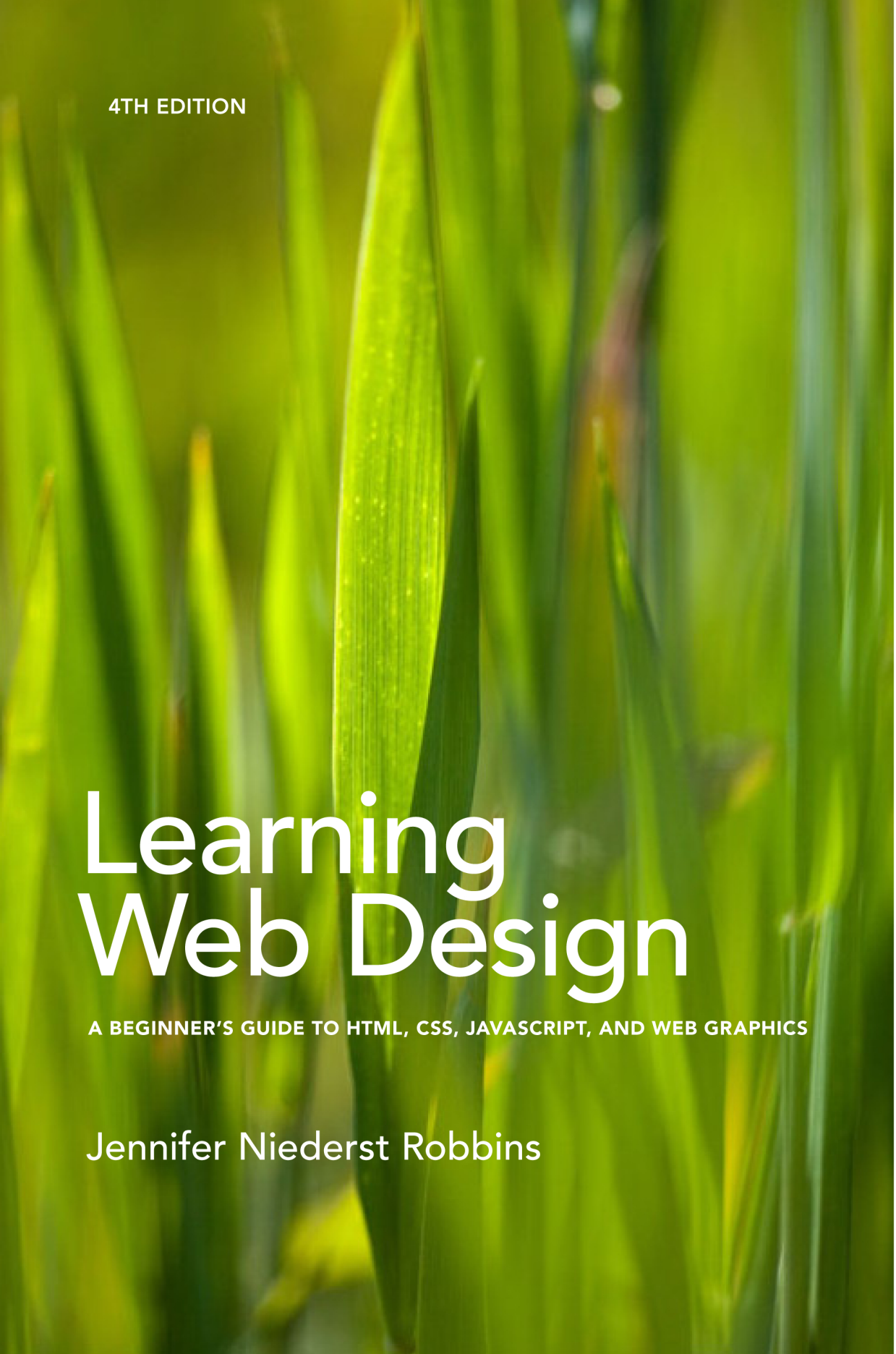


4TH EDITION

Learning Web Design

A BEGINNER'S GUIDE TO HTML, CSS, JAVASCRIPT, AND WEB GRAPHICS

Jennifer Niederst Robbins

Whether you're a beginner or bringing your skills up to date, this book gives you a solid footing in modern web production. I teach each topic visually at a pleasant pace, with frequent exercises to let you try out new skills. Reading it feels like sitting in my classroom! —Jennifer Robbins

O'REILLY®

HOW THE WEB WORKS

I got started in web design in early 1993—pretty close to the start of the Web itself. In web time, that makes me an old-timer, but it's not so long ago that I can't remember the first time I looked at a web page. It was difficult to tell where the information was coming from and how it all worked.

This chapter sorts out the pieces and introduces some basic terminology. We'll start with the big picture and work down to specifics.

IN THIS CHAPTER

An explanation of the Web, as it relates to the Internet

The role of the server

The role of the browser

Introduction to URLs and their components

The anatomy of a web page

The Internet Versus the Web

No, it's not a battle to the death, just an opportunity to point out the distinction between these two words that are increasingly being used interchangeably.

The Internet is a network of connected computers. No company owns the Internet; it is a cooperative effort governed by a system of standards and rules. The purpose of connecting computers together, of course, is to share information. There are many ways information can be passed between computers, including email, file transfer (FTP), and many more specialized modes upon which the Internet is built. These standardized methods for transferring data or documents over a network are known as protocols.

The Web (originally called the World Wide Web, thus the "www" in site addresses) is just one of the ways information can be shared over the Internet. It is unique in that it allows documents to be linked to one another using hypertext links—thus forming a huge "web" of connected information. The Web uses a protocol called HTTP (HyperText Transfer Protocol). That acronym should look familiar because it is the first four letters of nearly all website addresses, as we'll discuss in an upcoming section.

The Web is a subset of the Internet. It is just one of many ways information can be transferred over networked computers.

Serving Up Your Information

Let's talk more about the computers that make up the Internet. Because they "serve up" documents upon request, these computers are known as servers. More accurately, the server is the software (not the computer itself) that

A Brief History of the Web

The Web was born in a particle physics laboratory (CERN) in Geneva, Switzerland in 1989. There a computer specialist named Tim Berners-Lee first proposed a system of information management that used a "hypertext" process to link related documents over a network. He and his partner, Robert Cailliau, created a prototype and released it for review. For the first several years, web pages were text-only. It's difficult to believe that in 1992, the world had only about 50 web servers, total.

The real boost to the Web's popularity came in 1992 when the first graphical browser (NCSA Mosaic) was introduced, and the Web broke out of the realm of scientific research into mass media. The ongoing development of web technologies is overseen by the World Wide Web Consortium (W3C).

If you want to dig deeper into the Web's history, check out this site:

W3C's History Archives

www.w3.org/History.html

allows the computer to communicate with other computers; however, it is common to use the word "server" to refer to the computer as well. The role of server software is to wait for a request for information, then retrieve and send that information back as quickly as possible.

There's nothing special about the computers themselves...picture anything from a high-powered Unix machine to a humble personal computer. It's the server software that makes it all happen. In order for a computer to be part of the Web, it must be running special web server software that allows it to handle Hypertext Transfer Protocol transactions. Web servers are also called "HTTP servers."

There are many server software options out there, but the two most popular are Apache (open source software) and Microsoft Internet Information Services (IIS). Apache is freely available for Unix-based computers and comes installed on Macs running Mac OS X. There is a Windows version as well. Microsoft IIS is part of Microsoft's family of server solutions.

Every computer and device (modem, router, smartphone, cars, etc.) connected to the Internet is assigned a unique numeric IP address (IP stands for Internet Protocol). For example, the computer that hosts *oreilly.com* has the IP address 208.201.239.100. All those numbers can be dizzying, so fortunately, the Domain Name System (DNS) was developed to allow us to refer to that server by its domain name, "oreilly.com", as well. The numeric IP address is useful for computer software, while the domain name is more accessible to humans. Matching the text domain names to their respective numeric IP addresses is the job of a separate DNS server.

It is possible to configure your web server so that more than one domain name is mapped to a single IP address, allowing several sites to share a single server.

TERMINOLOGY

Open Source

Open source software is developed as a collaborative effort with the intent to make its source code available to other programmers for use and modification. Open source programs are usually available for free.

No More IP Addresses

The IANA, the organization that assigns IP numbers, handed out its last bundle of IP addresses on February 3, 2011. That's right, no more ###.###.###.###-style IPs. That format of IP address (called IPv4) is able to produce 4.3 billion unique addresses, which seemed like plenty when the Internet "experiment" was first conceived in 1977. There was no way the creators could anticipate that one day every phone, television, and object on store shelves would be clamoring for one.

The solution is a new IP format (IPv6, already in the works) that allows for trillions and trillions of unique IP numbers, with the slight snag that it is incompatible with our current IPv4-based network, so IPv6 will operate as a sort of parallel Internet to the one we have today. Eventually, IPv4 will be phased out, but some say it will take decades.

A Word About Browsers

We now know that the server does the servin', but what about the other half of the equation? The software that does the requesting is called the client. People use desktop browsers, mobile browsers, and other assistive technologies (such as screen readers) as clients to access documents on the Web. The server returns the documents for the browser (also referred to as the user agent in technical circles) to display.

The requests and responses are handled via the HTTP protocol, mentioned earlier. Although we've been talking about "documents," HTTP can be used to transfer images, movies, audio files, data, scripts, and all the other web resources that commonly make up web sites and applications.

It is common to think of a browser as a window on a computer monitor with a web page displayed in it. These are known as graphical browsers or desktop browsers and for a long time, they were the only web-viewing game in town. The most popular desktop browsers as of this writing include Internet Explorer for Windows, Chrome, Firefox, and Safari, with Opera bringing up the rear. These days, however, more and more people are accessing the Web on the go using browsing clients built into mobile phones or tablets.

It is also important to keep alternative web experiences in mind. Users with sight disabilities may be listening to a web page read by a screen reader (or simply make their text extremely large). Users with limited mobility may use assistive devices to access links and to type. The sites we build must be accessible and usable for all users, regardless of their browsing experiences.

Even on the desktop browsers that first introduced us to the wide world of the Web, pages may look and perform differently from browser to browser. This is due to varying support for web technologies and the users' ability to set their own browsing preferences.

TERMINOLOGY

Server-side and Client-side

Often in web design, you'll hear reference to "client-side" or "server-side" applications. These terms are used to indicate which machine is doing the processing. Client-side applications run on the user's machine, while server-side applications and functions use the processing power of the server computer.

Intranets and Extranets

When you think of a website, you generally assume that it is accessible to anyone surfing the Web. However, many companies take advantage of the awesome information sharing and gathering power of websites to exchange information just within their own business. These special web-based networks are called intranets. They are created and function like ordinary websites, but they use special security devices (called firewalls) that prevent the outside world from seeing them. Intranets have lots of uses, such as sharing human resource information or providing access to inventory databases.

An extranet is like an intranet, only it allows access to select users outside of the company. For instance, a manufacturing company may provide its customers with passwords that allow them to check the status of their orders in the company's orders database. Of course, the passwords determine which slice of the company's information is accessible.

Web Page Addresses (URLs)

Every page and resource on the Web has its own special address called a URL, which stands for Uniform Resource Locator. It's nearly impossible to get through a day without seeing a URL (pronounced "U-R-L," not "erl") plastered on the side of a bus, printed on a business card, or broadcast on a television commercial. Web addresses are fully integrated into modern vernacular.

