color blocks are displayed using absolute positioning.

This style sheet sets the `position` property to `absolute`, which is necessary for the style sheet to use absolute positioning. Additionally, the `left` and `top` properties are set for each of the inherited `<div>` style rules. However, the position of each of these rules is set so that the elements are displayed overlapping each other, as Figure 3.5 shows.

Now we're talking layout! Figure 3.5 shows how absolute positioning enables you to place elements exactly where you want them. It also reveals how easy it is to arrange elements so that they overlap. You might be curious about how a web browser knows which elements to draw on top when they overlap. The next section covers how you can control stacking order.

Controlling the Way Things Stack Up

In certain situations, you want to carefully control the manner in which elements overlap each other on a web page. The `z-index` style property

enables you to set the order of elements with respect to how they stack on top of each other. The name *z-index* might sound a little strange, but it refers to the notion of a third dimension (Z) that points into the computer screen, in addition to the two dimensions that go across (X) and down (Y) the screen. Another way to think of the z-index is to consider the relative position of a single magazine within a stack of magazines. A magazine nearer the top of the stack has a higher z-index than a magazine lower in the stack. Similarly, an overlapped element with a higher value for its `z-index` is displayed on top of an element with a lower value for its `z-index`.

The `z-index` property is used to set a numeric value that indicates the relative `z-index` of a style rule. The number assigned to `z-index` has meaning only with respect to other style rules in a style sheet, which means that setting the `z-index` property for a single rule doesn't mean much. On the other hand, if you set `z-index` for several style rules that apply to overlapped elements, the elements with higher `z-index` values appear on top of elements with lower `z-index` values.

NOTE

Regardless of the `z-index` value you set for a style rule, an element displayed with the rule will always appear on top of its parent.

Listing 3.3 contains another version of the color-blocks style sheet and HTML that uses `z-index` settings to alter the natural overlap of elements.

LISTING 3.3 Using `z-index` to Alter the Display of Elements in the Color-Blocks Example

Click here to view code image

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title>Positioning the Color Blocks</title>
   <style type="text/css">
   div {
```

```
     position: absolute;
     width: 250px;
     height: 100px;
     border: 5px solid #000;
     color: black;
     font-weight: bold;
     text-align: center;
    }
    div#d1 {
     background-color: #ff0000;
     left: 0px;
     top: 0px;
     z-index: 0;
    }
    div#d2 {
     background-color: #00ff00;
     left: 75px;
     top: 25px;
     z-index: 3;
    }
    div#d3 {
     background-color: #0000ff;
     left: 150px;
     top: 50px;
     z-index: 2;
    }
   div#d4 {
     background-color: #ffff00;
     left: 225px;
     top: 75px;
     z-index: 1;
    }
   </style>
</head>
<body>
  <div id="d1">DIV #1</div>
  <div id="d2">DIV #2</div>
  <div id="d3">DIV #3</div>
  <div id="d4">DIV #4</div>
</body>
</html>
```

The only change in this code from what you saw in Listing 3.2 is the addition of the `z-index` property in each of the numbered `div` style classes. Notice that the first numbered `div` has a `z-index` setting of `0`, which should make it the lowest element in terms of the z-index, whereas the second `div` has the highest `z-index`. Figure 3.6 shows the color-

blocks page as displayed with this style sheet, which clearly shows how the z-index affects the displayed content and makes it possible to carefully control the overlap of elements.

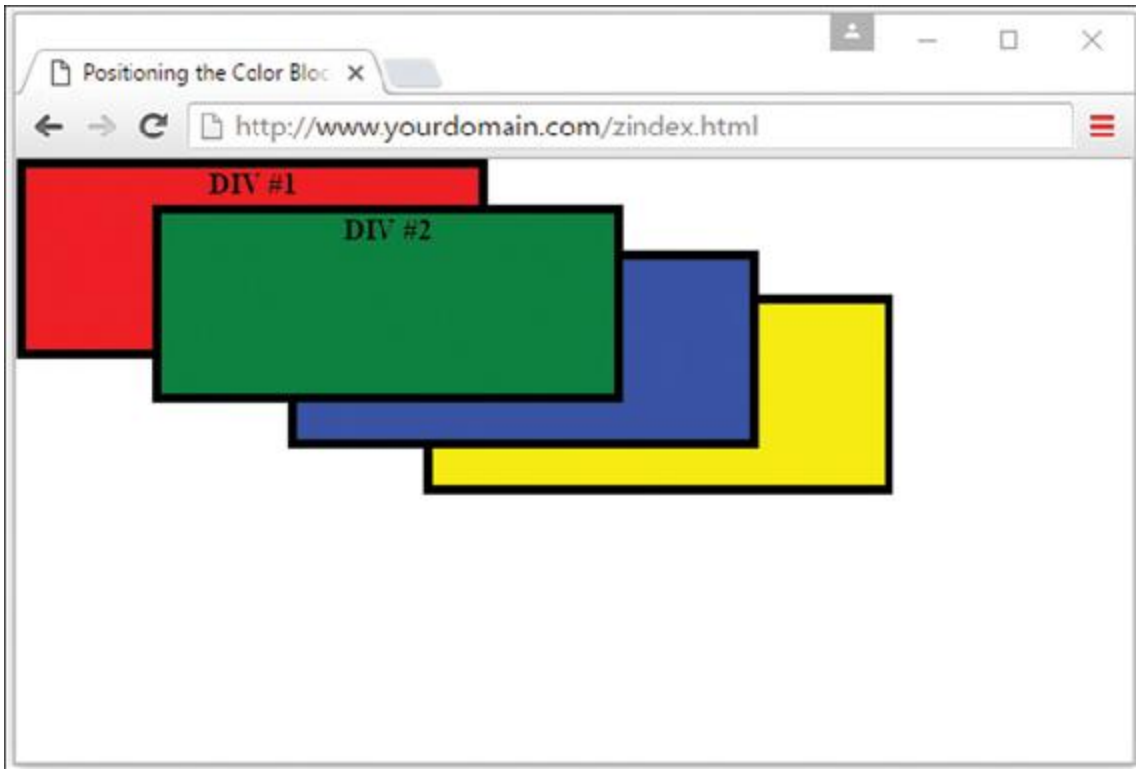

FIGURE 3.6
Using `z-index` to alter the display of the color blocks.

Although the examples show color blocks that are simple `<div>` elements, the `z-index` style property can affect any HTML content, including images.

Managing the Flow of Text

Now that you've seen some examples of placing elements relative to other elements or placing them absolutely, it's time to revisit the flow of content around elements. The conceptual *current line* is an invisible line used to place elements on a page. This line has to do with the flow of elements on a page; it comes into play as elements are arranged next to each other across and down the page. Part of the flow of elements is the flow of text on a

page. When you mix text with other elements (such as images), it's important to control how the text flows around those other elements.

Following are some style properties that give you control over text flow:

- **`float`**—Determines how text flows around an element
- **`clear`**—Stops the flow of text around an element
- **`overflow`**—Controls the overflow of text when an element is too small to contain all the text

The `float` property controls how text flows around an element. It can be set to either `left` or `right`. These values determine where to position an element with respect to flowing text. So setting an image's `float` property to `left` positions the image to the left of flowing text.

As you learned in the preceding chapter, you can prevent text from flowing next to an element by using the `clear` property, which you can set to `none`, `left`, `right`, or `both`. The default value for the `clear` property is `none`, indicating that text is to flow with no special considerations for the element. The `left` value causes text to stop flowing around an element until the left side of the page is free of the element. Likewise, the `right` value means that text is not to flow around the right side of the element. The `both` value indicates that text isn't to flow around either side of the element.

The `overflow` property handles overflow text, which is text that doesn't fit within its rectangular area; this can happen if you set the `width` and `height` of an element too small. The `overflow` property can be set to `visible`, `hidden`, or `scroll`. The `visible` setting automatically enlarges the element so that the overflow text fits within it; this is the default setting for the property. The `hidden` value leaves the element the same size, allowing the overflow text to remain hidden from view. Perhaps the most interesting value is `scroll`, which adds scrollbars to the element so that you can move around and see the text.

Understanding Fixed Layouts