Some URLs are short and sweet. Others may look like crazy strings of characters separated by dots (periods) and slashes, but each part has a specific purpose. Let's pick one apart.

The parts of a URL

A complete URL is generally made up of three components: the protocol, the site name, and the absolute path to the document or resource, as shown in Figure 2-1.

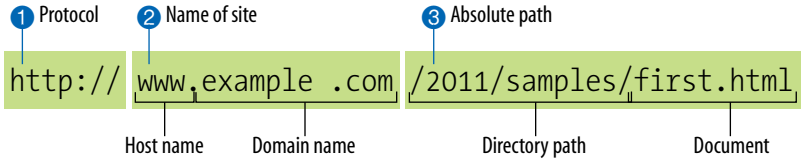


Figure 2-1. The parts of a URL.

❶ `http://`

The first thing the URL does is define the protocol that will be used for that particular transaction. The letters HTTP let the server know to use Hypertext Transfer Protocol, or get into "web mode."

❷ `www.example.com`

The next portion of the URL identifies the website by its domain name. In this example, the domain name is example.com. The "www." part at the beginning is the particular host name at that domain. The host name "www" has become a convention, but is not a rule. In fact, sometimes the host name may be omitted. There can be more than one website at a domain (sometimes called subdomains). For example, there might also be *development.example.com*, *clients.example.com*, and so on.

❸ `/2012/samples/first.html`

This is the absolute path through directories on the server to the requested HTML document, *first.html*. The words separated by slashes are the directory names, starting with the root directory of the host (as indicated by the initial `/`). Because the Internet originally comprised computers running the Unix operating system, our current way of doing things still

Hey, There's No http:// on That URL!

Because nearly all web pages use the Hypertext Transfer Protocol, the **`http://`** part is often just implied. This is the case when site names are advertised in print or on TV, as a way to keep the URL easy to remember.

Additionally, browsers are programmed to add **`http://`** automatically as a convenience to save you some keystrokes. It may seem like you're leaving it out, but it is being sent to the server behind the scenes.

When we begin using URLs to create hyperlinks in HTML documents in Chapter 6, Adding Links, you'll learn that it is necessary to include the protocol when making a link to a web page on another server.

NOTE

Sometimes you'll see a URL that begins with `https://`*. This is an indication that it is a secure server transaction. Secure servers have special encryption devices that hide delicate content, such as credit card numbers, while they are transferred to and from the browser. Look for it the next time you're shopping online.*

follows many Unix rules and conventions, hence the / separating directory names.

To sum it up, the URL in Figure 2-1 says it would like to use the HTTP protocol to connect to a web server on the Internet called *www.example.com* and request the document *first.html* (located in the *samples* directory, which is in the *2012* directory).

Default files

Obviously, not every URL you see is so lengthy. Many addresses do not include a filename, but simply point to a directory, like these:

```
http://www.oreilly.com
http://www.jendesign.com/resume/
```

When a server receives a request for a directory name rather than a specific file, it looks in that directory for a default document, typically named *index.html*. So when someone types the above URLs into their browser, what they'll actually see is this:

```
http://www.oreilly.com/index.html
http://www.jendesign.com/resume/index.html
```

The name of the default file (also referred to as the index file) may vary, and depends on how the server is configured. In these examples, it is named *index.html*, but some servers use the filename *default.htm*. If your site uses server-side programming to generate pages, the index file might be named *index.php* or *index.asp*. Just check with your server administrator or the tech support department at your hosting service to make sure you give your default file the proper name.

Another thing to notice is that in the first example, the original URL did not have a trailing slash to indicate it was a directory. When the slash is omitted, the server simply adds one if it finds a directory with that name.

The index file is also useful for security. Some servers (depending on their configuration) display the contents of the directory if the default file is not found. Figure 2-2 shows how the documents in the *housepics* directory are exposed as the result of a missing default file. One way to prevent people from snooping around in your files is to be sure there is an index file in every directory. Your server administrator may also add other protections to prevent your directories from displaying in the browser.

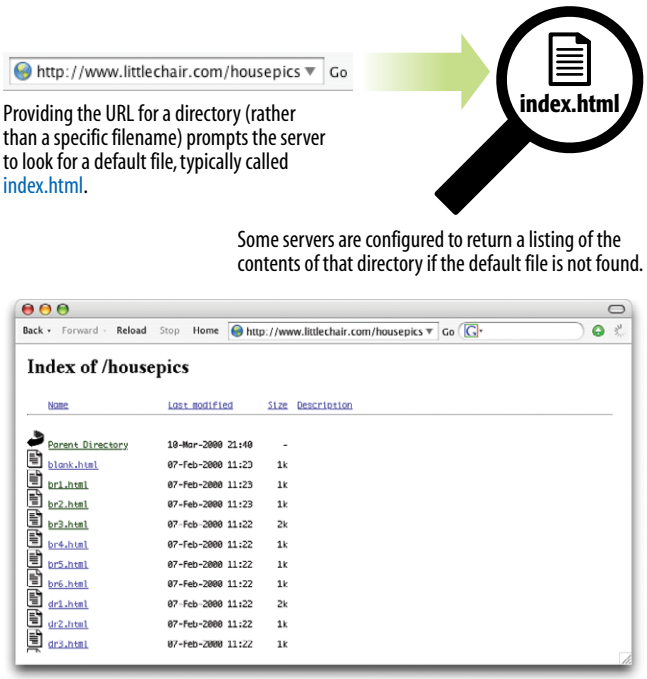


Figure 2-2. Some servers display the contents of the directory if an index file is not found.

The Anatomy of a Web Page

We're all familiar with what web pages look like in the browser window, but what's happening "under the hood?"

At the top of Figure 2-3, you see a minimal web page as it appears in a graphical browser. Although you see it as one coherent page, it is actually assembled from four separate files: an HTML document (*index.html*), a style sheet (*kitchen.css*), and two graphics (*foods.gif* and *spoon.gif*). The HTML document is running the show.

HTML documents

You may be as surprised as I was to learn that the graphically rich and interactive pages we see on the Web are generated by simple, text-only documents. This text file is referred to as the source document.

Take a look at *index.html*, the source document for the Jen's Kitchen web page. You can see it contains the text content of the page plus special tags (indicated with angle brackets, < and >) that describe each element on the page.

Adding descriptive tags to a text document is known as "marking up" the document. Web pages use a markup language called HyperText Markup Language, or HTML for short, which was created especially for documents with hypertext links. HTML defines dozens of text elements that make up documents such as headings, paragraphs, emphasized text, and of course, links. There are also elements that add information about the document (such as its title), media such as images and videos, and widgets for form inputs, just to name a few.

It is worth noting briefly that there are actually several versions of HTML in use today. The most firmly established are HTML version 4.01 and its stricter cousin, XHTML 1.0. And you may have heard how all the Web is a-buzz with the emerging HTML5 specification that is designed to better handle web applications and is gradually gaining browser support. I will give you the lowdown on all the various versions and what makes them unique in Chapter 10, What's Up, HTML5?. In the meantime, we have to cover some basics that apply regardless of the HTML flavor you choose.

A quick introduction to HTML markup

You'll be learning the nitty-gritty of markup in Part II, so I don't want to bog you down with too much detail right now, but there are a few things I'd like to point out about how HTML works and how browsers interpret it.

Read through the HTML document in Figure 2-3 and compare it to the browser results. It's easy to see how the elements marked up with HTML tags in the source document correspond to what displays in the browser window.

exercise 2-1 | View source

You can see the HTML file for any web page by choosing View → Page Source or (View → Source) in your browser's menu. Your browser typically opens the source document in a separate window. Let's take a look under the hood of a web page.

1. Enter this URL into your browser:

   www.learningwebdesign.com/4e/materials/chapter02/kitchen.html

   You should see the Jen's Kitchen web page from Figure 2-3.
2. Select View → Page Source (or View → Source) from the browser menu. On Chrome and Opera, View Source is located in the Developer menu. A window opens showing the source document shown in the figure.
3. The source for most sites is considerably more complicated. View the source of *oreilly.com* or the site of your choice. Don't worry if you don't understand what's going on. Much of it will look more familiar by the time you are done with this book.

WARNING

Keep in mind that while learning from others' work is fine, the all-out stealing of other people's code is poor form (or even illegal). If you want to use code as you see it, ask for permission and always give credit to those who did the work.

The web page shown in this browser window consists of four separate files: an HTML text document, a style sheet and two images. Tags in the HTML source document give the browser instructions for how the text is structured and where the images should be placed.

index.html

```
<!DOCTYPE html>
<html>
<head>
<title>Jen's Kitchen</title>
<link rel="stylesheet" href="kitchen.css" type="text/css" >
</head>

<body>
<h1><img src="foods.gif" alt="food illustration"> Jen&rsquo;s Kitchen</h1>

<p>If you love to read about <strong>cooking and eating</strong>, would like to find out about
of some of the best restaurants in the world, or just want a few choice recipes to add to your
collection, <em>this is the site for you!</em></p>

<p><img src="spoon.gif" alt="spoon illustration"> Your pal, Jen at Jen's Kitchen</p>
<hr>
<p><small>Copyright 2011, Jennifer Robbins</small></p>
</body>
</html>
```

kitchen.css

```
body { font: normal 1em Verdana; margin: 1em 10%;}
h1 { font: italic 3em Georgia; color: rgb(23, 109, 109); margin: 1em 0 1em;}
img { margin: 0 20px 0 0; }
h1 img { margin-bottom: -20px; }
small { color: #666666; }
```

foods.gif

spoon.gif

Figure 2-3. The source file and images that make up a simple web page.

First, you'll notice that the text within brackets (for example, **<body>**) does not display in the final page. The browser displays only what's between the tags—the content of the element. The markup is hidden. The tag provides the name of the HTML element—usually an abbreviation such as "h1" for "heading level 1," or "em" for "emphasized text."

Second, you'll see that most of the HTML tags appear in pairs surrounding the content of the element. In our HTML document, **<h1>** indicates that the following text should be a level-1 heading; **</h1>** indicates the end of the heading. Some elements, called empty elements, do not have content. In our sample, the **<hr>** tag indicates an empty element that tells the browser to "insert a thematic divider here" (most browsers indicate the thematic divider with a horizontal rule [line], which is how the **hr** element got its initials).

Because I was unfamiliar with computer programming when I first began writing HTML, it helped me to think of the tags and text as "beads on a string" that the browser interprets one by one, in sequence. For example, when the browser encounters an open bracket (**<**), it assumes all of the following characters are part of the markup until it finds the closing bracket (**>**). Similarly, it assumes all of the content following an opening **<h1>** tag is a heading until it encounters the closing **</h1>** tag. This is the manner in which the browser parses the HTML document. Understanding the browser's method can be helpful when troubleshooting a misbehaving HTML document.

But where are the pictures?

Obviously, there are no pictures in the HTML file itself, so how do they get there when you view the final page?

You can see in Figure 2-3 that each image is a separate file. The images are placed in the flow of the text with the HTML image element (**img**) that tells the browser where to find the graphic (its URL). When the browser sees the **img** element, it makes another request to the server for the image file, and then places it in the content flow. The browser software brings the separate pieces together into the final page. Videos and other embedded media files are added in much the same way.

The assembly of the page generally happens in an instant, so it appears as though the whole page loads all at once. Over slow connections or if the page includes huge graphics or media files, the assembly process may be more apparent as images lag behind the text. The page may even need to be redrawn as new images arrive (although you can construct your pages in a way to prevent that from happening).

Adding a little style

I want to direct your attention to one last key ingredient of our minimal page. Near the top of the HTML document there is a `link` element that points to the style sheet document *kitchen.css*. That style sheet includes a few lines of instructions for how the page should look in the browser. These are style instructions written according to the rules of Cascading Style Sheets (CSS). CSS allows designers to add visual style instructions (known as the document's presentation) to the marked-up text (the document's structure, in web design terminology). In Part III, you'll really get to know the power of Cascading Style Sheets.

Figure 2-4 shows the Jen's Kitchen page with and without the style instructions. Browsers come equipped with default styles for every HTML element they support, so if an HTML document lacks its own custom style instructions, the browser will use its own (that's what you see in the screen shot on the right). Even just a few style rules can make big improvements to the appearance of a page.

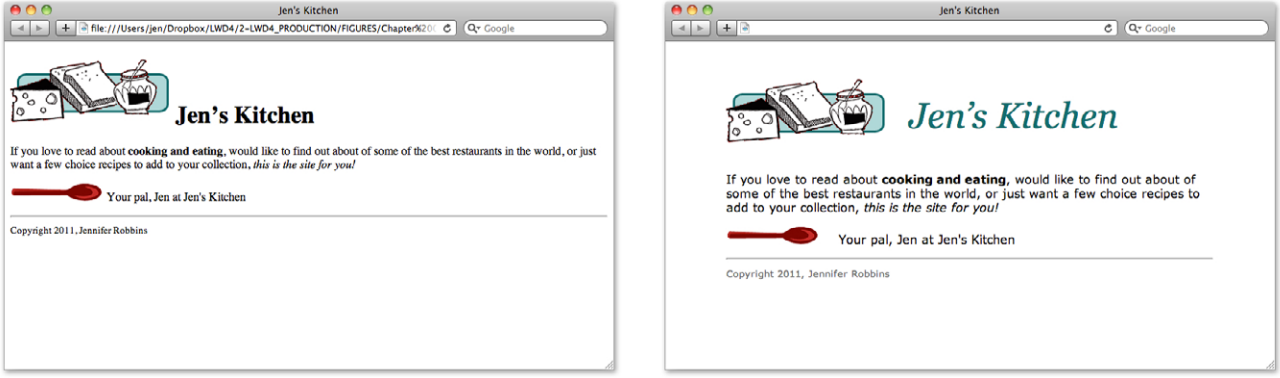


Figure 2-4. The Jen's Kitchen page before (left) and after (right) style rules.

Adding Behaviors with JavaScript

In addition to a document's structure and presentation, there is also a behavior component that defines how things *work*. On the Web, behaviors are defined by a scripting language called JavaScript. We'll touch on it lightly in this book in Part IV, but learning JavaScript from scratch is more than we can take on here. Many designers (myself included) rely on people with scripting experience to add functionality to sites. However, knowing how to write JavaScript is becoming more essential to the "web designer" job description.

Putting It All Together

To wrap up our introduction to how the web works, let's trace a typical stream of events that occurs with every web page that appears on your screen (Figure 2-5).

1. You request a web page by either typing its URL (for example, *http://jenskitchensite.com*) directly in the browser or by clicking on a link on a page. The URL contains all the information needed to target a specific document on a specific web server on the Internet.
2. Your browser sends an HTTP Request to the server named in the URL and asks for the specific file. If the URL specifies a directory (not a file), it is the same as requesting the default file in that directory.
3. The server looks for the requested file and issues an HTTP response.
   a. If the page cannot be found, the server returns an error message. The message typically says "404 Not Found," although more hospitable error messages may be provided.
   b. If the document *is* found, the server retrieves the requested file and returns it to the browser.
4. The browser parses the HTML document. If the page contains images (indicated by the HTML `img` element) or other external resources like scripts, the browser contacts the server again to request each resource specified in the markup.
5. The browser inserts each image in the document flow where indicated by the `img` element. And *voila*! The assembled web page is displayed for your viewing pleasure.

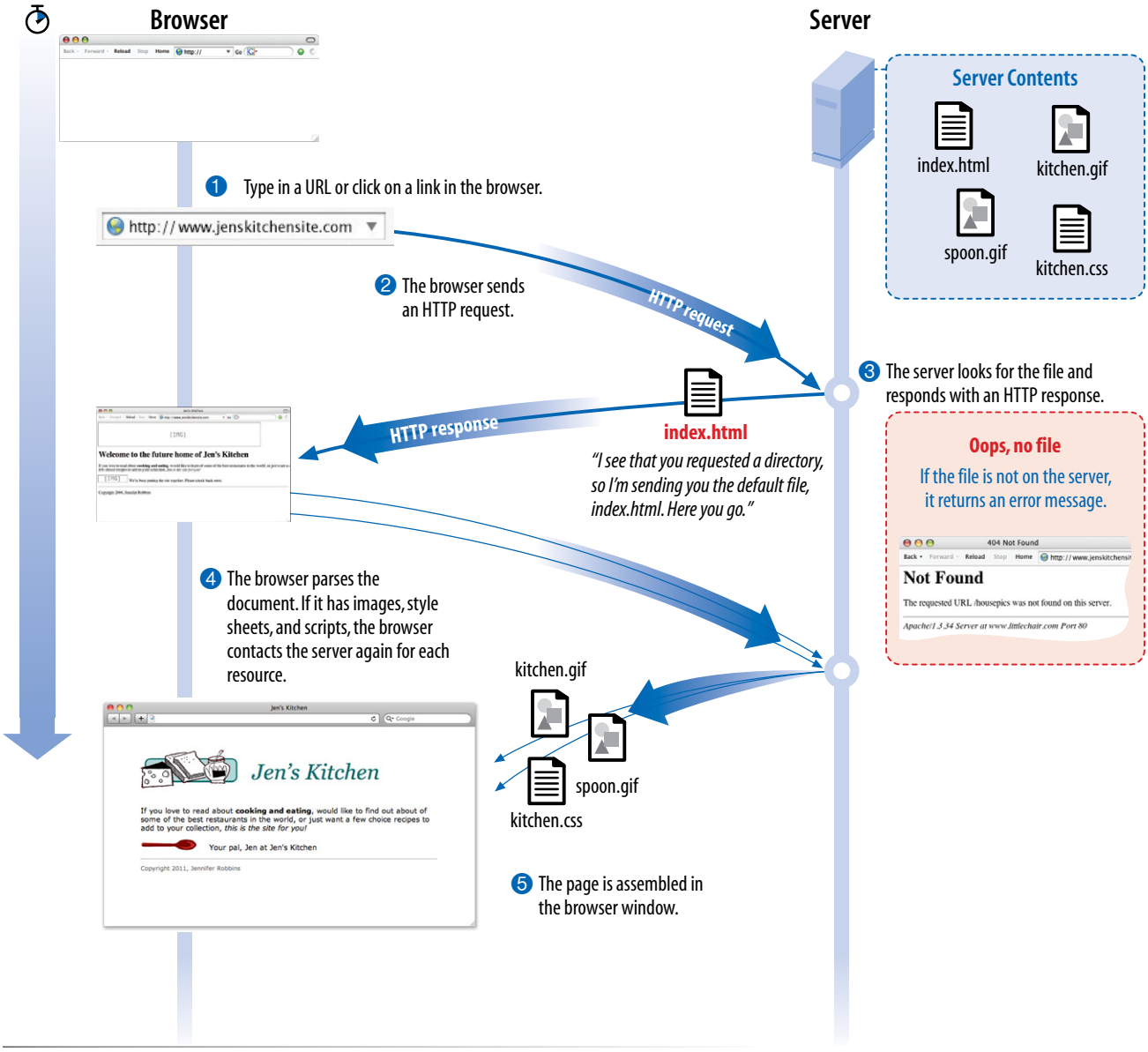


Figure 2-5. How browsers display web pages.

Test Yourself

Let's play a round of "Identify that Acronym!" The following are a few basic web terms mentioned in this chapter. Answers are in Appendix A.

1) HTML	_____	a) Home of Mosaic, the first graphical browser
2) W3C	_____	b) The location of a web document or resource
3) CERN	_____	c) The markup language used to describe web content
4) CSS	_____	d) Matches domain names with numeric IP addresses
5) HTTP	_____	e) A protocol for file transfer
6) IP	_____	f) Protocol for transferring web documents on the Internet
7) URL	_____	g) The language used to instruct how web content looks
8) NCSA	_____	h) Particle physics lab where the Web was born
9) DNS	_____	i) Internet Protocol
10) FTP	_____	j) The organization that monitors web technologies

CHAPTER 4

CREATING A SIMPLE PAGE

(HTML Overview)

Part I provided a general overview of the web design environment. Now that we've covered the big concepts, it's time to roll up our sleeves and start creating a real web page. It will be an extremely simple page, but even the most complicated pages are based on the principles described here.

In this chapter, we'll create a web page step by step so you can get a feel for what it's like to mark up a document with HTML tags. The exercises allow you to work along.

This is what I want you to get out of this chapter:

- Get a feel for how markup works, including an understanding of elements and attributes.
- See how browsers interpret HTML documents.
- Learn the basic structure of an HTML document.
- Get a first glimpse of a style sheet in action.

Don't worry about learning the specific text elements or style sheet rules at this point; we'll get to those in the following chapters. For now, just pay attention to the process, the overall structure of the document, and the new terminology.

IN THIS CHAPTER

An introduction to elements and attributes

A step-by-step demo of marking up a simple web page

The elements that provide document structure

A simple stylesheet

Troubleshooting broken web pages

A Web Page, Step by Step

You got a look at an HTML document in Chapter 2, How the Web Works, but now you'll get to create one yourself and play around with it in the browser. The demonstration in this chapter has five steps that cover the basics of page production.

Step 1: Start with content. As a starting point, we'll write up raw text content and see what browsers do with it.

Step 2: Give the document structure. You'll learn about HTML element syntax and the elements that give a document its structure.

HTML the Hard Way

I stand by my method of teaching HTML the old-fashioned way—*by hand*. There's no better way to truly understand how markup works than typing it out, one tag at a time, then opening your page in a browser. It doesn't take long to develop a feel for marking up documents properly.

Although you may choose to use a web-authoring tool down the line, understanding HTML will make using your tools easier and more efficient. In addition, you will be glad that you can look at a source file and understand what you're seeing. It is also crucial for troubleshooting broken pages or fine-tuning the default formatting that web tools produce.

And for what it's worth, professional web developers tend to mark up content manually because it gives them better control over the code and allows them to make deliberate decisions about what elements are used.

Step 3: Identify text elements. You'll describe the content using the appropriate text elements and learn about the proper way to use HTML.

Step 4: Add an image. By adding an image to the page, you'll learn about attributes and empty elements.

Step 5: Change the page appearance with a style sheet. This exercise gives you a taste of formatting content with Cascading Style Sheets.

By the time we're finished, you will have written the source document for the page shown in Figure 4-1. It's not very fancy, but you have to start somewhere.