A fixed layout, or fixed-width layout, is just that: a layout in which the body of the page is set to a specific width. That width is typically controlled by a master "wrapper" element that contains all the content. The `width` property of a wrapper element, such as a `<div>`, is set in the style sheet entry if the `<div>` was given an ID value such as `main` or `wrapper` (although the name is up to you).

When you're creating a fixed-width layout, the most important decision is determining the minimum screen resolution you want to accommodate. For many years, 800×600 was the "lowest common denominator" for web designers, resulting in a typical fixed width of approximately 760 pixels. However, the number of people using 800×600 screen resolution for non-mobile browsers is now less than 4%. Given that, many web designers consider 1,024×768 the current minimum screen resolution, so if they create fixed-width layouts, the fixed width typically is somewhere between 800 and 1,000 pixels wide.

CAUTION

Remember, the web browser window contains nonviewable areas, including the scrollbar. So if you are targeting a 1,024-pixel-wide screen resolution, you really can't use all 1,024 of those pixels.

A main reason for creating a fixed-width layout is so that you can have precise control over the appearance of the content area. However, if users visit your fixed-width site with smaller or much larger screen sizes or resolutions than the size or resolution you had in mind while you designed it, they will encounter scrollbars (if their size or resolution is smaller) or a large amount of empty space (if their size or resolution is greater). Finding fixed-width layouts is difficult among the most popular websites these days because site designers know they need to cater to the largest possible audience (and therefore make no assumptions about browser size). However, fixed-width layouts still have wide adoption, especially by site administrators using a content management system with a strict template.

The following figures show one such site, for San Jose State University (university websites commonly use a strict template and content management system, so this was an easy example to find); it has a wrapper element fixed at 960 pixels wide. In Figure 3.7, the browser window is a shade under 900 pixels wide. On the right side of the image, important content is cut off (and at the bottom of the figure, a horizontal scrollbar displays in the browser).

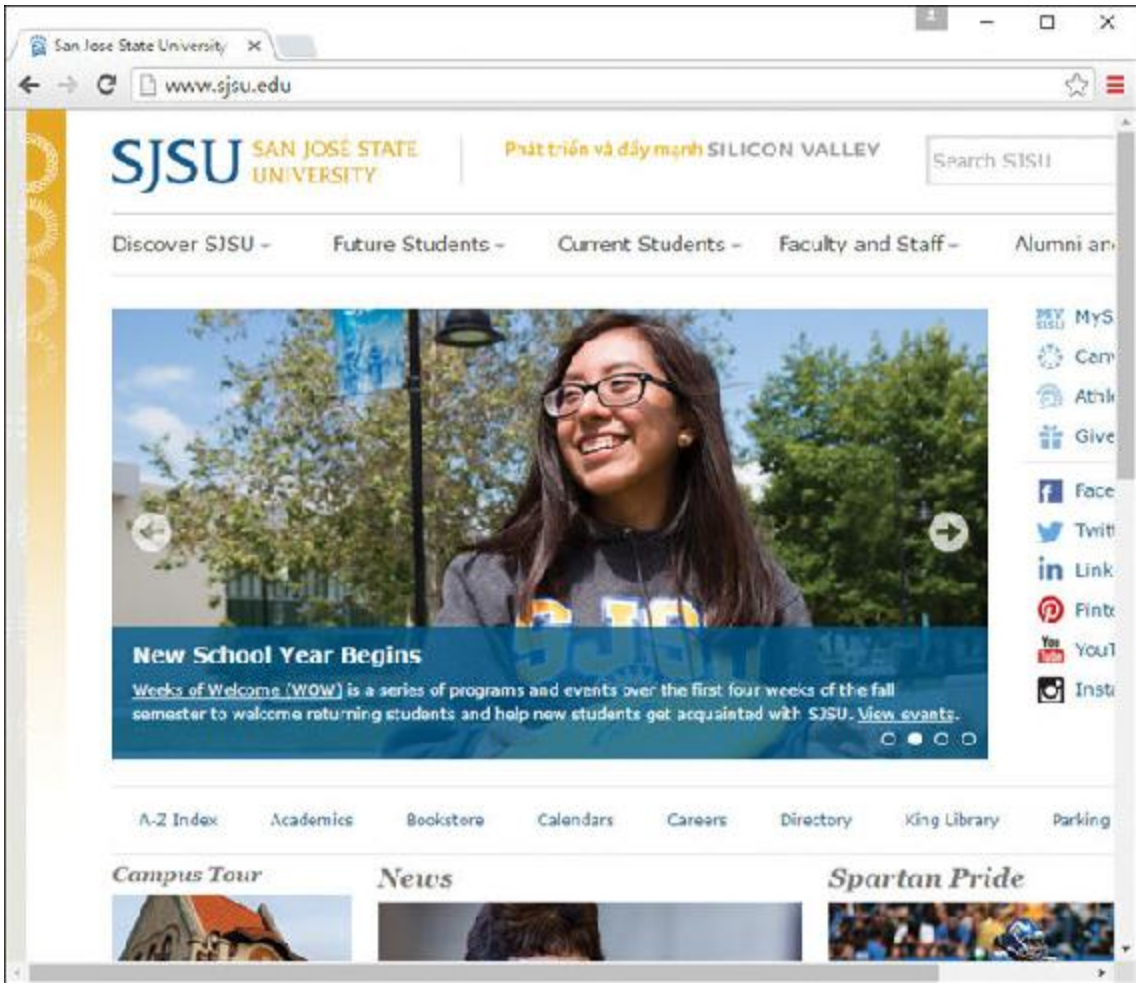

FIGURE 3.7
A fixed-width example with a smaller screen size.

However, Figure 3.8 shows how this site looks when the browser window is more than 1,300 pixels wide: You see a lot of empty space (or “real estate”) on both sides of the main body content, which some consider aesthetically displeasing.

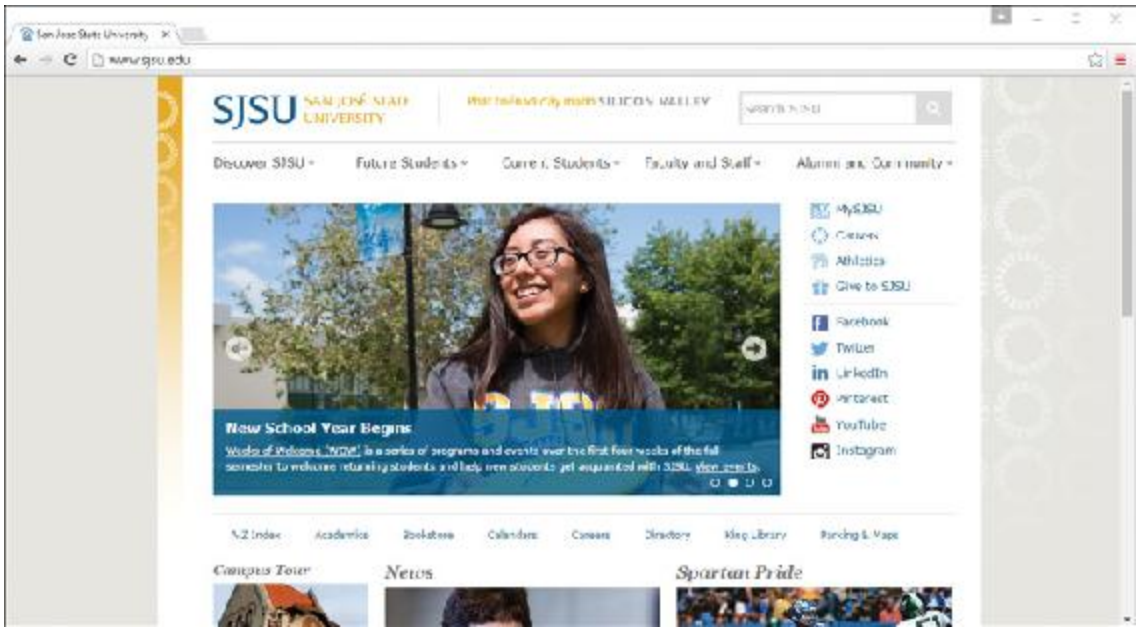

FIGURE 3.8
A fixed-width example with a larger screen size.

Besides the decision to create a fixed-width layout in the first place is the task of determining whether to place the fixed-width content flush left or center it. Placing the content flush left produces extra space on the right side only; centering the content area creates extra space on both sides. However, centering at least provides balance, whereas a flush-left design could end up looking like a small rectangle shoved in the corner of the browser, depending on the size and resolution of a user's monitor.

Understanding Fluid Layouts

A fluid layout—also called a *liquid* layout—is one in which the body of the page does not use a specified width in pixels, although it might be enclosed in a master "wrapper" element that uses a percentage width. The idea behind the fluid layout is that it can be perfectly usable and still retain the overall design aesthetic even if the user has a very small or very wide screen.

Figures 3.9, 3.10, and 3.11 show three examples of a fluid layout in action.

In Figure 3.9, the browser window is approximately 745 pixels wide. This example shows a reasonable minimum screen width before a horizontal scrollbar appears. In fact, the scrollbar does not appear until the browser is 735 pixels wide. On the other hand, Figure 3.10 shows a very small browser window (less than 600 pixels wide).

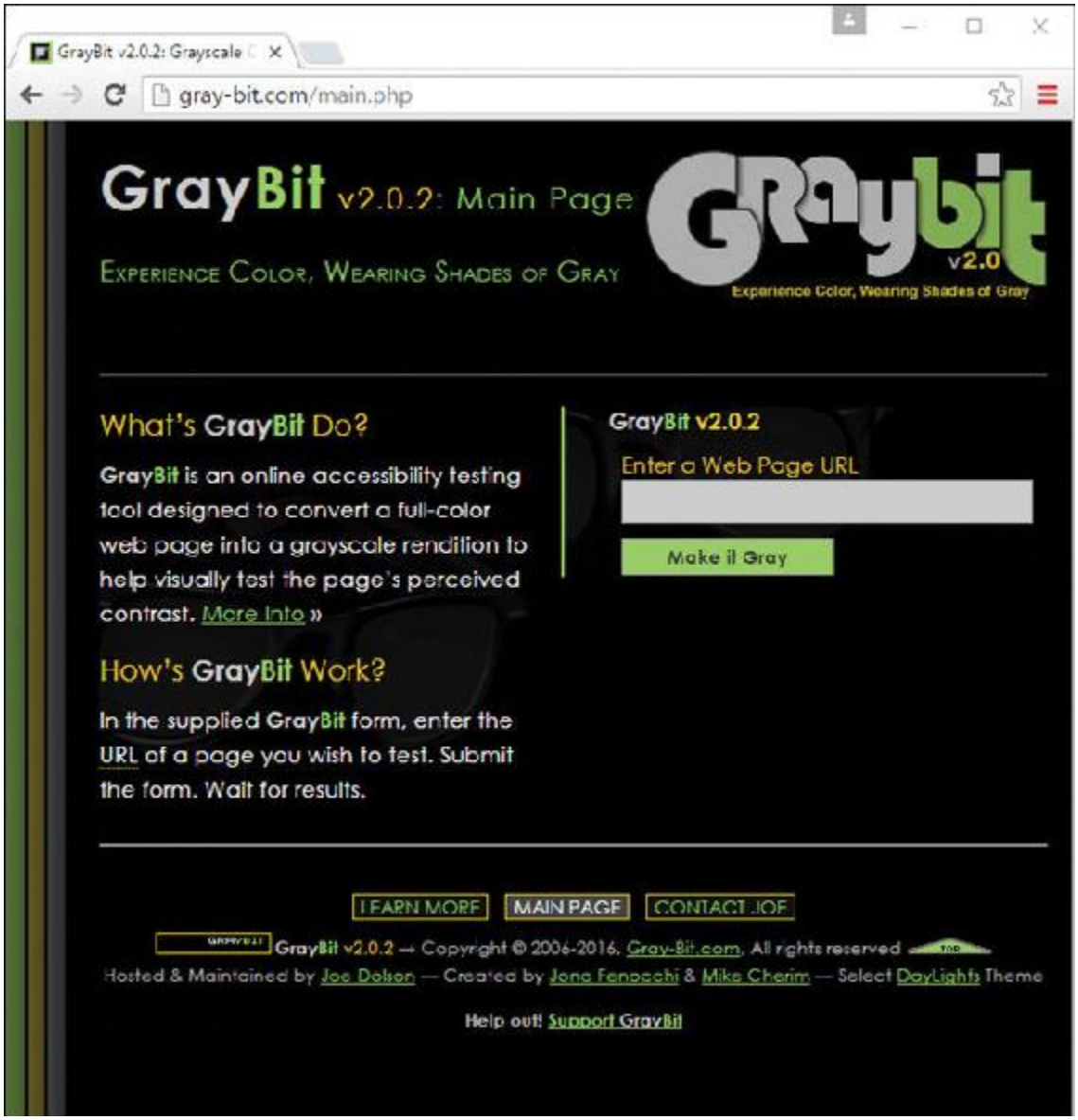

FIGURE 3.9
A fluid layout as viewed in a relatively small screen.

In Figure 3.10, you can see a horizontal scrollbar; in the header area of the page content, the logo graphic is beginning to take over the text and appear

on top of it. But the bulk of the page is still quite usable. The informational content on the left side of the page is still legible and is sharing the available space with the input form on the right side.

Figure 3.11 shows how this same page looks in a very wide screen. In Figure 3.11, the browser window is approximately 1,200 pixels wide. There is plenty of room for all the content on the page to spread out. This fluid layout is achieved because all the design elements have a percentage width specified (instead of a fixed width). Thus, the layout makes use of all the available browser real estate.

The fluid layout approach might seem like the best approach at first glance —after all, who wouldn't want to take advantage of all the screen real estate available? But there's a fine line between taking advantage of space and not allowing the content to breathe. Too much content is overwhelming; not enough content in an open space is underwhelming.

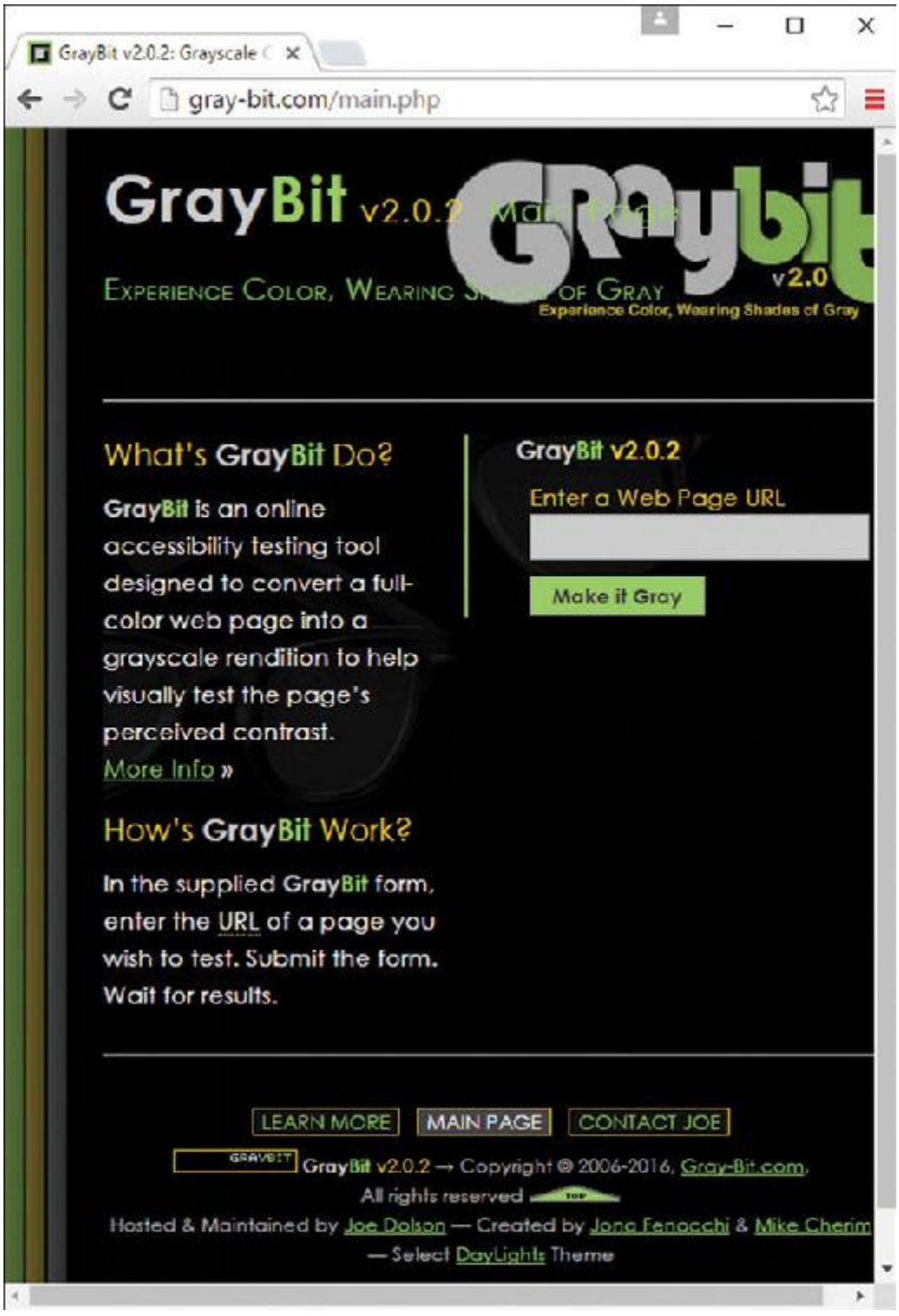

FIGURE 3.10
A fluid layout as viewed in a very small screen.

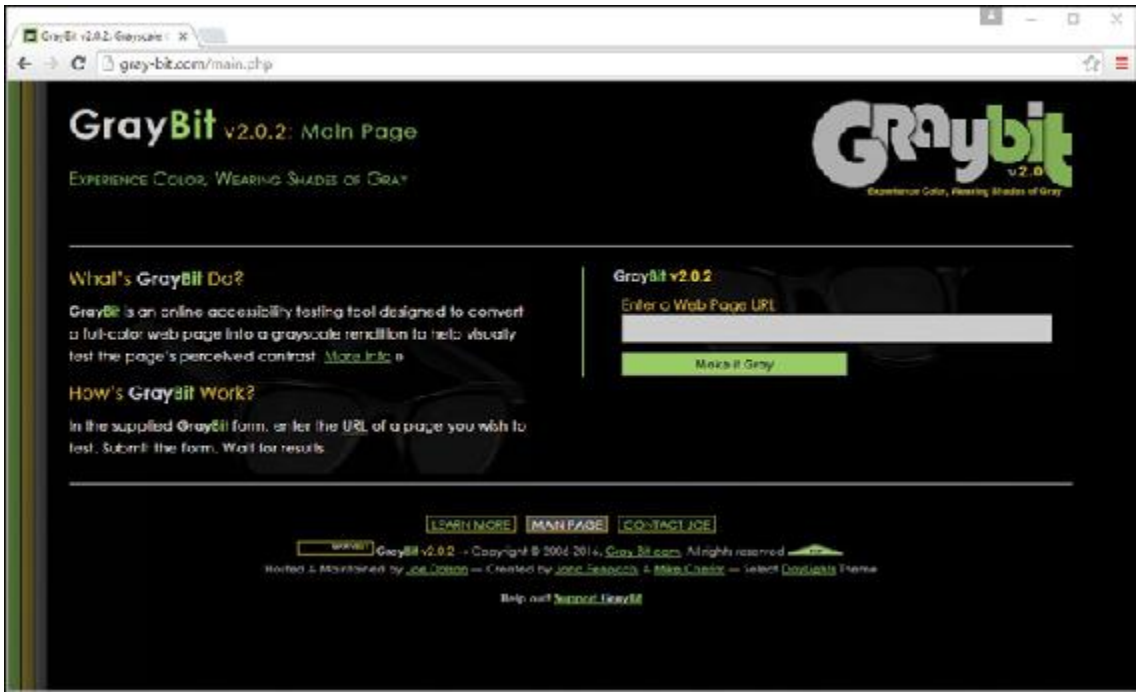

FIGURE 3.11
A fluid layout as viewed in a wide screen.

The pure fluid layout can be impressive, but it requires a significant amount of testing to ensure that it is usable in a wide range of browsers at varying screen resolutions. You might not have the time and effort to produce such a design; in that case, a reasonable compromise is the fixed/fluid hybrid layout, or a fully responsive design, as you'll learn about later in this chapter.

Creating a Fixed/Fluid Hybrid Layout

A fixed/fluid hybrid layout is one that contains elements of both types of layouts. For example, you could have a fluid layout that includes fixed-width content areas either within the body area or as anchor elements (such as a left-side column or as a top navigation strip). You can even create a fixed content area that acts like a frame, in which a content area remains fixed even as users scroll through the content.

Starting with a Basic Layout Structure

In this example, you'll learn to create a template that is fluid but with two fixed-width columns on either side of the main body area (which is a third column, if you think about it, only much wider than the others). The template also has a delineated header and footer area. Listing 3.4 shows the basic HTML structure for this layout.

LISTING 3.4 Basic Fixed/Fluid Hybrid Layout Structure

Click here to view code image