We'll be checking our work in a browser frequently throughout this demonstration—probably more than you would in real life. But because this is an introduction to HTML, it is helpful to see the cause and effect of each small change to the source file along the way.

Before We Begin, Launch a Text Editor

In this chapter and throughout the book, we'll be writing out HTML documents by hand, so the first thing we need to do is launch a text editor. The text editor that is provided with your operating system, such as Notepad (Windows) or TextEdit (Macintosh), will do for these purposes. Other text editors are fine as long as you can save plain text files with the *.html* extension. If you have a WYSIWYG web-authoring tool such as Dreamweaver, set it aside for now. I want you to get a feel for marking up a document manually (see the sidebar "HTML the Hard Way").

This section shows how to open new documents in Notepad and TextEdit. Even if you've used these programs before, skim through for some special settings that will make the exercises go more smoothly. We'll start with Notepad; Mac users can jump ahead.

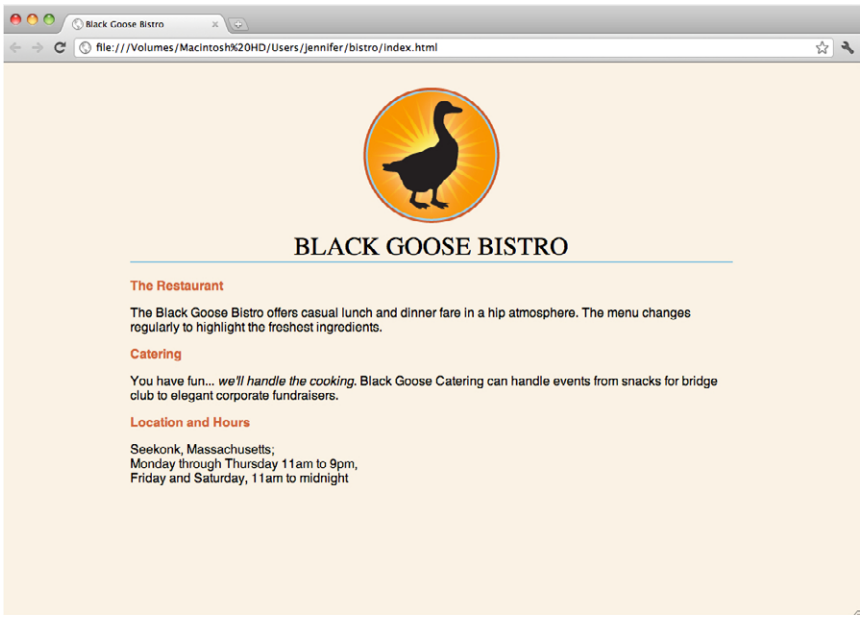


Figure 4-1. *In this chapter, we'll write the source document for this page step by step.*

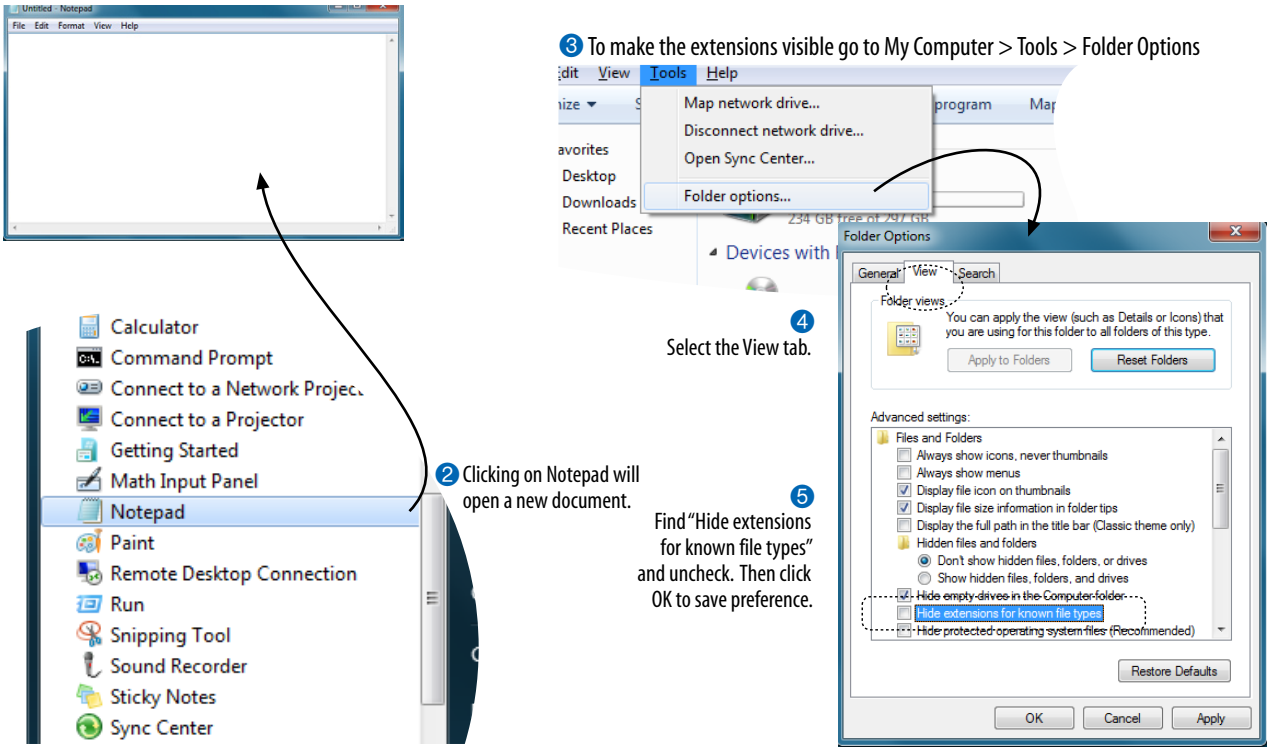
Creating a new document in Notepad (Windows)

These are the steps to creating a new document in Notepad on Windows 7 (Figure 4-2):

1. Open the Start menu and navigate to Notepad (in Accessories). ❶
2. Click on Notepad to open a new document window, and you're ready to start typing. ❷
3. Next, we'll make the extensions visible. This step is not required to make HTML documents, but it will help make the file types clearer at a glance. Select "Folder Options..." from the Tools menu ❸ and select the View tab ❹. Find "Hide extensions for known file types" and uncheck that option. ❺ Click OK to save the preference, and the file extensions will now be visible.

NOTE

In Windows 7, hit the ALT key to reveal the menu to access Tools and Folder Options. In Windows Vista, it is labeled "Folder and Search Options."



❶ Open the Start menu and navigate to Notepad *(All Programs > Accessories > Notepad)*

Figure 4-2. Creating a new document in Notepad.

Creating a new document in TextEdit (Mac OS X)

By default, TextEdit creates "rich text" documents, that is, documents that have hidden style formatting instructions for making text bold, setting font size, and so on. You can tell that TextEdit is in rich text mode when it has a formatting toolbar at the top of the window (plain text mode does not). HTML documents need to be plain text documents, so we'll need to change the Format, as shown in this example (Figure 4-3).

1. Use the Finder to look in the Applications folder for TextEdit. When you've found it, double-click the name or icon to launch the application.
2. TextEdit opens a new document. The text-formatting menu at the top shows that you are in Rich Text mode. Here's how you change it.
3. Open the Preferences dialog box from the TextEdit menu.
4. There are three settings you need to adjust:

   On the "New Document" tab, select "Plain text".

   On the "Open and Save" tab, select "Ignore rich text commands in HTML files" and turn off "Append '.txt' extensions to plain text files".
5. When you are done, click the red button in the top-left corner.
6. When you create a new document, the formatting menu will no longer be there and you can save your text as an HTML document. You can always convert a document back to rich text by selecting Format → Make Rich Text when you are not using TextEdit for HTML.

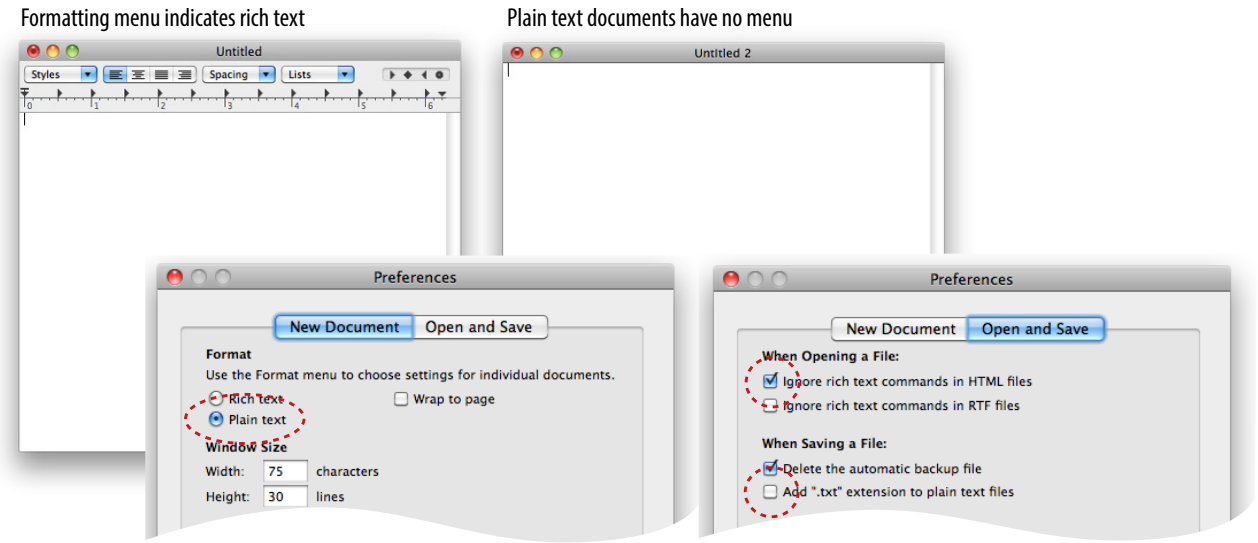


Figure 4-3. Launching TextEdit and choosing Plain Text settings in the Preferences.

Step 1: Start with Content

Now that we have our new document, it's time to get typing. A web page always starts with content, so that's where we begin our demonstration. Exercise 4-1 walks you through entering the raw text content and saving the document in a new folder.

exercise 4-1 | Entering content

1. Type the content below for the home page into the new document in your text editor. Copy it exactly as you see it here, keeping the line breaks the same for the sake of playing along. The raw text for this exercise is available online at *www.learningwebdesign.com/4e/materials/*.

```
Black Goose Bistro

The Restaurant
The Black Goose Bistro offers casual lunch and dinner fare in hip
atmosphere. The menu changes regularly to highlight the freshest
ingredients.

Catering
You have fun... we'll handle the cooking. Black Goose Catering
can handle events from snacks for bridge club to elegant corporate
fundraisers.

Location and Hours
Seekonk, Massachusetts;
Monday through Thursday 11am to 9pm, Friday and Saturday, 11am to
midnight
```

2. Select "Save" or "Save as" from the File menu to get the Save As dialog box (Figure 4-4). The first thing you need to do is create a new folder that will contain all of the files for the site (in other words, it's the local root folder).

   Windows: Click the folder icon at the top to create the new folder.

   Mac: Click the "New Folder" button.