```
<!DOCTYPE html>

<html lang="en">
  <head>
     <title>Sample Layout</title>
     <link href="layout.css" rel="stylesheet" type="text/css">
  </head>

  <body>
    <header>HEADER</header>
    <div id="wrapper">
        <div id="content_area">CONTENT</div>
        <div id="left_side">LEFT SIDE</div>
        <div id="right_side">RIGHT SIDE</div>
    </div>
    <footer>FOOTER</footer>
  </body>
</html>
```

First, note that the style sheet for this layout is linked to with the `<link>` tag instead of included in the template. Because a template is used for more than one page, you want to be able to control the display elements of the template in the most organized way possible. This means you need to change the definitions of those elements in only one place—the style sheet.

Next, notice that the basic HTML is just that: extremely basic. Truth be told, this basic HTML structure can be used for a fixed layout, a fluid layout, or the fixed/fluid hybrid you see here because all the actual styling that makes a layout fixed, fluid, or hybrid happens in the style sheet.

With the HTML structure in Listing 3.4, you actually have an identification of the content areas you want to include in your site. This planning is crucial to any development; you have to know what you want to include

before you even think about the type of layout you are going to use, let alone the specific styles that will be applied to that layout.

NOTE

I am using elements with named identifiers in this example instead of the semantic elements such as `<section>` or `<nav>` because I'm illustrating the point in the simplest way possible without being prescriptive to the content itself. However, if you know that the `<div>` on the left side is going to hold navigation, you should use the `<nav>` tag instead of a `<div>` element with an `id` something like `left_side`—but this type of naming can become problematic as depending on repositioning that content might not end up displaying on the left side of anything, so best to name based on purpose rather than appearance.

At this stage, the `layout.css` file includes only this entry:

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

Using a `0` value for `margin` and `padding` allows the entire page to be usable for element placement.

If you look at the HTML in Listing 3.4 and say to yourself, "But those `<div>` elements will just stack on top of each other without any styles," you are correct. As shown in Figure 3.12, there is no layout to speak of.

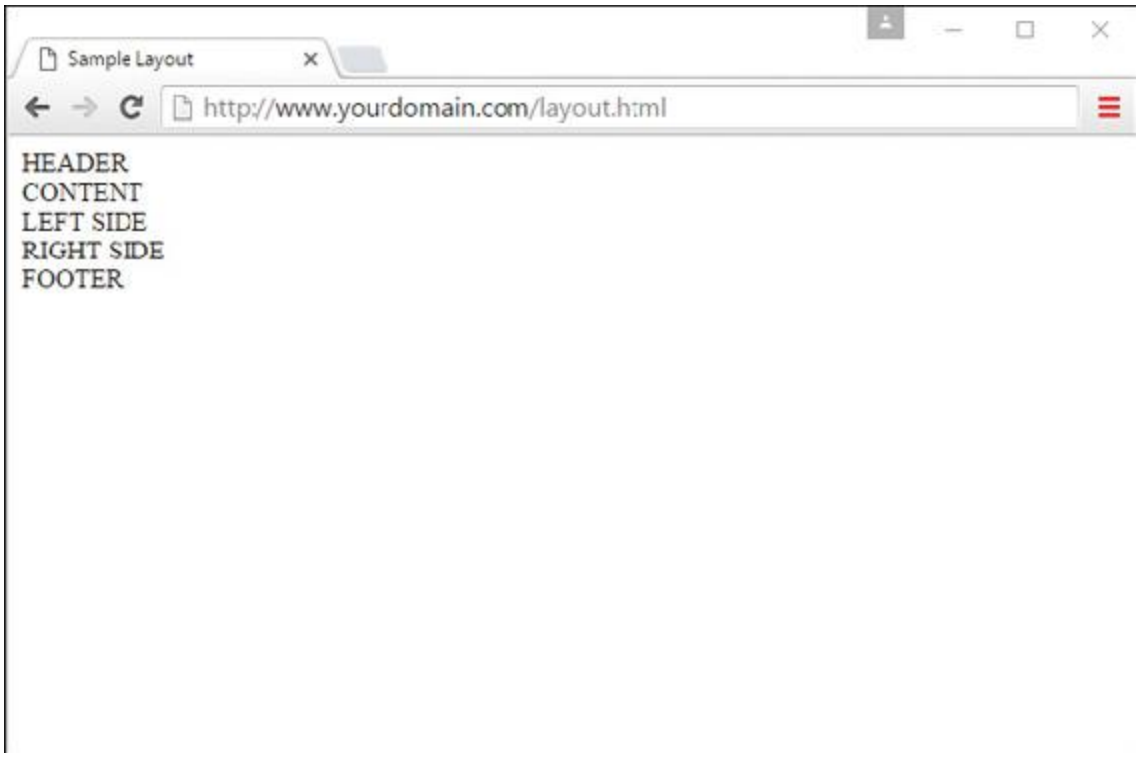

FIGURE 3.12
A basic HTML template with no styles applied to the container elements.

Defining Two Columns in a Fixed/Fluid Hybrid Layout

We can start with the easy things to get some styles and actual layout in there. Because this layout is supposed to be fluid, we know that whatever we put in the header and footer areas will extend the width of the browser window, regardless of how narrow or wide the window might be.

Adding the following code to the style sheet gives the header and footer area each a width of 100% as well as the same background color and text color:

```
header, footer {
  float: left;
  width: 100%;
  background-color: #7152f4;
  color: #ffffff;
}
```

Now things get a little trickier. We have to define the two fixed columns on either side of the page, plus the column in the middle. In the HTML we're using here, note that a `<div>` element, called `wrapper`, surrounds both. This element is defined in the style sheet as follows:

```
#wrapper {
  float: left;
  padding-left: 200px;
  padding-right: 125px;
}
```

The two padding definitions essentially reserve space for the two fixed-width columns on the left and right of the page. The column on the left will be 200 pixels wide, the column on the right will be 125 pixels wide, and each will have a different background color. But we also have to position the items relative to where they would be placed if the HTML remained unstyled (see Figure 3.12). This means adding `position: relative` to the style sheet entries for each of these columns. Additionally, we indicate that the `<div>` elements should float to the left.

But in the case of the `<div>` element `left_side`, we also indicate that we want the rightmost margin edge to be 200 pixels in from the edge (this is in addition to the column being defined as 200 pixels wide). We also want the margin on the left side to be a full negative margin; this will pull it into place (as you will soon see). The `<div>` element `right_side` does not include a value for `right`, but it does include a negative margin on the right side:

```
#left_side {
  position: relative;
  float: left;
  width: 200px;
  background-color: #52f471;
  right: 200px;
  margin-left: -100%;
}

#right_side {
  position: relative;
  float: left;
  width: 125px;
  background-color: #f452d5;
```

```
  margin-right: -125px;
}
```

At this point, let's also define the content area so that it has a white background, takes up 100% of the available area, and floats to the left relative to its position:

```
#content_area {
  position: relative;
  float: left;
  background-color: #ffffff;
  width: 100%;
}
```

At this point, the basic layout should look something like Figure 3.13, with the areas clearly delineated.

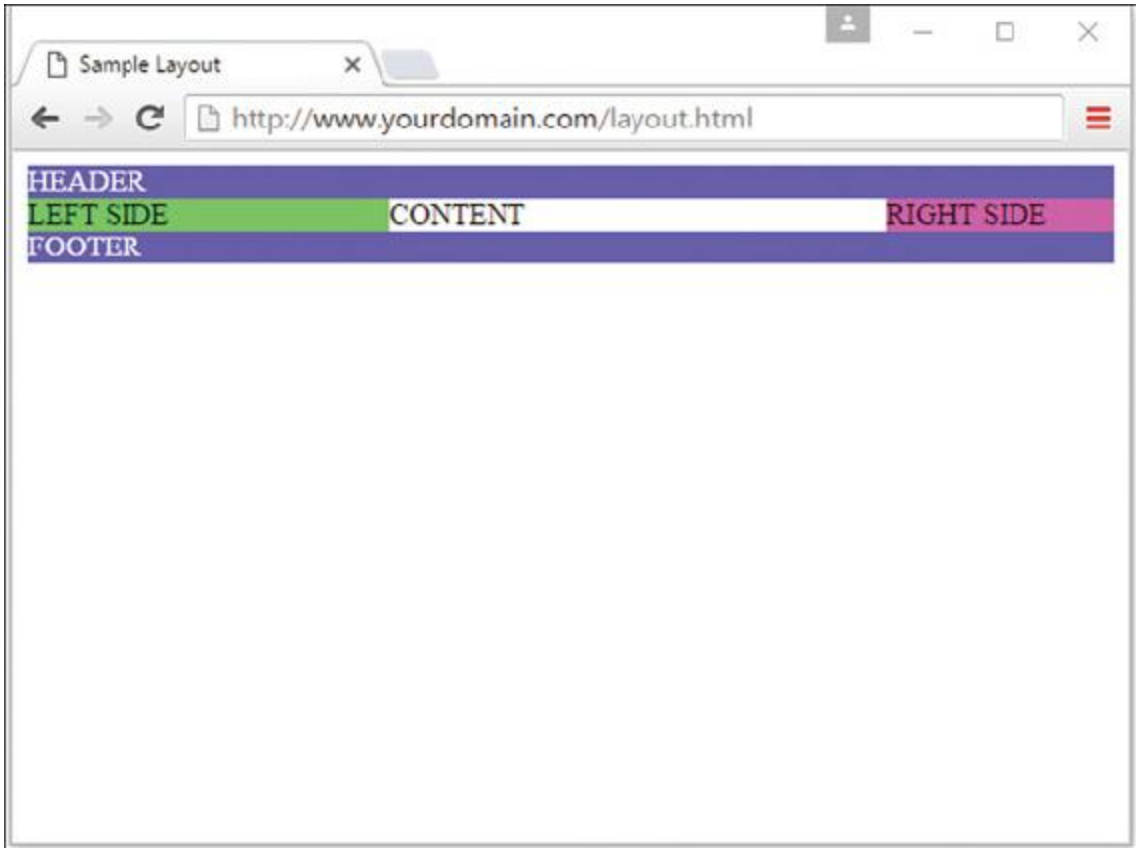

FIGURE 3.13
A basic HTML template after some styles have been put in place.

However, there's a problem with this template if the window is resized below a certain width. Because the left column is 200 pixels wide and the right column is 125 pixels wide, and we want at least *some* text in the content area, you can imagine that this page will break if the window is only 350 to 400 pixels wide. We address this issue in the next section.

Setting the Minimum Width of a Layout

Although users won't likely visit your site with a desktop browser that displays less than 400 pixels wide, the example serves its purpose within the confines of this lesson. You can extrapolate and apply this information broadly: Even in fixed/fluid hybrid sites, at some point, your layout will break down unless you do something about it.

One of those "somethings" is to use the `min-width` CSS property. The `min-width` property sets the minimum width of an element, not including padding, borders, and margins. Figure 3.14 shows what happens when `min-width` is applied to the `<body>` element.

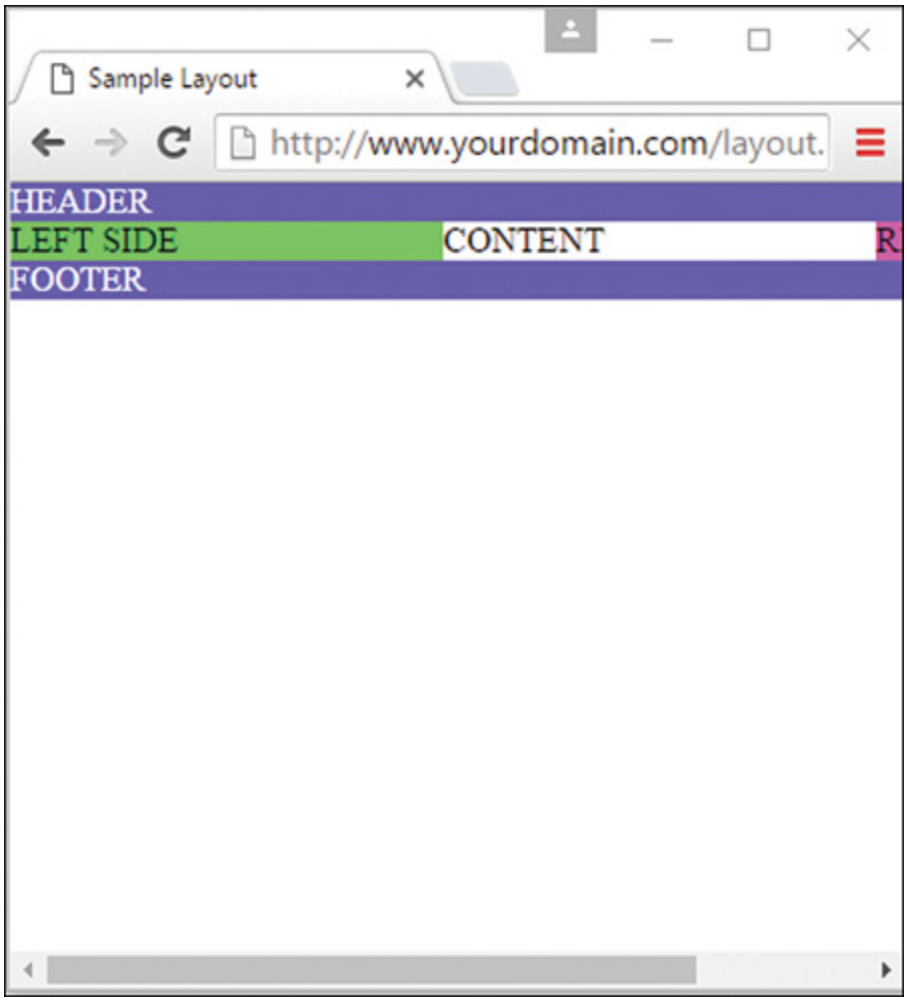

FIGURE 3.14
A basic HTML template resized to around 400 pixels, with a minimum width applied.

Figure 3.14 shows a small portion of the right column after the screen has been scrolled to the right, but the point is that the layout does not break apart when resized below a minimum width. In this case, the minimum width is 525 pixels:

```
body {
  margin: 0;
  padding: 0;
  min-width: 525px;
}
```

The horizontal scrollbar appears in this example because the browser window itself is less than 500 pixels wide. The scrollbar disappears when

the window is slightly larger than 525 pixels wide.

Handling Column Height in a Fixed/Fluid Hybrid Layout

This example is all well and good except for one problem: It has no content. When content is added to the various elements, more problems arise. As Figure 3.15 shows, the columns become as tall as necessary for the content they contain.

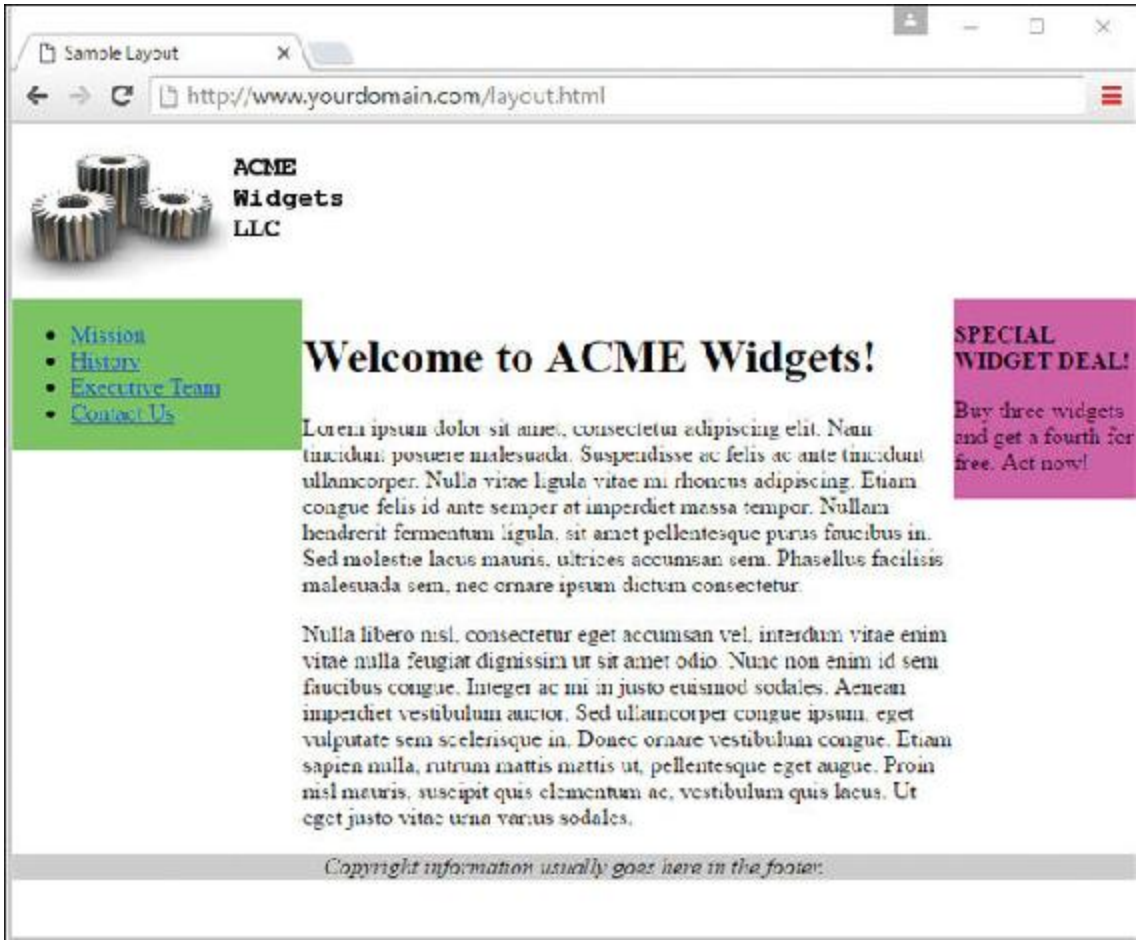

FIGURE 3.15
Columns are only as tall as their content.

NOTE

Because we have moved beyond the basic layout example, I also took the liberty to remove the background and text color properties for the header and footer, which is why the example no longer shows white text on a very dark background. Additionally, I've centered the text in the `<footer>` element, which now has a light gray background.

Because you cannot count on a user's browser being a specific height, or the content always being the same length, you might think this poses a problem with the fixed/fluid hybrid layout. Not so. If you think a little outside the box, you can apply a few more styles to bring all the pieces together.

First, add the following declarations in the style sheet entries for the `left_side`, `right_side`, and `content_area` IDs:

```
margin-bottom: -2000px;
padding-bottom: 2000px;
```

These declarations add a ridiculous amount of padding and assign a too-large margin to the bottom of all three elements. You must also add `position:relative` to the footer element definitions in the style sheet so that the footer is visible despite this padding.

At this point, the page looks as shown in Figure 3.16—still not what we want, but closer.

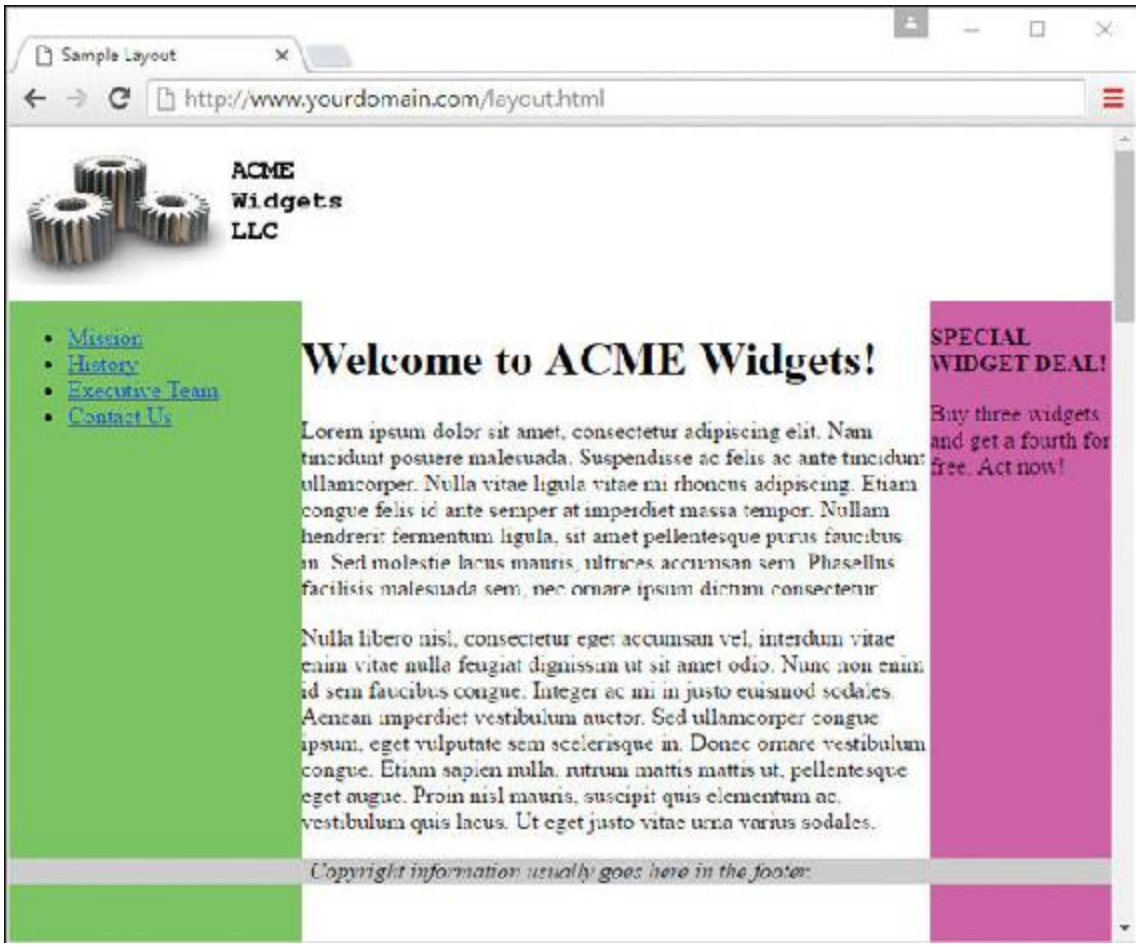

FIGURE 3.16
Color fields are now visible, despite the amount of content in the columns.

To clip off all that extra color, add the following to the style sheet for the `wrapper` ID:

```
overflow: hidden;
```

Figure 3.17 shows the final result: a fixed-width/fluid hybrid layout with the necessary column spacing. I also took the liberty of styling the navigational links and adjusting the margin around the welcome message; you can see the complete style sheet in Listing 3.6.

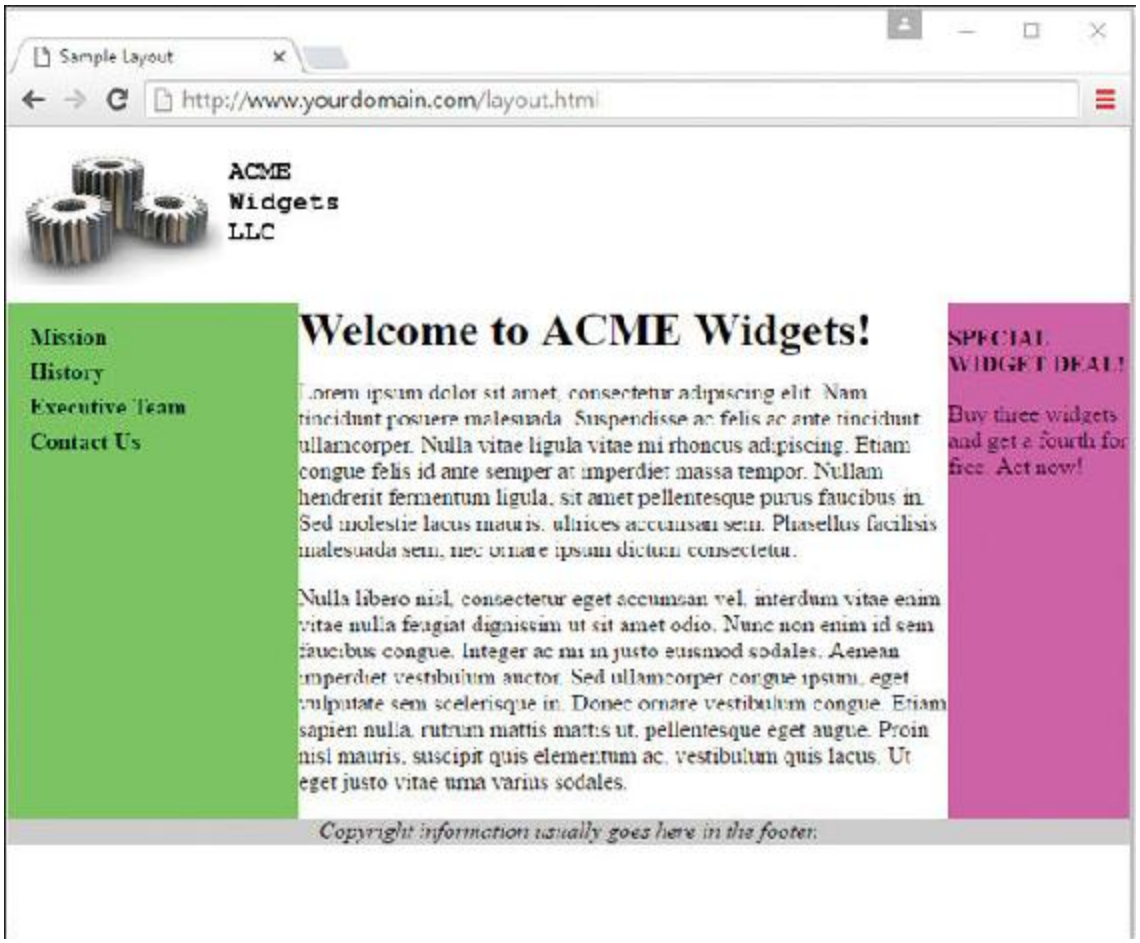

FIGURE 3.17
Congratulations! It's a fixed-width/fluid hybrid layout (although you'll want to do something about those colors!).

The full HTML code appears in Listing 3.5, and Listing 3.6 shows the final style sheet.

LISTING 3.5 Basic Fixed/Fluid Hybrid Layout Structure (with Content)

Click here to view code image

```
<!DOCTYPE html>

<html lang="en">
  <head>
      <title>Sample Layout</title>
      <link href="layout.css" rel="stylesheet" type="text/css">
  </head>
```

```
  <body>
    <header><img src="acmewidgets.jpg" alt="ACME Widgets
       LLC"/></header>
    <div id="wrapper">
       <div id="content_area">
        <h1>Welcome to ACME Widgets!</h1>
        <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
        Nam tincidunt posuere malesuada. Suspendisse ac felis ac
ante
        tincidunt ullamcorper. Nulla vitae ligula vitae mi rhoncus
        adipiscing. Etiam congue felis id ante semper at imperdiet
        massa tempor. Nullam hendrerit fermentum ligula, sit amet
        pellentesque purus faucibus in. Sed molestie lacus mauris,
        ultrices accumsan sem. Phasellus facilisis malesuada sem,
nec
        ornare ipsum dictum consectetur.</p>
        <p>Nulla libero nisl, consectetur eget accumsan vel,
interdum
        vitae enim vitae nulla feugiat dignissim ut sit amet odio.
        Nunc non enim id sem faucibus congue. Integer ac mi in
justo
        euismod sodales. Aenean imperdiet vestibulum auctor. Sed
        ullamcorper congue ipsum, eget vulputate sem scelerisque
in.
        Donec ornare vestibulum congue. Etiam sapien nulla, rutrum
        mattis mattis ut, pellentesque eget augue. Proin nisl
mauris,
        suscipit quis elementum ac, vestibulum quis lacus. Ut eget
        justo vitae urna varius sodales. </p>
       </div>
       <div id="left_side">
         <ul>
         <li><a href="#">Mission</a></li>
         <li><a href="#">History</a></li>
         <li><a href="#">Executive Team</a></li>
         <li><a href="#">Contact Us</a></li>
       </ul>
       </div>
       <div id="right_side">
         <p><strong>SPECIAL WIDGET DEAL!</strong></p>
         <p>Buy three widgets and get a fourth for free. Act now!
</p>
       </div>
    </div>
    <footer>Copyright information usually goes here in the
    footer.</footer>
  </body>
</html>
```

LISTING 3.6 Full Style Sheet for Fixed/Fluid Hybrid Layout

Click here to view code image

```
body {
  margin: 0;
  padding: 0;
  min-width: 525px;
}

header {
  float: left;
  width: 100%;
}

footer {
  position: relative;
  float: left;
  width: 100%;
  background-color: #cccccc;
  text-align: center;
  font-style: italic;
}

#wrapper {
  float: left;
  padding-left: 200px;
  padding-right: 125px;
  overflow: hidden;
}

#left_side {
  position: relative;
  float: left;
  width: 200px;
  background-color: #52f471;
  right: 200px;
  margin-left: -100%;
  margin-bottom: -2000px;
  padding-bottom: 2000px;
}

#right_side {
  position: relative;
  float: left;
  width: 125px;
  background-color: #f452d5;
  margin-right: -125px;
  margin-bottom: -2000px;
  padding-bottom: 2000px;
}
```

```
#content_area {
  position: relative;
  float: left;
  background-color: #ffffff;
  width: 100%;
  margin-bottom: -2000px;
  padding-bottom: 2000px;
}

h1 {
  margin: 0;
}

#left_side ul {
  list-style: none;
  margin: 12px 0px 0px 12px;
  padding: 0px;
}

#left_side li a:link, #nav li a:visited {
  font-size: 12pt;
  font-weight: bold;
  padding: 3px 0px 3px 3px;
  color: #000000;
  text-decoration: none;
  display: block;
}

#left_side li a:hover, #nav li a:active {
  font-size: 12pt;
  font-weight: bold;
  padding: 3px 0px 3px 3px;
  color: #ffffff;
  text-decoration: none;
  display: block;
}
```

Considering a Responsive Web Design

In 2010, web designer Ethan Marcotte coined the term *responsive web design* to refer to a web design approach that builds on the basics of fluid design you just learned a bit about. The goal of a responsive web design is that content is easy to view, read, and navigate, regardless of the device type and size on which you are viewing it. In other words, a designer who sets out to create a responsive website is doing to so to ensure that the site is

similarly enjoyable to and usable by audience members viewing on a large desktop display, a small smartphone, or a medium-size tablet.

The underlying structure of a responsive design is based on fluid (liquid) grid layouts, much as you learned about earlier in this chapter, but with a few modifications and additions. First, those grid layouts should always be in relative units rather than absolute ones. In other words, designers should use percentages rather than pixels to define container elements.

Second—and this is something we have not discussed previously—all images should be flexible. By this, I mean that instead of using a specific height and width for each image, we use relative percentages so that the images always display within the (relatively sized) element that contains them.

Finally, until you get a handle on intricate creations of style sheets for multiple uses, spend some time developing specific style sheets for each media type, and use media queries to employ these different rules based on the type. As you advance in your work and understanding of responsive design—well beyond the scope of this book—you will learn to progressively enhance your layouts in more meaningful ways.

Remember, you can specify a link to a style sheet like the following:

Click here to view code image