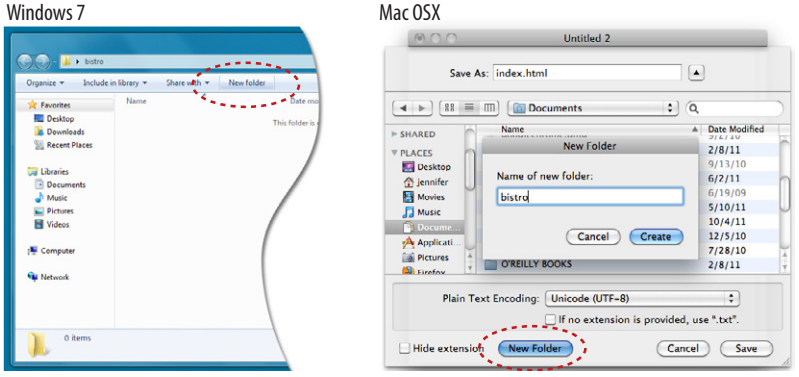


Figure 4-4. Saving index.html in a new folder called "bistro".

Naming Conventions

It is important that you follow these rules and conventions when naming your files:

Use proper suffixes for your files. HTML and XHTML files must end with .html. Web graphics must be labeled according to their file format: *.gif*, *.png*, or *.jpg* (*.jpeg* is also acceptable).

Never use character spaces within filenames. It is common to use an underline character or hyphen to visually separate words within filenames, such as *robbins_bio html* or *robbins-bio html*.

Avoid special characters such as **?, %, #, /, :, ;, •,** etc. Limit filenames to letters, numbers, underscores, hyphens, and periods.

Filenames may be case-sensitive, depending on your server configuration. Consistently using all lowercase letters in filenames, although not necessary, is one way to make your filenames easier to manage.

Keep filenames short. Short names keep the character count and file size of your HTML file in check. If you really must give the file a long, multiword name, you can separate words with hyphens, such as *a-long-document-title.html*, to improve readability.

Self-imposed conventions. It is helpful to develop a consistent naming scheme for huge sites. For instance, always using lowercase with hyphens between words. This takes some of the guesswork out of remembering what you named a file when you go to link to it later.

Name the new folder ***bistro***, and save the text file as ***index.html*** in it. Windows users, you will also need to choose "All Files" after "Save as type" to prevent Notepad from adding a ".txt" extension to your filename. The filename needs to end in ***.html*** to be recognized by the browser as a web document. See the sidebar "Naming Conventions" for more tips on naming files.

3. Just for kicks, let's take a look at ***index.html*** in a browser. Launch your favorite browser (I'm using Google Chrome) and choose "Open" or "Open File" from the File menu. Navigate to ***index.html***, and then select the document to open it in the browser. You should see something like the page shown in Figure 4-5. We'll talk

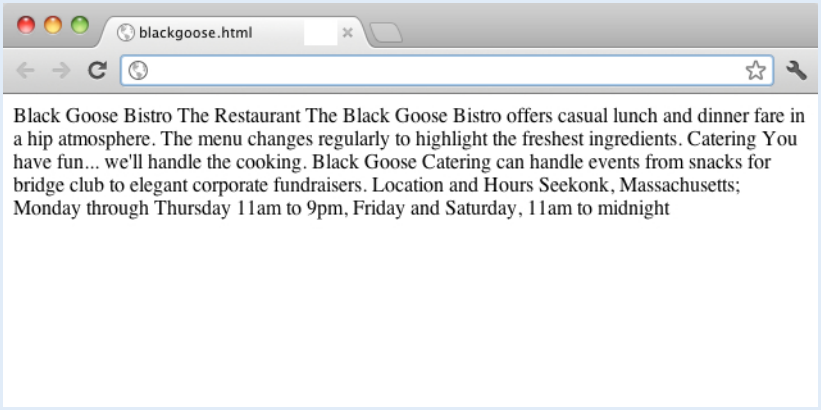


Figure 4-5. *A first look at the content in a browser.*

Learning from step 1

Our content isn't looking so good (Figure 4-5). The text is all run together—that's not how it looked in the original document. There are a couple of things to be learned here. The first thing that is apparent is that the browser ignores line breaks in the source document. The sidebar "What Browsers Ignore" lists other information in the source that is not displayed in the browser window.

Second, we see that simply typing in some content and naming the document *.html* is not enough. While the browser can display the text from the file, we haven't indicated the *structure* of the content. That's where HTML comes in. We'll use markup to add structure: first to the HTML document itself (coming up in Step 2), then to the page's content (Step 3). Once the browser knows the structure of the content, it can display the page in a more meaningful way.

What Browsers Ignore

Some information in the source document will be ignored when it is viewed in a browser, including:

Multiple (white) spaces. When a browser encounters more than one consecutive blank character space, it displays a single space. So if the document contains:

```
long,     long     ago
```

the browser displays:

long, long ago

Line breaks (carriage returns). Browsers convert carriage returns to white spaces, so following the earlier "ignore multiple white spaces rule," line breaks have no effect on formatting the page. Text and elements wrap continuously until a new block element, such as a heading (`h1`) or paragraph (`p`), or the line break (`br`) element is encountered in the flow of the document text.

Tabs. Tabs are also converted to character spaces, so guess what? Useless.

Unrecognized markup. Browsers are instructed to ignore any tag they don't understand or that was specified incorrectly. Depending on the element and the browser, this can have varied results. The browser may display nothing at all, or it may display the contents of the tag as though it were normal text.

Text in comments. Browsers will not display text between the special `<!--` and `-->` tags used to denote a comment. See the Adding Hidden Comments sidebar later in this chapter.

Step 2: Give the Document Structure

We have our content saved in an *.html* document—now we're ready to start marking it up.

Introducing…HTML elements

An element consists of both the content and its markup.

Back in Chapter 2, How the Web Works, you saw examples of HTML elements with an opening tag (** ** for a paragraph, for example) and closing tag (** **). Before we start adding tags to our document, let's look at the anatomy of an HTML element (its syntax) and firm up some important terminology. A generic container element is labeled in Figure 4-6.

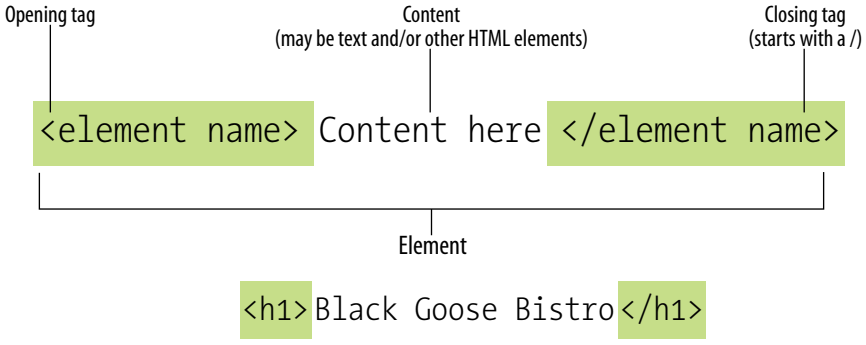


Figure 4-6. *The parts of an HTML container element.*

Elements are identified by tags in the text source. A tag consists of the element name (usually an abbreviation of a longer descriptive name) within angle brackets (**< >**). The browser knows that any text within brackets is hidden and not displayed in the browser window.

The element name appears in the opening tag (also called a start tag) and again in the closing (or end) tag preceded by a slash (**/**). The closing tag works something like an "off" switch for the element. Be careful not to use the similar backslash character in end tags (see the tip Slash vs. Backslash).

The tags added around content are referred to as the markup. It is important to note that an element consists of both the content *and* its markup (the start and end tags). Not all elements have content, however. Some are empty by definition, such as the **img** element used to add an image to the page. We'll talk about empty elements a little later in this chapter.

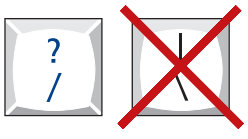
One last thing...capitalization. In HTML, the capitalization of element names is not important. So **<img>**, **<Img>**, and **<IMG>** are all the same as far as the browser is concerned. However, in XHTML (the stricter version of HTML) all element names must be all lowercase in order to be valid. Many web developers have come to like the orderliness of the stricter XHTML markup rules and stick with all lowercase, as I will do in this book.

TIP

Slash vs. Backslash

HTML tags and URLs use the slash character (/). The slash character is found under the question mark (?) on the standard QWERTY keyboard.

It is easy to confuse the slash with the backslash character (\), which is found under the bar character (|). The backslash key will not work in tags or URLs, so be careful not to use it.



Basic document structure

Figure 4-7 shows the recommended minimal skeleton of an HTML5 document. I say "recommended" because the only element that is *required* in HTML is the **title**. But I feel it is better, particularly for beginners, to explicitly organize documents with the proper structural markup. And if you are writing in the stricter XHTML, all of the following elements except **meta** must be included in order to be valid. Let's take a look at what's going on in Figure 4-7.

❶ I don't want to confuse things, but the first line in the example isn't an element at all; it is a document type declaration (also called DOCTYPE declaration) that identifies this document as an HTML5 document. I have a lot more to say about DOCTYPE declarations in Chapter 10, What's Up, HTML5?, but for this discussion, suffice it to say that including it lets modern browsers know they should interpret the document as written according to the HTML5 specification.

❷ The entire document is contained within an **html** element. The **html** element is called the root element because it contains all the elements in the document, and it may not be contained within any other element. It is used for both HTML and XHTML documents.

❸ Within the **html** element, the document is divided into a head and a body. The **head** element contains descriptive information about the document itself, such as its title, the style sheet(s) it uses, scripts, and other types of "meta" information.

❹ The **meta** elements within the **head** element provide information *about* the document itself. A **meta** element can be used to provide all sorts of information, but in this case, it specifies the character encoding (the standardized collection of letters, numbers, and symbols) used in the document. I don't want to go into too much detail on this right now, but know that there are many good reasons for specifying the **charset** in every document, so I have included it as part of the minimal document structure.

❺ Also in the **head** is the mandatory **title** element. According to the HTML specification, every document must contain a descriptive title.

❻ Finally, the **body** element contains everything that we want to show up in the browser window.

Are you ready to add some structure to the Black Goose Bistro home page? Open the *index.html* document and move on to Exercise 4-2.

NOTE

Prior to HTML5, the syntax for specifying the character set with the **meta** *element was a bit more elaborate. If you are writing your documents in HTML 4.01 or XHTML 1.0, your* **meta** *element should look like this:*

```
<meta http-equiv="content-
type" content="text/html;
charset=UTF-8">
```

Figure 4-7. The minimal structure of an HTML document.

```
❶ <!DOCTYPE html>

  <html>

❸   <head>
❹   <meta charset="utf-8">
❺   <title>Title here</title>
    </head>

❷

❻   <body>
    Page content goes here.
    </body>

  </html>
```

exercise 4-2 | Adding basic structure

1. Open the newly created document, *index html*, if it isn't open already.
2. Start by adding the HTML5 DOCTYPE declaration:

   ```
   <!DOCTYPE html>
   ```

3. Put the entire document in an HTML root element by adding an **<html>** start tag at the very beginning and an end **<html>** tag at the end of the text.
4. Next, created the document head that contains the title for the page. Insert **<head>** and **</head>** tags before the content. Within the head element, add informatino about the character encoding **<meta charset="utf-8">**, and the title, "Black Goose Bistro", surrounded by opening and closing **<title>** tags.

   The correct terminology is to say that the **title** *element is nested within the* **head** *element. We'll talk about nesting more in later chapters.*

5. Finally, define the body of the document by wrapping the content in **<body>** and **</body>** tags. When you are done, the source document should look like this (the markup is shown in color to make it stand out):

   ```
   <!DOCTYPE html>
   <html>

   <head>
   <meta charset ="utf-8">
   <title>Black Goose Bistro</title>
   </head>

   <body>
   Black Goose Bistro

   The Restaurant
   The Black Goose Bistro offers casual lunch and dinner fare in
   a hip atmosphere. The menu changes regularly to highlight the
   freshest ingredients.

   Catering Services
   You have fun... we'll do the cooking. Black Goose catering can
   handle events from snacks for bridge club to elegant corporate
   fundraisers.
   Location and Hours
   Seekonk, Massachusetts;
   Monday through Thursday 11am to 9pm, Friday and Saturday, 11am to
   midnight
   </body>

   </html>
   ```

6. Save the document in the bistro directory, so that it overwrites the old version. Open the file in the browser or hit "refresh" or "reload" if it is open already. Figure 4-8 shows how it should look now.

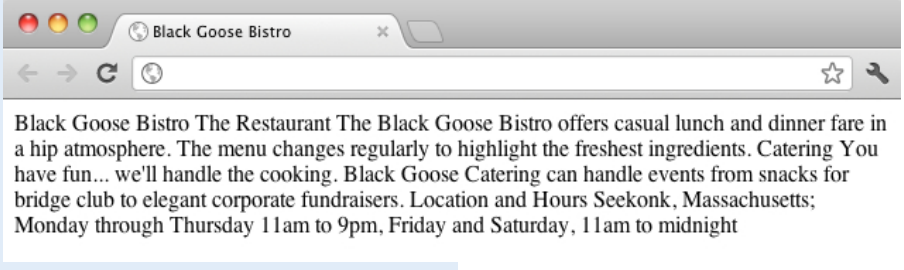


Figure 4-8. The page in a browser after the document structure elements have been defined.

Not much has changed after structuring the document, except that the browser now displays the title of the document in the top bar or tab. If someone were to bookmark this page, that title would be added to his Bookmarks or Favorites list as well (see the sidebar Don't Forget a Good Title). But the content still runs together because we haven't given the browser any indication of how it should be structured. We'll take care of that next.

Step 3: Identify Text Elements

With a little markup experience under your belt, it should be a no-brainer to add the markup that identifies headings and subheads (`h1` and `h2`), paragraphs (`p`), and emphasized text (`em`) to our content, as we'll do in Exercise 4-3. However, before we begin, I want to take a moment to talk about what we're doing and not doing when marking up content with HTML.

Introducing…semantic markup

The purpose of HTML is to add meaning and structure to the content. It is *not* intended to provide instructions for how the content should look (its presentation).

Your job when marking up content is to choose the HTML element that provides the most meaningful description of the content at hand. In the biz, we call this semantic markup. For example, the most important heading at the beginning of the document should be marked up as an `h1` because it is the most important heading on the page. Don't worry about what that looks like in the browser...you can easily change that with a style sheet. The important thing is that you choose elements based on what makes the most sense for the content.

In addition to adding meaning to content, the markup gives the document structure. The way elements follow each other or nest within one another creates relationships between the elements. You can think of it as an outline (its technical name is the DOM, for Document Object Model). The underlying document hierarchy is important because it gives browsers cues on how to handle the content. It is also the foundation upon which we add presentation instructions with style sheets and behaviors with JavaScript. We'll talk about document structure more in Part III, when we discuss Cascading Style Sheets, and in Part IV in the JavaScript overview.

Although HTML was intended to be used strictly for meaning and structure since its creation, that mission was somewhat thwarted in the early years of the web. With no style sheet system in place, HTML was extended to give authors ways to change the appearance of fonts, colors, and alignment using markup alone. Those presentational extras are still out there, so you may run across them if you view the source of older sites or a site made with old tools.

Don't Forget a Good Title

Not only is a `title` element required for every document, it is quite useful as well. The title is what is displayed in a user's Bookmarks or Favorites list and on tabs in desktop browsers. Descriptive titles are also a key tool for improving accessibility, as they are the first thing a person hears when using a screen reader. Search engines rely heavily on document titles as well. For these reasons, it's important to provide thoughtful and descriptive titles for all your documents and avoid vague titles, such as "Welcome" or "My Page." You may also want to keep the length of your titles in check so they are able to display in the browser's title area. Another best practice is to put the part of the title with more specific information first (for example, the page description ahead of the company name) so that the page title is visible when multiple tabs are lined up in the browser window.

In this book, however, we'll focus on using HTML the right way, in keeping with the contemporary standards-based, semantic approach to web design.

OK, enough lecturing. It's time to get to work on that content in Exercise 4-3.

exercise 4-3 | Defining text elements

1. Open the document ***index.html*** in your text editor, if it isn't open already.
2. The first line of text, "Black Goose Bistro," is the main heading for the page, so we'll mark it up as a Heading Level 1 (**h1**) element. Put the opening tag, **<h1>**, at the beginning of the line and the closing tag, **</h1>**, after it, like this:

   ```
   <h1>Black Goose Bistro</h1>
   ```
3. Our page also has three subheads. Mark them up as Heading Level 2 (**h2**) elements in a similar manner. I'll do the first one here; you do the same for "Catering" and "Location and Hours".

   ```
   <h2>The Restaurant</h2>
   ```
4. Each **h2** element is followed by a brief paragraph of text, so let's mark those up as paragraph (**p**) elements in a similar manner. Here's the first one; you do the rest.

   ```
   <p>The Black Goose Bistro offers casual lunch and
   dinner fare in a hip atmosphere. The menu changes
   regularly to highlight the freshest ingredients.
   </p>
   ```
5. Finally, in the Catering section, I want to emphasize that visitors should just leave the cooking to us. To make text emphasized, mark it up in an emphasis element (**em**) element, as shown here.

   ```
   <p>You have fun... <em>we'll handle the cooking
   </em>. Black Goose Catering can handle events
   from snacks for bridge club to elegant corporate
   fundraisers.</p>
   ```
6. Now that we've marked up the document, let's save it as we did before, and open (or refresh) the page in the browser. You should see a page that looks much like the one in Figure 4-9. If it doesn't, check your markup to be sure that you aren't missing any angle brackets or a slash in a closing tag.

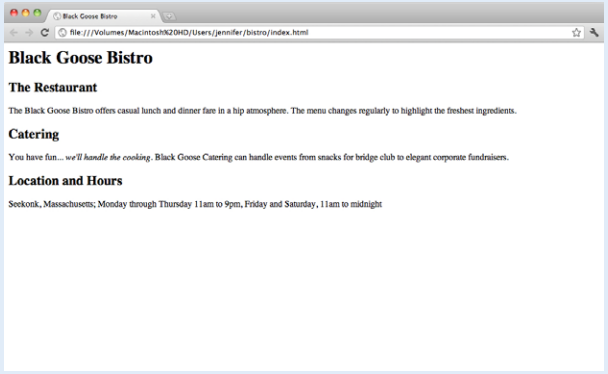


Figure 4-9. *The home page after the content has been marked up with HTML elements.*

Now we're getting somewhere. With the elements properly identified, the browser can now display the text in a more meaningful manner. There are a few significant things to note about what's happening in Figure 4-9.

Block and inline elements

Although it may seem like stating the obvious, it is worth pointing out that the heading and paragraph elements start on new lines and do not run together as they did before. That is because by default, headings and paragraphs display as block elements. Browsers treat block elements as though they are in little rectangular boxes, stacked up in the page. Each block element begins on a new line, and some space is also usually added above and below the entire element by default. In Figure 4-10, the edges of the block elements are outlined in red.

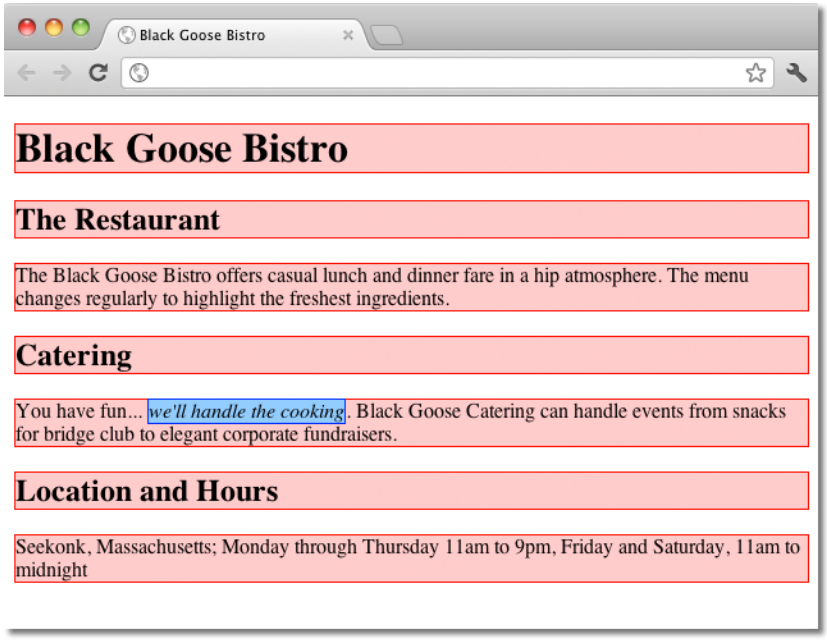


Figure 4-10. The outlines show the structure of the elements in the home page.

By contrast, look at the text we marked up as emphasized (`em`). It does not start a new line, but rather stays in the flow of the paragraph. That is because the `em` element is an inline element. Inline elements do not start new lines; they just go with the flow. In Figure 4-10, the inline `em` element is outlined in light blue.

Default styles

The other thing that you will notice about the marked-up page in Figures 4-9 and 4-10 is that the browser makes an attempt to give the page some visual hierarchy by making the first-level heading the biggest and boldest thing on the page, with the second-level headings slightly smaller, and so on.

How does the browser determine what an `h1` should look like? It uses a style sheet! All browsers have their own built-in style sheets (called user agent style sheets in the spec) that describe the default rendering of elements. The default rendering is similar from browser to browser (for example, `h1`s are always big and bold), but there are some variations (long quotes may or may not be indented).

If you think the `h1` is too big and clunky as the browser renders it, just change it with a style sheet rule. Resist the urge to mark up the heading with another element just to get it to look better, for example, using an `h3` instead of an `h1` so it isn't as large. In the days before ubiquitous style sheet support, elements were abused in just that way. Now that there are style sheets for controlling the design, you should always choose elements based on how

Adding Hidden Comments

You can leave notes in the source document for yourself and others by marking them up as comments. Anything you put between comment tags (`<!-- -->`) will not display in the browser and will not have any effect on the rest of the source.

```
<!-- This is a comment -->
<!-- This is a
  multiple-line comment
  that ends here. -->
```

Comments are useful for labeling and organizing long documents, particularly when they are shared by a team of developers. In this example, comments are used to point out the section of the source that contains the navigation.

```
<!-- start global nav -->
<ul>
 ...
</ul>
<!-- end global nav -->
```

Bear in mind that although the browser will not display comments in the web page, readers can see them if they "view source," so be sure that the comments you leave are appropriate for everyone. It's probably a good idea just to strip out notes to your fellow developers before the site is published. It cuts some bytes off the file size as well.

accurately they describe the content, and don't worry about the browser's default rendering.