```
<link rel="stylesheet" type="text/css"
   media="screen and (max-device-width: 480px)"
   href="wee.css">
```

In this example, the `media` attribute contains a type and a query: The type is screen and the query portion is `(max-device-width: 480px)`. This means that if the device attempting to render the display is one with a screen and the horizontal resolution (device width) is less than 480 pixels wide—as with a smartphone—then load the style sheet called `wee.css` and render the display using the rules found within it.

Of course, a few short paragraphs in this book cannot do justice to the entirety of responsive web design. I highly recommend reading Marcotte's book *Responsive Web Design*

(http://www.abookapart.com/products/responsive-web-design) after you have firmly grounded yourself in the basics of HTML5 and CSS3 that are discussed and used throughout this book. Additionally, several of the HTML and CSS frameworks discussed in Chapter 22, "Managing Web Applications," take advantage of principles of responsive design, and that makes a great starting point for building up a responsive site and tinkering with the fluid grid, image resizing, and media queries that make it so.

Summary

This chapter began with an important discussion about the CSS box model and how to calculate the width and height of elements when taking margins, padding, and borders into consideration. The lesson continued by tackling absolute positioning of elements, and you learned about positioning using `z-index`. You then learned about a few nifty style properties that enable you to control the flow of text on a page.

Next, you saw some practical examples of the three main types of layouts: fixed, fluid, and a fixed/fluid hybrid. In the third section of the lesson, you saw an extended example that walked you through the process of creating a fixed/fluid hybrid layout in which the HTML and CSS all validate properly. Remember, the most important part of creating a layout is figuring out the sections of content you think you might need to account for in the design.

Finally, you were introduced to the concept of responsive web design, which itself is a book-length topic. Given the brief information you learned here, such as using a fluid grid layout, responsive images, and media queries, you have some basic concepts to begin testing on your own.

Q&A

Q. How would I determine when to use relative positioning and when to use absolute positioning?

A. Although there are no set guidelines regarding the usage of relative versus absolute positioning, the general idea is that absolute

positioning is required only when you want to exert a finer degree of control over how content is positioned. This has to do with the fact that absolute positioning enables you to position content down to the exact pixel, whereas relative positioning is much less predictable in terms of how it positions content. This isn't to say that relative positioning can't do a good job of positioning elements on a page; it just means that absolute positioning is more exact. Of course, this also makes absolute positioning potentially more susceptible to changes in screen size, which you can't really control.

Q. If I don't specify the z-index of two elements that overlap each other, how do I know which element will appear on top?

A. If the `z-index` property isn't set for overlapping elements, the element that appears later in the web page will appear on top. The easy way to remember this is to think of a web browser drawing each element on a page as it reads it from the HTML document; elements read later in the document are drawn on top of those that were read earlier.

Q. I've heard about something called an *elastic layout*. How does that differ from the fluid layout?

A. An *elastic layout* is a layout whose content areas resize when the user resizes the text. Elastic layouts use ems, which are inherently proportional to text and font size. An em is a typographical unit of measurement equal to the point size of the current font. When ems are used in an elastic layout, if a user forces the text size to increase or decrease in size using Ctrl and the mouse scroll wheel, the areas containing the text increase or decrease proportionally.

Q. You've spent a lot of time talking about fluid layouts and hybrid layouts—are they better than a purely fixed layout?

A. *Better* is a subjective term; in this book, the concern is with standards-compliant code. Most designers will tell you that fluid layouts take longer to create (and perfect), but the usability enhancements are worth it, especially when it leads to a responsive design. When might the

time not be worth it? If your client does not have an opinion and is paying you a flat rate instead of an hourly rate. In that case, you are working only to showcase your own skills (that might be worth it to you, however).

Workshop

The Workshop contains quiz questions and exercises to help you solidify your understanding of the material covered. Try to answer all questions before looking at the "Answers" section that follows.

Quiz

1. What's the difference between relative positioning and absolute positioning?
2. Which CSS style property controls the manner in which elements overlap each other?
3. What does `min-width` do?

Answers

1. In relative positioning, content is displayed according to the flow of a page, with each element physically appearing after the element preceding it in the HTML code. Absolute positioning, on the other hand, enables you to set the exact position of content on a page.
2. The `z-index` style property controls the manner in which elements overlap each other.
3. The `min-width` property sets the minimum width of an element, not including padding, borders, and margins.

Exercises

- Practice working with the intricacies of the CSS box model by creating a series of elements with different margins, padding, and borders, and see how these properties affect their height and width.
- Figure 3.17 shows the finished fixed/fluid hybrid layout, but notice a few areas for improvement: There isn't any space around the text in the right-side column, there aren't any margins between the body text and either column, the footer strip is a little sparse, and so on. Take some time to fix these design elements.
- Using the code you fixed in the preceding exercise, try to make it responsive, using only the brief information you learned in this chapter. Just converting container elements to relative sizes should go a long way toward making the template viewable on your smartphone or other small device, but a media query and alternative style sheet certainly wouldn't hurt, either.

CHAPTER 4

Introducing JavaScript

What You'll Learn in This Chapter:

- What web scripting is and what it's good for
- How scripting and programming are different (and similar)
- What JavaScript is and where it came from
- How to include JavaScript statements in a web page
- What JavaScript can do for your web pages
- Beginning and ending scripts
- Formatting JavaScript statements
- How a script can display a result
- Including a script within a web document
- Testing a script in your browser
- Moving scripts into separate files
- Basic syntax rules for avoiding JavaScript errors
- What JSON is and how it can be used
- Dealing with errors in scripts

The World Wide Web (WWW) began as a text-only medium—the first browsers didn't even support images within web pages. The Web has come

a long way since those early days. Today's websites include a wealth of visual and interactive features in addition to useful content: graphics, sounds, animation, and video. Using a web scripting language such as JavaScript is one of the easiest ways to spice up a web page and to interact with users in new ways. In fact, using JavaScript is the next step in taking the static HTML you've learned about in the chapters so far and turning it into something dynamic.

The first part of this chapter introduces the concept of web scripting and the JavaScript language. As the chapter moves ahead, you'll learn how to include JavaScript commands directly in your HTML documents, and how your scripts will be executed when the page is viewed in a browser. You will work with a simple script, edit it, and test it in your browser, all the while learning the basic tasks involved in creating and using JavaScript scripts.

Learning Web Scripting Basics

You already know how to use one type of computer language: HTML. You use HTML tags to describe how you want your document formatted, and the browser obeys your commands and shows the formatted document to the user. But because HTML is a simple text markup language, it can't respond to the user, make decisions, or automate repetitive tasks. Interactive tasks such as these require a more sophisticated language: a programming language or a *scripting* language.

Whereas many programming languages are complex, scripting languages are generally simple. They have a simple syntax, can perform tasks with a minimum of commands, and are easy to learn. JavaScript is a web scripting language that enables you to combine scripting with HTML to create interactive web pages.

Here are a few of the things you can do with JavaScript:

- Display messages to the user as part of a web page, in the browser's status line, or in alert boxes.

- Validate the contents of a form and make calculations (for example, an order form can automatically display a running total as you enter item quantities).
- Animate images or create images that change when you move the mouse over them.
- Create ad banners that interact with the user, rather than simply displaying a graphic.
- Detect features supported by browsers and perform advanced functions only on browsers that support those features.
- Detect installed plug-ins and notify the user if a plug-in is required
- Modify all or part of a web page without requiring the user to reload it
- Display or interact with data retrieved from a remote server

Scripts and Programs

A movie or a play follows a script—a list of actions (or lines) for the actors to perform. A web script provides the same type of instructions for the web browser. A script in JavaScript can range from a single line to a full-scale application. (In either case, JavaScript scripts usually run within a browser.)

Some programming languages must be *compiled*, or translated, into machine code before they can be executed. JavaScript, on the other hand, is an *interpreted* language: The browser executes each line of script as it comes to it.

There is one main advantage to interpreted languages: Writing or changing a script is very simple. Changing a JavaScript script is as easy as changing a typical HTML document, and the change is enacted as soon as you reload the document in the browser.

How JavaScript Fits into a Web Page

Using the `<script>` tag, you can add a short script (in this case, just one line) to a web document, as shown in Listing 4.1. The `<script>` tag tells the browser to start treating the text as a script, and the closing

</script> tag tells the browser to return to HTML mode. In most cases, you can't use JavaScript statements in an HTML document except within <script> tags. The exception is event handlers, which are described later in this chapter.

LISTING 4.1 A Simple HTML Document with a Simple Script

Click here to view code image

```
<!DOCTYPE html>

<html lang="en">
  <head>
    <title>A Spectacular Time!</title>
  </head>

  <body>
    <h1>It is a Spectacular Time!</h1>
    <p>What time is it, you ask?</p>
    <p>Well, indeed it is: <br>
    <script type="text/javascript">
    var currentTime = new Date();
    document.write(currentTime);
    </script>
    </p>
  </body>
</html>
```

JavaScript's document.write method, which you'll learn more about later, sends output as part of the web document. In this case, it displays the current date and time, as shown in Figure 4.1.

In this example, we placed the script within the body of the HTML document. There are actually four places where you might place scripts:

- **In the body of the page**—In this case, the script's output is displayed as part of the HTML document when the browser loads the page.
- **In the header of the page between the <head> tags**—Scripts in the header should not be used to create output within the <head> section of an HTML document because that would likely result in poorly formed and invalid HTML documents, but these scripts can be referred to by other scripts here and elsewhere. The <head> section is often

used for functions—groups of JavaScript statements that can be used as a single unit. You will learn more about functions in Chapter 9, "Understanding JavaScript Event Handling."

- **Within an HTML tag, such as `<body>` or `<form>`**—This is called an *event handler*, and it enables the script to work with HTML elements. When using JavaScript in event handlers, you don't need to use the `<script>` tag. You'll learn more about event handlers in Chapter 9.
- **In a separate file entirely**—JavaScript supports the use of files that can be included by specifying a file in the `<script>` tag. Although the `.js` extension is a convention, scripts can actually have any file extension, or none.

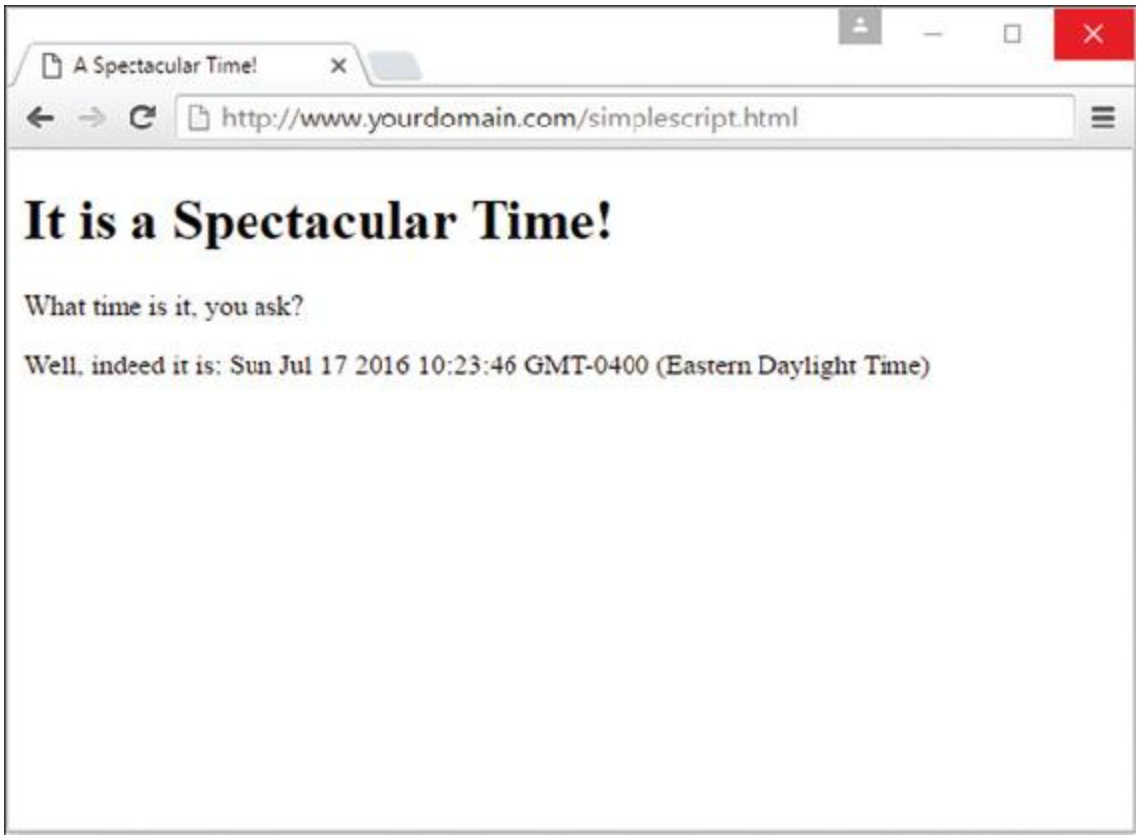

FIGURE 4.1
Using `document.write` to display the current date.

Using Separate JavaScript Files

When you create more complicated scripts, you'll quickly find that your HTML documents become large and confusing. To avoid this problem, you can use one or more external JavaScript files. These are files with the `.js` extension that contain JavaScript statements.

External scripts are supported by all modern browsers. To use an external script, you specify its filename in the `<script>` tag, like so—and don't forget to close the `<script>` tag using `</script>`:

Click here to view code image

```
<script type="text/javascript" src="filename.js"></script>
```

Because you'll be placing the JavaScript statements in a separate file, you don't need anything between the opening and closing `<script>` tags—in fact, anything between them will be ignored by the browser.

You can create the `.js` file using a text editor. It should contain one or more JavaScript statements, and only JavaScript—don't include `<script>` tags, other HTML tags, or HTML comments. Save the `.js` file in the same directory as the HTML documents that refer to it.

TIP

External JavaScript files have a distinct advantage: You can link to the same `.js` file from two or more HTML documents. Because the browser stores this file in its cache, the time it takes your web pages to display is reduced.

Using Basic JavaScript Events

Many of the useful things you can do with JavaScript involve interacting with the user, and that means responding to *events*—for example, a link or a button being clicked. You can define event handlers within HTML tags to tell the browser how to respond to an event. For example, Listing 4.2 defines a button that displays a message when clicked.

LISTING 4.2 A Simple Event Handler

Click here to view code image

```
<!DOCTYPE html>

<html lang="en">
  <head>
    <title>Event Test</title>
  </head>

  <body>
    <h1>Event Test</h1>
    <button type="button"
            onclick="alert('You clicked the button.');">
            Click Me!</button>
  </body>
</html>
```

In various places throughout this book, you'll learn more about JavaScript's event model and how to create simple and complex event handlers—in some cases, invoking PHP scripts for even greater dynamic experiences as you'll learn in Chapter 11, "AJAX: Getting Started with Remote Scripting".

Exploring JavaScript's Capabilities

If you've spent any time browsing the Web, you've undoubtedly seen lots of examples of JavaScript, even if you were not aware that JavaScript was powering your interactions. Here are some brief descriptions of typical applications for JavaScript.

Validating Forms

Form validation is a common use of JavaScript, although the form validation features of HTML5 have stolen a lot of JavaScript's thunder here. A simple script can read values the user types into a form and make sure they're in the right format, such as with ZIP Codes, phone numbers, and email addresses. This type of client-side validation enables users to fix common errors without waiting for a response from the web server telling them that their form submission was invalid.

Special Effects

One of the earliest, and admittedly most annoying, uses of JavaScript was to create attention-getting special effects—for example, scrolling a message in the browser's status line or flashing the background color of a page.

These techniques have fortunately fallen out of style, but thanks to the W3C DOM and the latest browsers, some more impressive effects are possible with JavaScript—for example, creating objects that can be dragged and dropped on a page, or creating fading transitions between images in a slideshow. Additionally, some developers use HTML5, CSS3, and JavaScript in tandem to create fully functioning interactive games.

Remote Scripting (AJAX)

For a long time, one of the greatest limitations of JavaScript was that there was no way for it to communicate with a web server. For example, you could use JavaScript to verify that a phone number had the right number of digits, but you could not use JavaScript to look up the user's location in a database based on the number.

However, now your scripts can get data from a server without loading a page or sending data back to be saved. These features are collectively known as AJAX (Asynchronous JavaScript and XML), or *remote scripting*. You'll learn how to develop AJAX scripts in Chapter 11.

You've seen AJAX in action if you've used Google's Gmail mail application, Facebook, or any online news site that allows you to comment on stories, vote for favorites, or participate in a poll (among many other things). All of these use remote scripting to present you with a dynamic user interface that interacts with a server in the background.

Basic JavaScript Language Concepts

As you build on your knowledge, there are a few foundational concepts you should know that form the basic building blocks of JavaScript.

Statements

Statements are the basic units of programs, be it a JavaScript or PHP program (or any other language). A statement is a section of code that performs a single action. For example, the following four statements create a new `Date` object and then assign the values for the current hour, minutes, and seconds into variables called `hours`, `mins`, and `secs`, respectively. You can then use these variables elsewhere in your JavaScript code.