We'll fix the presentation of the page with style sheets in a moment, but first, let's add an image to the page.

Step 4: Add an Image

What fun is a web page with no image? In Exercise 4-4, we'll add an image to the page using the **img** element. Images will be discussed in more detail in Chapter 7, Adding Images, but for now, it gives us an opportunity to introduce two more basic markup concepts: empty elements and attributes.

Empty elements

So far, nearly all of the elements we've used in the Black Goose Bistro home page have followed the syntax shown in Figure 4-1: a bit of text content surrounded by start and end tags.

A handful of elements, however, do not have text content because they are used to provide a simple directive. These elements are said to be empty. The image element (**img**) is an example of such an element; it tells the browser to get an image file from the server and insert it at that spot in the flow of the text. Other empty elements include the line break (**br**), thematic breaks (**hr**), and elements that provide information about a document but don't affect its displayed content, such as the **meta** element that we used earlier.

Figure 4-11 shows the very simple syntax of an empty element (compare to Figure 4-4). If you are writing an XHTML document, the syntax is slightly different (see the sidebar Empty Elements in XHTML).

Example: The `br` element inserts a line break.

```
<p>1005 Gravenstein Highway North<br>Sebastopol, CA 95472</p>
```

Figure 4-11. Empty element structure.

Empty Elements in XHTML

In XHTML, all elements, including empty elements, must be closed (or terminated, to use the proper term). Empty elements are terminated by adding a trailing slash preceded by a space before the closing bracket, like so: **<img />**, **
, **<meta />, and **<hr />**. Here is the line break example using XHTML syntax.

```
<p>1005 Gravenstein Highway
North <br />Sebastopol, CA
95472</p>
```

Attributes

Let's get back to adding an image with the empty **img** element. Obviously, an **<img>** tag is not very useful by itself—there's no way to know which image to use. That's where attributes come in. Attributes are instructions that clarify or modify an element. For the **img** element, the **src** (short for "source") attribute is required, and specifies the location (URL) of the image file.

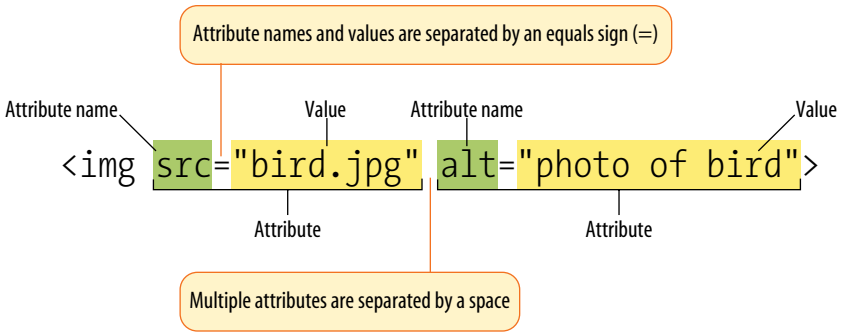


Figure 4-12. An `img` element with attributes.

The syntax for an attribute is as follows:

```
attributename="value"
```

Attributes go after the element name, separated by a space. In non-empty elements, attributes go in the opening tag only:

```
<element attributename="value">
<element attributename="value">Content</element>
```

You can also put more than one attribute in an element in any order. Just keep them separated with spaces.

```
<element attribute1="value" attribute2="value">
```

For another way to look at it, Figure 4-12 shows an **`img`** element with its required attributes labeled.

Here's what you need to know about attributes:

- Attributes go after the element name in the opening tag only, never in the end tag.
- There may be several attributes applied to an element, separated by spaces in the opening tag. Their order is not important.
- Most attributes take values, which follow an equals sign (=). In HTML, some attribute values can be reduced to single descriptive words, for example, the **`checked`** attribute, which makes a checkbox checked when a form loads. In XHTML, however, all attributes must have explicit values (**`checked="checked"`**). You may hear this type of attribute called a Boolean attribute because it describes a feature that is either on or off.
- A value might be a number, a word, a string of text, a URL, or a measurement, depending on the purpose of the attribute. You'll see examples of all of these throughout this book.
- Some values don't have to be in quotation marks in HTML, but XHTML requires them. Many developers like the consistency and tidiness of quotation marks even when authoring HTML. Either single or double quotation marks are acceptable as long as they are used consistently; however,

double quotation marks are the convention. Note that quotation marks in HTML files need to be straight (") not curly (”).

- Some attributes are required, such as the **src** and **alt** attributes in the **img** element.
- The attribute names available for each element are defined in the HTML specifications; in other words, you can't make up an attribute for an element.

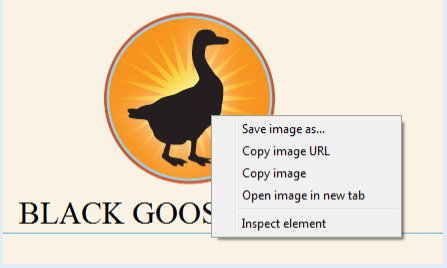
Now you should be more than ready to try your hand at adding the **img** element with its attributes to the Black Goose Bistro page in the next exercise. We'll throw a few line breaks in there as well.

exercise 4-4 | Adding an image

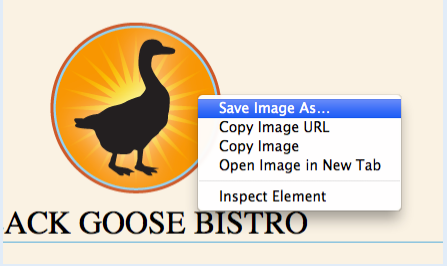
1. If you're working along, the first thing you'll need to do is get a copy of the image file on your hard drive so you can see it in place when you open the file locally. The image file is provided in the materials for this chapter. You can also get the image file by saving it right from the sample web page online at *www.learningwebdesign.com/4e/chapter04/bistro*. Right-click (or Ctrl-click on a Mac) on the goose image and select "Save to disk" (or similar) from the pop-up menu as shown in Figure 4-13. Name the file ***blackgoose.png***. Be sure to save it in the ***bistro*** folder with ***index.html***.
2. Once you have the image, insert it at the beginning of the first-level heading by typing in the **img** element and its attributes as shown here:

```
<h1><img src="blackgoose.png" alt="Black Goose logo">Black Goose
Bistro</h1>
```

   The **src** attribute provides the name of the image file that should be inserted, and the **alt** attribute provides text that should be displayed if the image is not available. Both of these attributes are required in every **img** element.



Windows:
Right-click on the image to access the pop-up menu



Mac:
Control-click on the image to access the popup menu. The options my vary by browser.

Figure 4-13. Saving an image file from a page on the Web.

3. I'd like the image to appear above the title, so lets add a line break (**br**) after the **img** element to start the headline text on a new line.

```
<h1><img src="blackgoose.png" alt="Black Goose logo"><br>Black
Goose Bistro</h1>
```

4. Let's break up the last paragraph into three lines for better clarity. Drop a **
** tag at the spots you'd like the line breaks to occur. Try to match the screenshot in Figure 4-14.
5. Now save ***index.html*** and open or refresh it in the browser window. The page should look like the one shown in Figure 4-14. If it doesn't, check to make sure that the image file, ***blackgoose.png***, is in the same directory as ***index.html***. If it is, then check to make sure that you aren't missing any characters, such as a closing quote or bracket, in the **img** element markup.

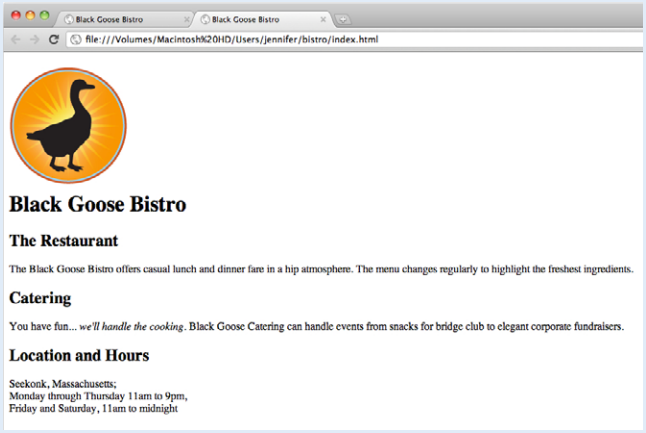


Figure 4-14. The Black Goose Bistro page with the logo image.

Step 5: Change the Look with a Style Sheet

Depending on the content and purpose of your website, you may decide that the browser's default rendering of your document is perfectly adequate. However, I think I'd like to pretty up the Black Goose Bistro home page a bit to make a good first impression on potential patrons. "Prettying up" is just my way of saying that I'd like to change its presentation, which is the job of Cascading Style Sheets (CSS).

In Exercise 4-5, we'll change the appearance of the text elements and the page background using some simple style sheet rules. Don't worry about understanding them all right now; we'll get into CSS in more detail in Part III. But I want to at least give you a taste of what it means to add a "layer" of presentation onto the structure we've created with our markup.

exercise 4-5 | Adding a style sheet

1. Open *index.html* if it isn't open already.
2. We're going to use the **style** element to apply a very simple embedded style sheet to the page. (This is just one of the ways to add a style sheet; the others are covered in Chapter 11, Style Sheet Orientation.)

   The **style** element is placed inside the **head** of the document. Start by adding the **style** element to the document as shown here:

```
<head>
  <meta charset="utf-8">
  <title>Black Goose Bistro</title>
  <style>

  </style>
</head>
```

3. Now, type the following style rules within the **style** element just as you see them here. Don't worry if you don't know exactly what is going on (although it is fairly intuitive). You'll learn all about style rules in Part III.

```
<style>

body {
  background-color: #faf2e4;
  margin: 0 15%;
  font-family: sans-serif;
  }

h1 {
  text-align: center;
  font-family: serif;
  font-weight: normal;
  text-transform: uppercase;
  border-bottom: 1px solid #57b1dc;
  margin-top: 30px;
}

h2 {
  color: #d1633c;
  font-size: 1em;
}

</style>
```

4. Now it's time to save the file and take a look at it in the browser. It should look like the page in Figure 4-15. If it doesn't, go over the style sheet code to make sure you didn't miss a semicolon or a curly bracket.

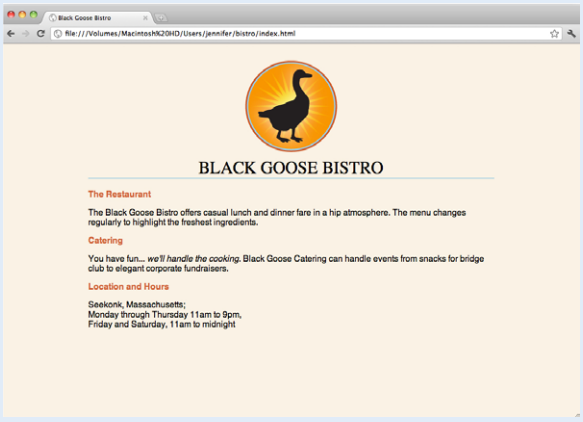


Figure 4-15. The Black Goose Bistro page after CSS style rules have been applied.