```
now = new Date();
hours = now.getHours();
mins = now.getMinutes();
secs = now.getSeconds();
```

Although a statement is typically a single line of JavaScript, this is not a rule—it's possible (and fairly common) to break a statement across multiple lines, or to include more than one statement in a single line.

A semicolon marks the end of a statement, but you can also omit the semicolon if you start a new line after the statement—if that is your coding style. In other words, these are three valid JavaScript statements:

```
hours = now.getHours()
mins = now.getMinutes()
secs = now.getSeconds()
```

However, if you combine statements into a single line, you must use semicolons to separate them. For example, the following line is valid:

Click here to view code image

```
hours = now.getHours(); mins = now.getMinutes(); secs =
now.getSeconds();
```

But this line is invalid:

Click here to view code image

```
hours = now.getHours() mins = now.getMinutes() secs =
now.getSeconds();
```

Again, your style is always up to you but I personally recommend always using semicolons to end statements.

Combining Tasks with Functions

Functions are groups of JavaScript statements that are treated as a single unit—this is a term you'll use in PHP as well. A statement that uses a function is referred to as a *function call*. For example, you might create a function called `alertMe`, which produces an alert when called, like so:

Click here to view code image

```
function alertMe() {
   alert("I am alerting you!");
}
```

When this function is called, a JavaScript alert pops up and the text `I am alerting you!` is displayed.

Functions can take arguments—the expression inside the parentheses—to tell them what to do. Additionally, a function can return a value to a waiting variable. For example, the following function call prompts the user for a response and stores it in the `text` variable:

Click here to view code image

```
text = prompt("Enter some text.")
```

Creating your own functions is useful for two main reasons: First, you can separate logical portions of your script to make it easier to understand. Second, and more important, you can use the function several times or with different data to avoid repeating script statements.

NOTE

Entire chapters of this book are devoted to learning how to create and use functions both in JavaScript and in PHP.

Variables

Variables are containers that can store a number, a string of text, or another value. For example, the following statement creates a variable called `food` and assigns it the value `cheese`:

```
var food = "cheese";
```

JavaScript variables can contain numbers, text strings, and other values. You'll learn more about variables in much greater detail in Chapter 7, "JavaScript Fundamentals: Variables, Strings, and Arrays," and in the context of PHP in Chapter 12, "PHP Fundamentals: Variables, Strings, and Arrays."

Objects

JavaScript also supports *objects*. Like variables, objects can store data—but they can store two or more pieces of data at once. As you'll learn throughout the JavaScript-specific chapters of this book, using built-in objects and their methods is fundamental to JavaScript—it's one of the ways the language works, by providing a predetermined set of actions you can perform. For example, the `document.write` functionality you saw earlier in this chapter is actually a situation in which you use the `write` method of the `document` object to output text to the browser for eventual rendering.

The data stored in an object is called a *property* of the object. For example, you could use objects to store information about people in an address book. The properties of each person object might include a name, an address, and a telephone number.

You'll want to become intimately familiar with object-related syntax, because you will see objects quite a lot, even if you don't build your own. You'll definitely find yourself using built-in objects, and objects will very likely form a large part of any JavaScript libraries you import for use. JavaScript uses periods to separate object names and property names. For example, for a person object called Bob, the properties might include `Bob.address` and `Bob.phone`.

Objects can also include *methods*. These are functions that work with the object's data. For example, our person object for the address book might include a `display()` method to display the person's information. In JavaScript terminology, the statement `Bob.display()` would display Bob's details.

Don’t worry if this sounds confusing—you’ll be exploring objects in much more detail later in this book both in the context of learning JavaScript fundamentals and PHP fundamentals. For now, you just need to know the basics, which are that JavaScript supports three kinds of objects:

- *Built-in objects* are built in to the JavaScript language. You’ve already encountered one of these: `Date`. Other built-in objects include `Array`, `String`, `Math`, `Boolean`, `Number`, and `RegExp`.
- *DOM (Document Object Model) objects* represent various components of the browser and the current HTML document. For example, the `alert()` function you used earlier in this chapter is actually a method of the `window` object.
- *Custom objects* are objects you create yourself. For example, you could create a `Person` object, as mentioned earlier in this section.

Conditionals

Although you can use event handlers to notify your script (and potentially the user) when something happens, you might need to check certain conditions yourself as your script runs. For example, you might want to validate on your own that a user entered a valid email address in a web form.

JavaScript supports *conditional statements*, which enable you to answer questions like this. A typical conditional uses the `if` keyword, as in this example:

Click here to view code image

```
if (count == 1) {
   alert("The countdown has reached 1.");
}
```

This compares the variable count with the constant `1` and displays an alert message to the user if they are the same. It is quite likely you will use one or more conditional statements like this in most of your scripts, and more space is devoted to this concept in Chapter 8, “JavaScript Fundamentals: Functions, Objects, and Flow Control.”

Loops

Another useful feature of JavaScript—and most other programming languages—is the capability to create *loops*, or groups of statements that repeat a certain number of times. For example, these statements display the same alert 10 times, greatly annoying the user but showing how loops work:

Click here to view code image

```
for (i=1; i<=10; i++) {
   alert("Yes, it's yet another alert!");
}
```

The `for` statement is one of several statements JavaScript uses for loops. This is the sort of thing computers are supposed to be good at—performing repetitive tasks. You will use loops in many of your scripts, in much more useful ways than this example, as you'll see in Chapter 8.

Event Handlers

As mentioned previously, not all scripts are located within `<script>` tags. You can also use scripts as *event handlers*. Although this might sound like a complex programming term, it actually means exactly what it says: Event handlers are scripts that handle events. You learned a little bit about events already, but not to the extent you'll read about now or learn in Chapter 9, "Understanding JavaScript Event Handling."

In real life, an event is something that happens to you. For example, the things you write on your calendar are scheduled events, such as "Dentist appointment" and "Fred's birthday." You also encounter unscheduled events in your life, such as a traffic ticket and an unexpected visit from relatives.

Whether events are scheduled or unscheduled, you probably have normal ways of handling them. Your event handlers might include things such as *When Fred's birthday arrives, send him a present* and *When relatives visit unexpectedly, turn off the lights and pretend nobody is home*.

Event handlers in JavaScript are similar: They tell the browser what to do when a certain event occurs. The events JavaScript deals with aren't as exciting as the ones you deal with—they include such events as *When the*

mouse button is pressed and *When this page is finished loading*. Nevertheless, they're a very useful part of JavaScript's environment.

Many events (such as mouse clicks, which you've seen previously) are caused by the user. Rather than doing things in a set order, your script can respond to the user's actions. Other events don't involve the user directly—for example, an event can be triggered when an HTML document finishes loading.

Each event handler is associated with a particular browser object, and you can specify the event handler in the tag that defines the object. For example, images and text links have an event, `onmouseover`, that happens when the mouse pointer moves over the object. Here is a typical HTML image tag with an event handler:

Click here to view code image

```
<img src="button.gif" onmouseover="highlight();">
```

You specify the event handler as an attribute within the HTML tag and include the JavaScript statement to handle the event within the quotation marks. This is an ideal use for functions because function names are short and to the point and can refer to a whole series of statements.

▼ TRY IT YOURSELF

Using an Event Handler

Here's a simple example of an event handler that will give you some practice setting up an event and working with JavaScript without using `<script>` tags. Listing 4.3 shows an HTML document that includes a simple event handler.

LISTING 4.3 An HTML Document with a Simple Event Handler

Click here to view code image

```
<!DOCTYPE html>
```

```
<html lang="en">
  <head>
    <title>Simple Event Handler Example</title>
  </head>

  <body>
    <h1>Simple Event Handler Example</h1>
     <p><a href="http://www.google.com/"
           onclick="alert('A-ha! An Event!');">Go to Google</a>
     </p>
  </body>
</html>
```

The event handler is defined with the following `onclick` attribute within the `<a>` tag that defines a link:

Click here to view code image

```
onclick="alert('A-ha! An Event!');"
```

This event handler uses the DOM's built-in `alert` method of the `window` object to display a message when you click the link; after you click OK to dismiss the alert, your browser will continue on to the URL. In more complex scripts, you will usually define your own functions to act as event handlers.

You'll use other event handlers throughout this book, leading up to a more comprehensive lesson in Chapter 9.

After you click the OK button to dismiss the alert, the browser follows the link defined in the `<a>` tag. Your event handler could also stop the browser from following the link, as you will learn in Chapter 9.

Which Script Runs First?

You are not limited to a single script within a web document: one or more sets of `<script>` tags, external JavaScript files, and any number of event handlers can be used within a single document. With all of these scripts,

you might wonder how the browser knows which to execute first. Fortunately, this is done in a logical fashion:

- Sets of `<script>` tags within the `<head>` element of an HTML document are handled first, whether they include embedded code or refer to a JavaScript file. Because scripts in the `<head>` element will not create output in the web page, it's a good place to define functions for use later.
- Sets of `<script>` tags within the `<body>` section of the HTML document are executed after those in the `<head>` section, while the web page loads and displays. If there are two or more scripts in the body, they are executed in order.
- Event handlers are executed when their events happen. For example, the `onload` event handler is executed when the body of a web page loads. Because the `<head>` section is loaded before any events, you can define functions there and use them in event handlers.

JavaScript Syntax Rules

JavaScript is a simple language, but you do need to be careful to use its *syntax*—the rules that define how you use the language—correctly. The rest of this book covers many aspects of JavaScript syntax, but there are a few basic rules you can begin to keep in mind now, throughout the chapters in this book, and then when you are working on your own.

Case Sensitivity

Almost everything in JavaScript is *case sensitive*: You cannot use lowercase and capital letters interchangeably. Here are a few general rules:

- JavaScript keywords, such as `for` and `if`, are always lowercase.
- Built-in objects such as `Math` and `Date` are capitalized.
- DOM object names are usually lowercase, but their methods are often a combination of capitals and lowercase. Usually capitals are used for all but the first word, as in `setAttribute` and `getElementById`.

When in doubt, follow the exact case used in this book or another JavaScript reference. If you use the wrong case, the browser will usually display an error message.

Variable, Object, and Function Names

When you define your own variables, objects, or functions, you can choose their names. Names can include uppercase letters, lowercase letters, numbers, and the underscore (_) character. Names must begin with a letter or an underscore.

You can choose whether to use capitals or lowercase in your variable names, but remember that JavaScript is case sensitive: `score`, `Score`, and `SCORE` would be considered three different variables. Be sure to use the same name each time you refer to a variable.

Reserved Words

One more rule for variable names: They must not be *reserved words*. These include the words that make up the JavaScript language, such as `if` and `for`, DOM object names such as `window` and `document`, and built-in object names such as `Math` and `Date`.

A list of JavaScript reserved words can be found at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Reserved_Words.

Spacing

Blank space (known as *whitespace* by programmers) is mostly ignored by JavaScript. You can usually include spaces and tabs within a line, or blank lines, without causing an error, although if your statements are not clearly terminated or bracketed you may get into a sticky situation. Overall, whitespace often makes the script more readable, so do not hesitate to use it.

Using Comments

JavaScript *comments* enable you to include documentation within your script. Brief documentation is useful if someone else needs to understand the script, or even if you try to understand it after returning to your code after a long break. To include comments in a JavaScript program, begin a line with two slashes, as in this example:

```
// this is a comment.
```

You can also begin a comment with two slashes in the middle of a line, which is useful for documenting a script. In this case, everything on the line after the slashes is treated as a comment and ignored by the JavaScript. For example, the following line is a valid JavaScript statement followed by a comment explaining what is going on in the code:

Click here to view code image

```
a = a + 1; // add 1 to the value of the variable a
```

JavaScript also supports C-style comments, which begin with `/*` and end with `*/`. These comments can extend across more than one line, as the following example demonstrates:

Click here to view code image

```
/* This script includes a variety
of features, including this comment. */
```

Because JavaScript statements within a comment are ignored, this type of comment is often used for *commenting out* sections of code. If you have some lines of JavaScript that you want to temporarily take out of the picture while you debug a script, you can add `/*` at the beginning of the section and `*/` at the end.

Best Practices for JavaScript

Now that you've learned some of the very basic rules for writing valid JavaScript, it's also a good idea to follow a few *best practices*. The following practices are not required, but you'll save yourself and others

some headaches if you begin to integrate them into your development process:

- *Use comments liberally*. These make your code easier for others to understand, and also easier for you to understand when you edit it later. Comments are also useful for marking the major divisions of a script.
- *Use a semicolon at the end of each statement, and use only one statement per line*. Although you learned in this chapter that semicolons are not necessary to end a statement (if you use a new line), using semicolons and only one statement per line will make your scripts easier to read and also easier to debug.
- *Use separate JavaScript files whenever possible*. Separating large chunks of JavaScript makes debugging easier, and also encourages you to write modular scripts that can be reused.
- *Avoid being browser-specific*. As you learn more about JavaScript, you'll learn some features that work in only one browser. Avoid them unless absolutely necessary, and always test your code in more than one browser.
- *Keep JavaScript optional*. Don't use JavaScript to perform an essential function on your site—for example, the primary navigation links. Whenever possible, users without JavaScript should be able to use your site, although it might not be quite as attractive or convenient. This strategy is known as *progressive enhancement*.

There are many more best practices involving more advanced aspects of JavaScript. You'll learn about them not only as you progress through the chapters, but also over time and as you collaborate with others on web development projects.

Understanding JSON

Although JSON, or *JavaScript Object Notation*, is not a part of the core JavaScript language, it is in fact a common way to structure and store information either used by or created by JavaScript-based functionality on

the client side. Now is a good time to familiarize yourself with JSON (pronounced “Jason”) and some of its uses.

JSON-encoded data is expressed as a sequence of parameter and value pairs, with each pair using a colon to separate parameter from value. These “parameter”:“value” pairs are themselves separated by commas, as shown here:

Click here to view code image

```
"param1":"value1", "param2":"value2", "param3":"value3"
```

Finally, the whole sequence is enclosed between curly braces to form a JSON object; the following example creates a variable called `yourJSONObject`:

```
var yourJSONObject = {
   "param1":"value1",
   "param2":"value2",
   "param3":"value3"
}
```

Note that there is no comma after the final parameter—doing so would be a syntax error since no additional parameter follows.

JSON objects can have properties accessed directly using the usual dot notation, such as this:

Click here to view code image

```
alert(yourJSONObject.param1); // alerts 'value1'
```

More generally, though, JSON is a general-purpose syntax for exchanging data in a string format. It is then easy to convert the JSON object into a string by a process known as serialization; serialized data is convenient for storage or transmission around networks. You’ll see some uses of serialized JSON objects as this book progresses.

One of the most common uses of JSON these days is as a data interchange format used by application programming interfaces (APIs) and other data feeds consumed by a front-end application that uses JavaScript to parse this

data. This increased use of JSON in place of other data formats such as XML has come about because JSON is

- Easy to read for both people and computers
- Simple in concept, with a JSON object being nothing more than a series of “parameter”:“value” pairs enclosed by curly braces
- Largely self-describing
- Fast to create and parse
- No special interpreters or other additional packages are necessary

Using the JavaScript Console to Debug JavaScript

As you develop more complex JavaScript applications, you’re going to run into errors from time to time. JavaScript errors are usually caused by mistyped JavaScript statements.

To see an example of a JavaScript error message, modify the statement you added in the preceding section. We’ll use a common error: omitting one of the parentheses. Change the last `document.write` statement in Listing 4.1 to read as follows:

```
document.write(currentTime;
```

Save your HTML document again and load the document into the browser. Depending on the browser version you’re using, one of two things will happen: Either an error message will be displayed or the script will simply fail to execute.

If an error message is displayed, you’re halfway to fixing the problem by adding the missing parenthesis. If no error was displayed, you should configure your browser to display error messages so that you can diagnose future problems:

- In Firefox, you can select the Developer option, then Browser Console from the main menu.

- In Chrome, select More Tools, Developer Tools from the main menu. A console displays in the bottom of the browser window. The console is shown in Figure 4.2, displaying the error message you created in this example.
- In Microsoft Edge, select F12 Developer Tools from the main menu.

The error we get in this case is `Uncaught SyntaxError` and it points to line 15. In this case, clicking the name of the script takes you directly to the highlighted line containing the error, as shown in Figure 4.3.

Most modern browsers contain JavaScript debugging tools such as the one you just witnessed. These tools will be incredibly useful to you as you begin your journey in web application development.

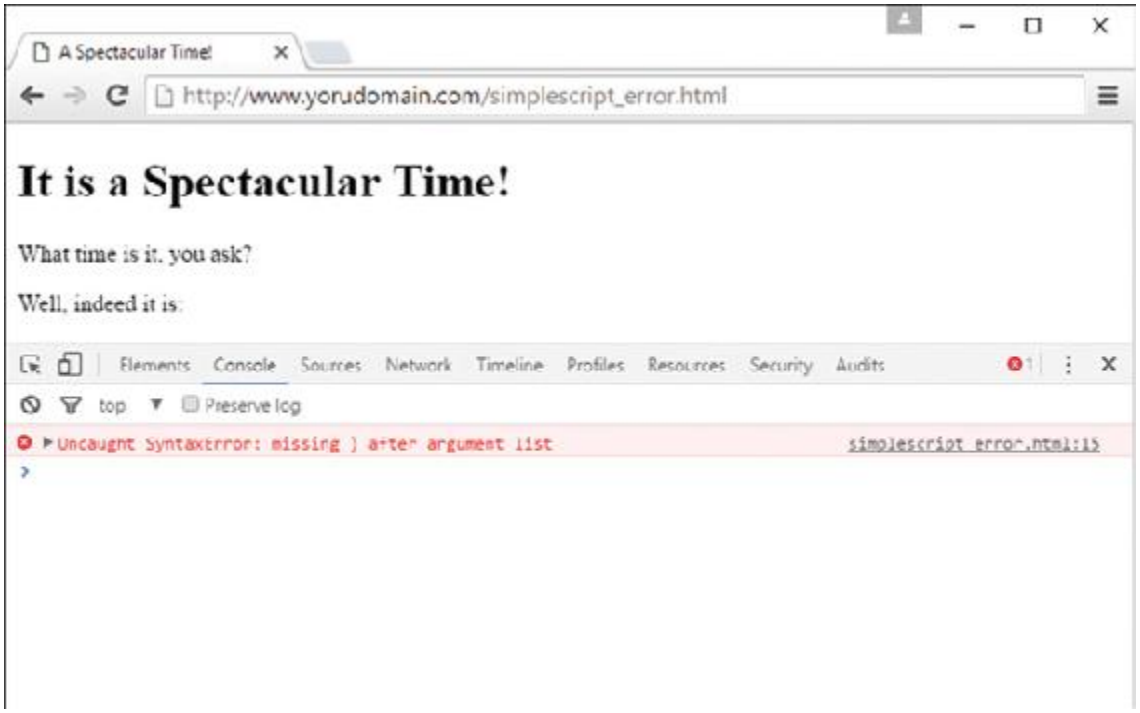

FIGURE 4.2
Showing an error in the JavaScript console in Chrome.

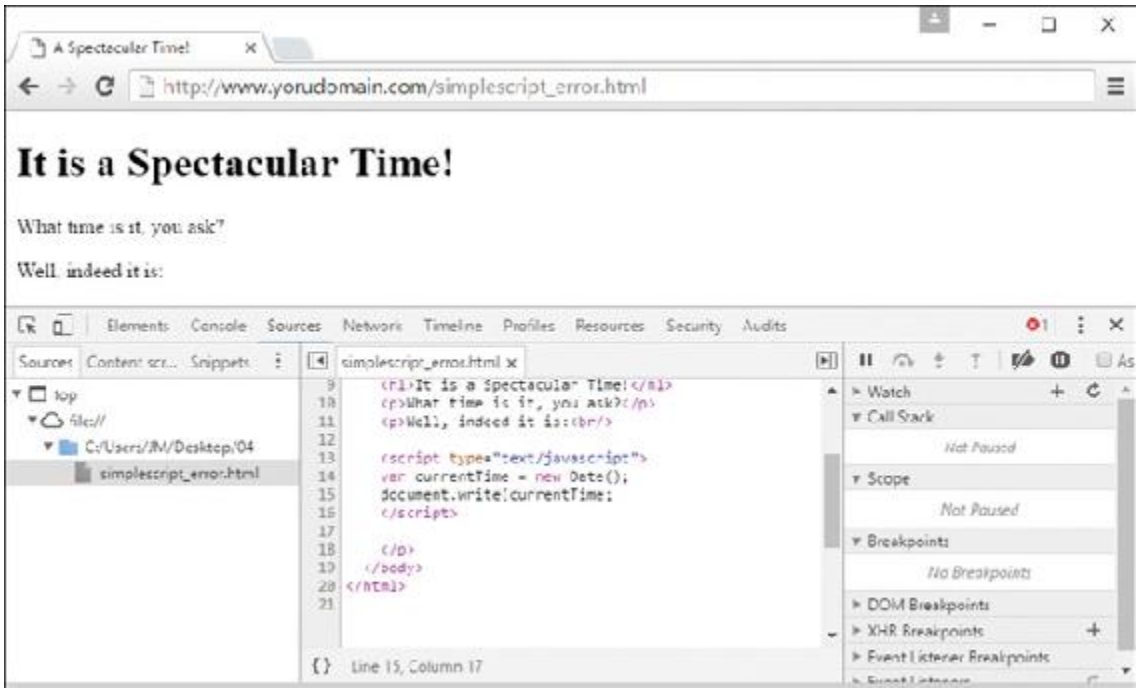

FIGURE 4.3
Chrome helpfully points out the offending line.

Summary

In this chapter, you learned what web scripting is and what JavaScript is. You also learned how to insert a script into an HTML document or refer to an external JavaScript file, what sorts of things JavaScript can do, and how JavaScript differs from other web languages. You also wrote a simple JavaScript program and tested it using a web browser.

In the process of writing this script, you have used some of JavaScript's basic features: variables, the `document.write` statement, and functions for working with dates and times. You were also introduced to the concept of JavaScript functions, objects, event handlers, conditions, and loops.

Additionally, you learned how to use JavaScript comments to make your script easier to read, and we looked at a simple example of an event handler. Finally, you were introduced to JSON, a data interchange format that is commonly used by JavaScript-based applications, and what happens when a JavaScript program runs into an error (and how to debug it).

Now that you've learned a bit of JavaScript syntax, you're ready to build on that in future chapters. But before that, take a moment in the next chapter to learn a similar set of basic information about a server-side scripting language: PHP.

Q&A

Q. Do I need to test my JavaScript on more than one browser?

A. In an ideal world, any script you write that follows the standards for JavaScript will work in all browsers, and 98% of the time (give or take) that's true in the real world. But browsers do have their quirks, and you should test your scripts in Chrome, Internet Explorer, and Firefox at a minimum.

Q. When I try to run my script, the browser displays the actual script in the browser window instead of executing it. What did I do wrong?

A. This is most likely caused by one of three errors. First, you might be missing the beginning or ending `<script>` tags. Check them, and then also verify that the first reads `<script type="text/javascript">`. Finally, your file might have been saved with a `.txt` extension, causing the browser to treat it as a text file. Rename it to `.htm` or `.html` to fix the problem.

Q. I've heard the term *object-oriented* applied to languages such as C++ and Java. If JavaScript supports objects, is it an object-oriented language?

A. Yes, although it might not fit some people's strict definitions. JavaScript objects do not support all the features that languages such as C++ and Java support, although the latest versions of JavaScript have added more object-oriented features.

Workshop

The Workshop contains quiz questions and exercises to help you solidify your understanding of the material covered. Try to answer all questions before looking at the "Answers" section that follows.

Quiz

1. When a user views a page containing a JavaScript program, which machine actually executes the script?

   A. The user's machine running a web browser

   B. The web server

   C. A central machine deep within Netscape's corporate offices

2. What software do you use to create and edit JavaScript programs?

   A. A browser

   B. A text editor

   C. A pencil and a piece of paper

3. What are variables used for in JavaScript programs?

   A. Storing numbers, dates, or other values

   B. Varying randomly

   C. Causing high-school algebra flashbacks

4. What should appear at the very end of a JavaScript script embedded in an HTML file?

   A. The `<script type="text/javascript">` tag

   B. The `</script>` tag

   C. The `END` statement

5. Which of the following is executed first by a browser?

   A. A script in the `<head>` section

   B. A script in the `<body>` section

   C. An event handler for a button

Answers

1. **A.** JavaScript programs execute on the web browser. (There is actually a server-side version of JavaScript, but that's another story.)
2. **B.** Any text editor can be used to create scripts. You can also use a word processor if you're careful to save the document as a text file with the `.html` or `.htm` extension.
3. **A.** Variables are used to store numbers, dates, or other values.
4. **B.** Your script should end with the `</script>` tag.
5. **A.** Scripts defined in the `<head>` section of an HTML document are executed first by the browser.

Exercises

- Examine the simple time display script you created and comment the code to explain everything that is happening all along the way. Verify that the script still runs properly.
- Use the techniques you learned in this chapter to create a script that displays the current date and time in an alert box when the user clicks a link.

CHAPTER 5

Introducing PHP

What You'll Learn in This Chapter:

- How PHP works with the web server as a "server-side" language
- How to include PHP code in a web page
- Beginning and ending PHP scripts
- How to use HTML, JavaScript, and PHP in the same file

In the first four chapters of this book, you worked with client-side or front-end languages: HTML and JavaScript. In this chapter, you'll learn a bit about PHP, which is a server-side scripting language that operates on a web server. You'll learn how to include PHP code into your HTML documents and how these scripts are executed when you refer to a page in your web browser that accesses a remote web server. This is a short chapter that is meant to be a bridge between the front end functionality you just learned about, and the backend functionality that you'll pick up again in Part III of this book.

How PHP Works with a Web Server

Often, when a user submits a request to a web server for a web page, the server reads a simple HTML file (that may or may not include JavaScript) and sends its contents back to the browser in response. If the request is for a

PHP file, or for an HTML document that includes PHP code, and the server supports PHP, then the server looks for PHP code in the document, executes it, and includes the output of that code in the page in place of the PHP code. Here's a simple example:

Click here to view code image

```
<!DOCTYPE html>
<html>
   <head>
     <title>There's PHP in Here</title>
   </head>
   <body>
      <?php echo "Howdy!"; ?>
   </body>
</html>
```

If this page is requested from a web server that supports PHP, the HTML sent to the browser will look like this:

Click here to view code image

```
<!DOCTYPE html>
<html>
   <head>
     <title>There's PHP in Here</title>
   </head>
   <body>
      Howdy!
   </body>
</html>
```

When the user requests the page, the web server determines that it is a PHP page rather than a regular HTML page. If a web server supports PHP, it usually treats any files with the extension `.php` as PHP pages. Assuming this page is called something like `howdy.php`, when the web server receives the request, it scans the page looking for PHP code and then runs any code it finds. PHP code is distinguished from the rest of a page by PHP tags, which you'll learn more about in the next section.

Whenever the server finds those tags, it treats whatever is within them as PHP code. That's not so different from the way things work with JavaScript, where anything inside `<script>` tags is treated as JavaScript code.

The Basics of PHP Scripts

Let's jump straight into the fray with the simplest PHP script possible that also produces some meaningful output. To begin, open your favorite text editor. Like HTML documents, PHP files are made up of plain text. You can create them with any text editor, and most popular HTML editors and programming IDEs (integrated development environments) provide support for PHP.

TIP

If you have an IDE or simple text editor that you enjoy using for HTML and JavaScript, it's likely to work just fine with PHP as well.

Type in the example shown in Listing 5.1 and save the file to the document root of your web server, using a name something like `test.php`.

LISTING 5.1 A Simple PHP Script

```
<?php
    phpinfo();
?>
```

This script simply tells PHP to use the built-in function called `phpinfo()`. This function automatically generates a significant amount of detail about the configuration of PHP on your system. You can also begin to see the power of a scripting language, in which a little bit of text can go a long way toward producing something useful.

If you are not working directly on the machine that will be serving your PHP script, you need to use a File Transfer Protocol (FTP) or Secure Copy Protocol (SCP) client to upload your saved document to the server. When the document is in place on the server, you should be able to access it using your browser. If all has gone well, you should see the script's output. Figure 5.1 shows the output from the `test.php` script.

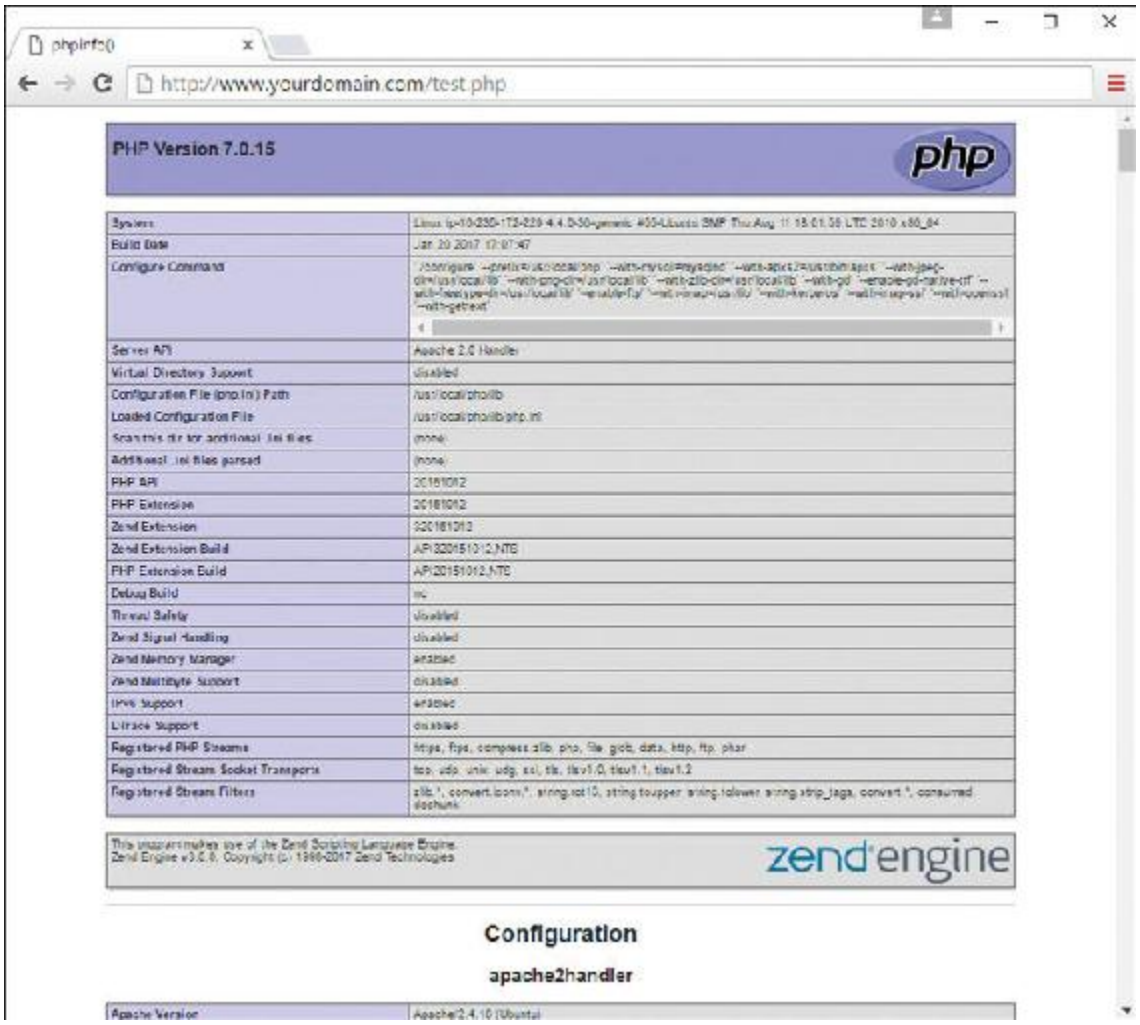

FIGURE 5.1
Success: the output from `test.php`.

Beginning and Ending a Block of PHP Statements

When writing PHP, you need to inform the PHP engine that you want it to execute your commands. If you don't do this, the code you write will be mistaken for HTML and will be output to the browser. You can designate your code as PHP with special tags that mark the beginning and end of PHP code blocks. Table 5.1 shows the two possible PHP delimiter tags.

TABLE 5.1 PHP Start and End Tags

Tag Style	Start Tag	End Tag

Standard tags	`<?php`	`?>`
Short tags	`<?`	`?>`

Standard tags are highly recommended and are the default expectation by the PHP engine. If you want to use short tags, you must explicitly enable short tags in your PHP configuration by making sure that the `short_open_tag` switch is set to `On` in `php.ini`:

```
short_open_tag = On;
```

Such a configuration change will require a restart of your web server. This is largely a matter of preference, although if you intend to include XML in your script, you should not enable the short tags (`<?` and `?>`) and should work with the standard tags (`<?php` and `?>`). Standard tags will be used throughout this book, but you should be aware you may run into this other type of tag if you are reading other people's code; if you use that code, you must adjust the PHP tags before you use it.

CAUTION

The character sequence `<?` tells an XML parser to expect a processing instruction and is therefore often included in XML documents. If you include XML in your script and have short tags enabled, the PHP engine is likely to confuse XML processing instructions and PHP start tags. Definitely do not enable short tags if you intend to incorporate XML in your document. Again, the standard tags will really serve you best in the long run.

Let's run through the two ways in which you can legally write the code in Listing 5.1, including using short tags if that configuration option is enabled:

```
<?php
     phpinfo();
?>
```

```
<?
     phpinfo();
?>
```

You can also put single lines of code in PHP on the same line as the PHP start and end tags:

```
<?php phpinfo(); ?>
```

Now that you know how to define a block of PHP code with PHP tag delimiters, let's move forward.

The `echo` and `print()` Statements

The following bit of PHP code does exactly one thing—it displays "Hello!" on the screen:

```
<?php
     echo "Hello!";
?>
```

The `echo` statement, which is what we used here, outputs data. In most cases, anything output by `echo` ends up viewable in the browser. Alternatively, you could have used the `print()` statement in place of the `echo` statement. Using `echo` or `print()` is a matter of taste; when you look at other people's scripts, you might see either used, which is why I mention both here.

Referring back to the code you have seen so far, note the only line of code in Listing 5.1 ended with a semicolon. The semicolon informs the PHP engine that you have completed a statement, and it's probably the most important bit of coding syntax you could learn at this stage.

A *statement* represents an instruction to the PHP engine. Broadly, it is to PHP what a sentence is to written or spoken English. A sentence should usually end with a period; a statement should usually end with a semicolon. Exceptions to this rule include statements that enclose other statements and statements that end a block of code. In most cases, however, failure to end a statement with a semicolon will confuse the PHP engine and result in an error.

Combining HTML and PHP

The script in Listing 5.1 is pure PHP, but that does not mean you can only use PHP in PHP scripts. You can use PHP and HTML (and JavaScript too!) in the same document by simply ensuring that all of the PHP is enclosed by PHP start and end tags, as shown in Listing 5.2.

LISTING 5.2 Some PHP Embedded Inside HTML

Click here to view code image

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Some PHP Embedded Inside HTML</title>
  </head>
  <body>
     <h1><?php echo "Hello World!"; ?></h1>
  </body>
</html>
```

As you can see, incorporating PHP code into a predominantly HTML document is simply a matter of typing in the code because the PHP engine ignores everything outside the PHP start and end tags.

Save the contents of Listing 5.2 as `helloworld.php`, place it in your document root, and then view it with a browser. You should see the string `Hello World!` in a large, bold heading, as shown in Figure 5.2. If you were to view the document source, as shown in Figure 5.3, the listing would look exactly like a normal HTML document, because all of the processing by the PHP engine will have already taken place before the output gets to the browser for rendering.

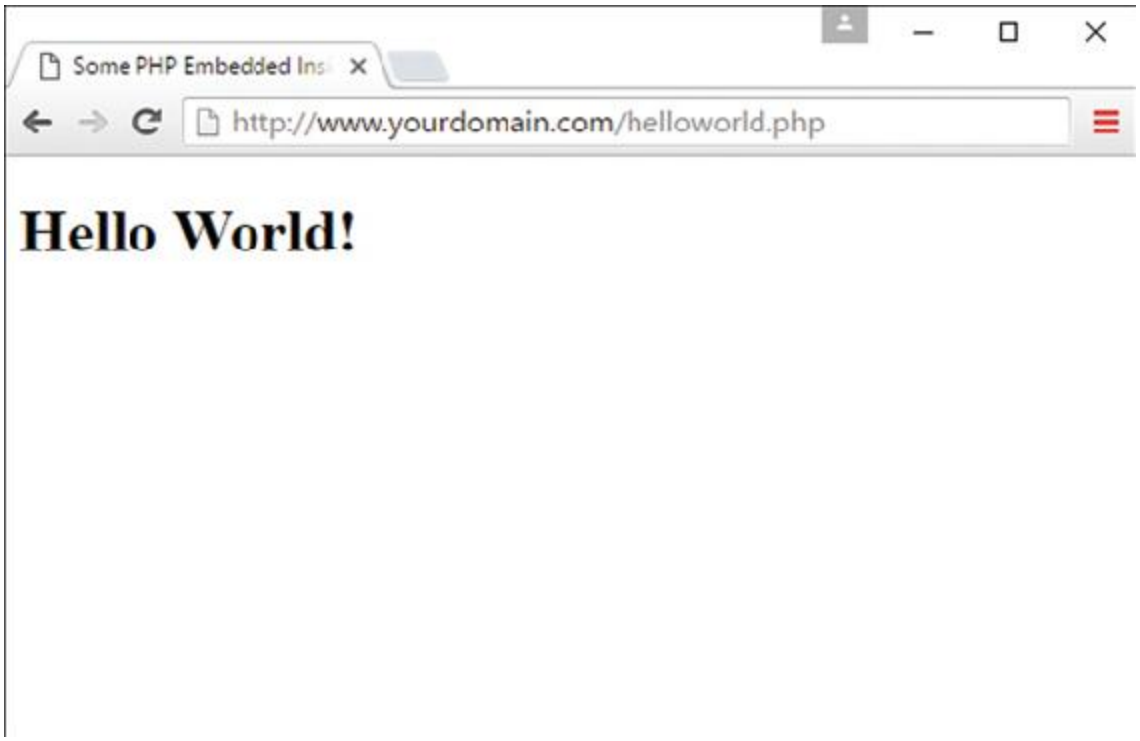

FIGURE 5.2
The output of `helloworld.php` as viewed in a browser.

You can include as many blocks of PHP code as you need in a single document, interspersing them with HTML however it makes sense. Although you can have multiple blocks of code in a single document, they combine to form a single continuous script as far as the PHP engine is concerned.

FIGURE 5.3
The output of `helloworld.php` as HTML source code.

Adding Comments to PHP Code

Code that seems clear at the time you write it can seem like a hopeless tangle when you try to amend it 6 months later. Adding comments to your code as you write can save you time later on and make it easier for other programmers to work with your code.

A *comment* is text in a script that is ignored by the PHP engine. Comments can make code more readable or can be use to annotate a script.

Single-line comments begin with two forward slashes (/ /), which is the preferred style, or a single hash or pound sign (#), which you may see in other people's code if those folks are more used to working in Perl or other languages where that comment type is more the norm. Regardless of which valid comment type you use, the PHP engine ignores all text between these marks and either the end of the line or the PHP close tag:

```
// this is a comment
#  this is another comment
```

Multiline comments begin with a forward slash followed by an asterisk (`/*`) and end with an asterisk followed by a forward slash (`*/`):

```
/*
this is a comment
none of this will
be parsed by the
PHP engine
*/
```

Code comments, just like HTML and JavaScript (and anything else that isn't PHP), are ignored by the PHP engine and will not cause errors that halt the execution of your scripts.

Code Blocks and Browser Output

In the previous section, you learned that you can slip in and out of HTML mode at will using the PHP start and end tags. In later chapters, you'll learn more about how you can present distinct output to the user according to a decision-making process you can control by using what are called *flow control statements*, which exist in both JavaScript and PHP. Although flow control in both languages will be discussed at length as the book moves forward, I use a basic example here to continue the lesson about mingling PHP and HTML with ease.

Imagine a script that outputs a table of values only when some condition is true, and doesn't output anything when a condition is false. Listing 5.3 shows a simplified HTML table constructed with the code block of an `if` statement.

LISTING 5.3 PHP Displays Text if a Condition Is True

Click here to view code image

```
<!DOCTYPE html>
<html lang="en">
<head>
   <title>More PHP Embedded Inside HTML</title>
```

```
    <style type="text/css">
    table, tr, th, td {
         border: 1px solid #000;
         border-collapse: collapse;
         padding: 3px;
    }
    th {
         font-weight: bold;
    }
    </style>
    </head>
    <body>
    <?php
    $some_condition = true;
    if ($some_condition) {
       echo "<table>
       <tr><th colspan=\"3\">
       Today's Prices
       </th></tr>
       <tr><td>14.00</td><td>32.00</td><td>71.00</td></tr>
       </table>";
     }
     ?>
     </body>
</html>
```

If the value of `$some_condition` is `true`, the table is printed. For the sake of readability, we split the output into multiple lines surrounded by one `echo` statement, and we use the backslash to escape any quotation marks used in the HTML output.

Put these lines into a text file called `embedcondition.php` and place this file in your web server document root. When you access this script through your web browser, it should look like Figure 5.4.

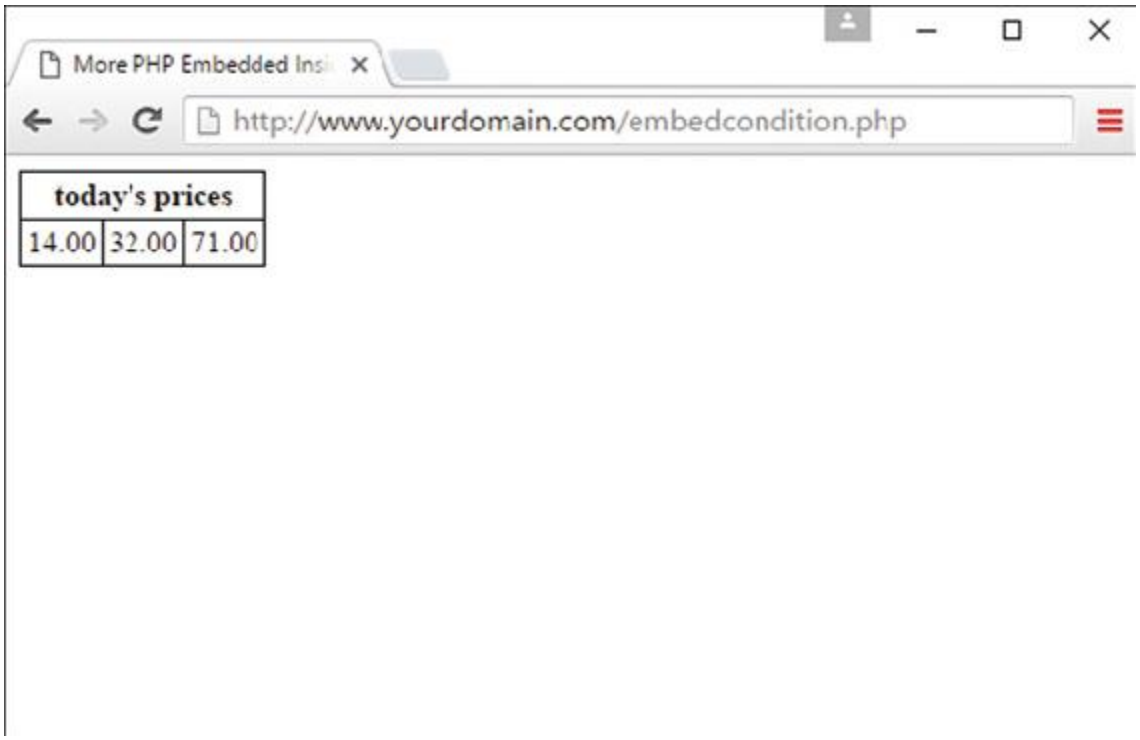

FIGURE 5.4
Output of `embedcondition.php`.

There's nothing wrong with the way this is coded, but you may find your code more readable to slip back into HTML mode within the code block itself. Listing 5.4 does just that.

LISTING 5.4 Returning to HTML Mode Within a Code Block

Click here to view code image

```
<!DOCTYPE html>
<html lang="en">
<head>
   <title>More PHP Embedded Inside HTML</title>
   <style type="text/css">
   table, tr, th, td {
        border: 1px solid #000;
        border-collapse: collapse;
        padding: 3px;
   }
   th {
        font-weight: bold;
   }
   </style>
   </head>
```

```
   <body>
       <?php
       $some_condition = true;
       if ($some_condition) {
       ?>
       <table>
       <tr><th colspan="3">Today's Prices</th></tr>
       <tr><td>14.00</td><td>32.00</td><td>71.00</td></tr>
       </table>
       <?php
       }
       ?>
    </body>
</html>
```

The important thing to note here is that the shift to HTML mode occurs only if the condition of the `if` statement is fulfilled. Slipping in and out of PHP or HTML mode like this can save you from the time and effort of escaping quotation marks and wrapping output in `echo` statements. This approach might, however, affect the readability of the code in the long run, especially if the script grows larger.

Summary

In this chapter, you learned the concept of using PHP as a server-side scripting language, and using the `phpinfo()` function, you produced a list of the PHP configuration values for the server you are using. You created a simple PHP script using a text editor, and you learned about the tags you can use to begin and end blocks of PHP code.

You learned how to use the `echo` and `print` statements to send data to the browser, and you brought HTML and PHP together into the same script. You also looked at a technique for using PHP start and end tags in conjunction with conditional code blocks, as another way to embed PHP within HTML.

The first five chapters in this book lay the groundwork for creating web applications by introducing you to the idea of interactive websites and basic concepts used when developing web content with HTML, JavaScript, and PHP. The PHP language and its use on the back end of applications (the

server side), as opposed to the front end (your web browser), makes it the more complex of the three. However, there are many similarities in the fundamentals and syntax of both PHP and JavaScript, such that directly after this chapter we will refocus on JavaScript for a bit before returning to PHP to build interactive and dynamic applications.

Q&A

Q. Which are the best start and end tags to use?

A. It is largely a matter of preference. For the sake of portability, the standard tags (`<?php` and `?>`) are preferred.

Q. What editors should I avoid when creating PHP code?

A. Do not use word processors that format text for printing (Microsoft Word, for example). Even if you save files created using this type of editor in plain-text format, hidden characters are likely to creep into your code.

Q. When should I comment my code?

A. Again, this is a matter of preference. Some short scripts will be self-explanatory, even when you return to them after a long interval. For scripts of any length or complexity, you should comment your code. Comments in your code often save you time and frustration in the long run.

Workshop

The Workshop is designed to help you review what you've learned and begin putting your knowledge into practice.

Quiz

1. Can a person browsing your website read the source code of a PHP script you have successfully installed?
2. Is the following a valid PHP script that will run without errors? If so, what will display in the browser?

```
<?php echo "Hello World!" ?>
```

3. Is the following a valid PHP script that will run without errors? If so, what will display in the browser?

```
<?php
// I learned some PHP!
?>
```

Answers

1. No, the user will see only the output of your script.
2. No, this code will produce an error because the `echo` statement is not terminated by a semicolon.
3. Yes, this code will run without errors. However, it will produce nothing in the browser because the only code within the PHP start and end tags is a PHP comment, which will be ignored by the PHP engine. Therefore, no output will be rendered.

Exercises

1. Familiarize yourself with the process of creating, uploading, and running PHP scripts. In particular, create your own "Hello World" script. Add HTML code to it as well as additional blocks of PHP.