We're finished with the Black Goose Bistro page. Not only have you written your first web page, complete with a style sheet, but you've learned about elements, attributes, empty elements, block and inline elements, the basic structure of an HTML document, and the correct use of markup along the way. Not bad for one chapter!

When Good Pages Go Bad

The previous demonstration went smoothly, but it's easy for small things to go wrong when typing out HTML markup by hand. Unfortunately, one missed character can break a whole page. I'm going to break my page on purpose so we can see what happens.

NOTE

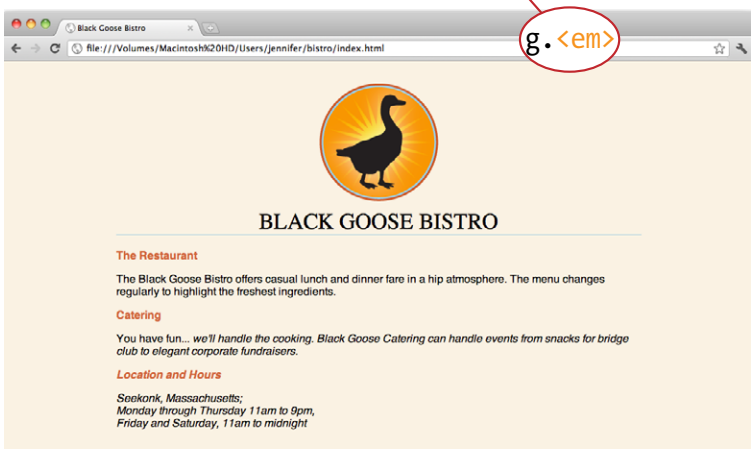
Omitting the slash in the closing tag (or even omitting the closing tag itself) for block elements, such as headings or paragraphs, may not be so dramatic. Browsers interpret the start of a new block element to mean that the previous block element is finished.

What if I had forgotten to type the slash (/) in the closing emphasis tag ()? With just one character out of place (Figure 4-16), the remainder of the document displays in emphasized (italic) text. That's because without that slash, there's nothing telling the browser to turn "off" the emphasized formatting, so it just keeps going.

I've fixed the slash, but this time, let's see what would have happened if I had accidentally omitted a bracket from the end of the first <h2> tag (Figure 4-17).

Figure 4-16. When a slash is omitted, the browser doesn't know when the element ends, as is the case in this example.

```
<h2>Catering</h2>
<p>You have fun... <em>we'll handle the cooking.<em> Black Goose
Catering can handle events from snacks for bridge club to elegant
corporate fundraisers.</p>
```

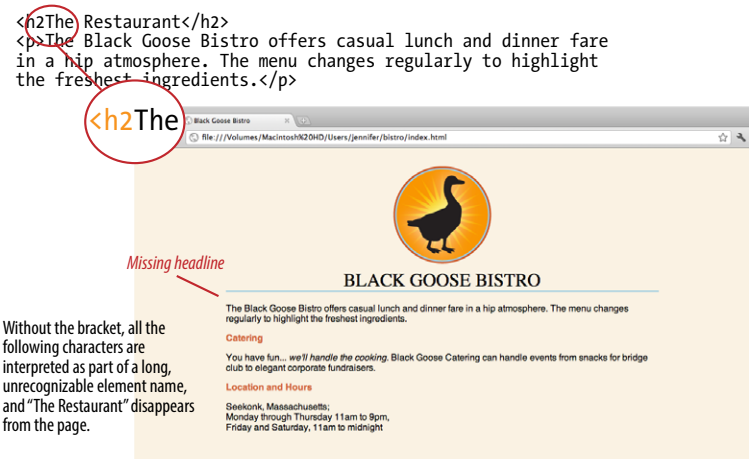


Browsers don't display any text within a tag, so my heading disappeared. The browser just ignored the foreign-looking element name and moved on to the next element.

Making mistakes in your first HTML documents and fixing them is a great way to learn. If you write your first pages perfectly, I'd recommend fiddling with the code as I have here to see how the browser reacts to various changes. This can be extremely useful in troubleshooting pages later. I've listed some common problems in the sidebar Having Problems? Note that these problems are not specific to beginners. Little stuff like this goes wrong all the time, even for the pros.

Figure 4-17. A missing end bracket makes all the following content part of the tag, and therefore it doesn't display.

```
<h2The Restaurant</h2>
<p>The Black Goose Bistro offers casual lunch and dinner fare
in a hip atmosphere. The menu changes regularly to highlight
the freshest ingredients.</p>
```



Validating Your Documents

One way that professional web developers catch errors in their markup is to validate their documents. What does that mean? To validate a document is to check your markup to make sure that you have abided by all the rules of whatever version of HTML you are using (there are more than one, as we'll discuss in Chapter 10, What's Up, HTML5?). Documents that are error-free are said to be valid. It is strongly recommended that you validate your documents, especially for professional sites. Valid documents are more consistent on a variety of browsers, they display more quickly, and they are more accessible.

Right now, browsers don't require documents to be valid (in other words, they'll do their best to display them, errors and all), but any time you stray from the standard you introduce unpredictability in the way the page is displayed or handled by alternative devices.

So how do you make sure your document is valid? You could check it yourself or ask a friend, but humans make mistakes, and you aren't really expected to memorize every minute rule in the specifications. Instead, you use a validator, software that checks your source against the HTML version you specify. These are some of the things validators check for:

- The inclusion of a DOCTYPE declaration. Without it the validator doesn't know which version of HTML or XHTML to validate against.
- An indication of the character encoding for the document.
- The inclusion of required rules and attributes.
- Non-standard elements.
- Mismatched tags.
- Nesting errors.
- Typos and other minor errors.

Developers use a number of helpful tools for checking and correcting errors in HTML documents. The W3C offers a free online validator at *validator.w3.org*. For HTML5 documents, use the online validator located at *html5.validator.nu*. Browser developer tools like the Firebug plug-in for Firefox or the built-in developer tools in Safari and Chrome also have validators so you can check your work on the fly. If you use Dreamweaver to create your sites, there is a validator built into that as well.

Having Problems?

The following are some typical problems that crop up when creating web pages and viewing them in a browser:

I've changed my document, but when I reload the page in my browser, it looks exactly the same.

It could be you didn't save your document before reloading, or you may have saved it in a different directory.

Half my page disappeared.

This could happen if you are missing a closing bracket (**>**) or a quotation mark within a tag. This is a common error when writing HTML by hand.

I put in a graphic using the `img` *element, but all that shows up is a broken image icon.*

The broken graphic could mean a couple of things. First, it might mean that the browser is not finding the graphic. Make sure that the URL to the image file is correct. (We'll discuss URLs further in Chapter 6, Adding Links.) Make sure that the image file is actually in the directory you've specified. If the file is there, make sure it is in one of the formats that web browsers can display (GIF, JPEG, or PNG) and that it is named with the proper suffix (***.gif***, ***.jpeg*** or ***.jpg***, or ***.png***, respectively).

Test Yourself

Now is a good time to make sure you understand the basics of markup. Use what you've learned in this chapter to answer the following questions. Answers are in Appendix A.

1. What is the difference between a tag and an element?

2. Write out the recommended minimal structure of an HTML5 document.

3. Indicate whether each of these filenames is an acceptable name for a web document by circling "Yes" or "No." If it is not acceptable, provide the reason why.

a. *Sunflower.html*	Yes	No
b. *index.doc*	Yes	No
c. *cooking home page.html*	Yes	No
d. *Song_Lyrics.html*	Yes	No
e. *games/rubix.html*	Yes	No
f. *%whatever.html*	Yes	No

4. All of the following markup examples are incorrect. Describe what is wrong with each one, and then write it correctly.

   a. `<img "birthday.jpg">`

   b. `<i>Congratulations!<i>`

   c. `<a href="file.html">linked text</a href="file.html">`

   d. `<p>This is a new paragraph<\p>`

5. How would you mark up this comment in an HTML document so that it doesn't display in the browser window?

```
product list begins here
```

Element Review: Document Structure

This chapter introduced the elements that establish the structure of the document. The remaining elements introduced in the exercises will be treated in more depth in the following chapters.

Element	Description
`body`	Identifies the body of the document that holds the content
`head`	Identifies the head of the document that contains information about the document
`html`	The root element that contains all the other elements
`meta`	Provides information about the document
`title`	Gives the page a title

CHAPTER 10

WHAT'S UP, HTML5?

We've been using HTML5 elements in the past several chapters, but there is a lot more to the HTML5 specification than new markup possibilities (although that is an important part). HTML5 is actually a bundle of new methods for accomplishing tasks that previously required special programming or proprietary plug-in technology such as Flash or Silverlight. It offers a standardized, open source way to put audio, video, and interactive elements on the page as well as the ability to store data locally, work offline, take advantage of location information, and more. With HTML5 for common tasks, developers can rely on built-in browser capabilities and not need to reinvent the wheel for every application.

HTML5 offers so many promising possibilities, in fact, that it has become something of a buzzword with connotations far beyond the spec itself. When marketers and journalists use the term "HTML5," they are sometimes referring to CSS3 techniques or any new web technology that isn't Flash. In this chapter you'll learn what is actually included in the spec, and you can join the rest of us in being slightly irked when the HTML5 label is applied incorrectly. The important thing, however, is that mainstream awareness of web standards is certainly a win and makes our job easier when communicating with clients.

Of course, with any spec in development, browser support is uneven at best. There are some features that can be used right away and some that aren't quite ready for prime time. But this time around, instead of waiting for the entire spec to be "done," browsers are implementing one feature at a time, and developers are encouraged to begin using them (see the Tracking Browser Support sidebar). I should also mention that the HTML5 spec is evolving rapidly and parts are likely to have changed by the time you are reading this. I'll do my best to give you a good overview, and you can decide which features to research and follow on your own.

Much of what's new in HTML5 requires advanced web development skills, so it is unlikely you'll use them right away (if ever), but as always, I think it is beneficial to everyone to have a basic familiarity with what can be done.

IN THIS CHAPTER

And "basic familiarity" is what I'm aiming at with this chapter. For more in-depth discussions of HTML5 features, I recommend the following books:

- *HTML5, Up and Running* by Mark Pilgrim (O'Reilly Media and Google Press)
- *Introducing HTML5* by Bruce Lawson and Remy Sharp (New Riders)

I feel it's only fair to warn you that this chapter is the cod liver oil of this book. Not pleasant to get down, but good for you. An understanding of the big picture and the context of why we do things the way we do is something any budding web designer should have.

Tracking Browser Support

There are several nice resources out there to help you know which HTML5 features are ready to use. Most show support for CSS properties and selectors as well.

- When Can I Use… (*caniuse.com*)
- HTML5 Please (*html5please.com*)
- "Comparison of Layout Engines (HTML5)" on Wikipedia (*en.wikipedia.org/wiki/Comparison_of_layout_engines_(HTML_5)*)

A Funny Thing Happened on the Way to XHTML 2

Understanding where we've been provides useful context for where we are going, so let's kick this off with a quick history lesson. We'll start at the very beginning.

A "don't blink or you'll miss it" history of HTML

The story of HTML, from Tim Berners-Lee's initial draft in 1991 to the HTML5 standard in development today, is both fascinating and tumultuous. Early versions of HTML (HTML+ in 1994 and HTML 2.0 in 1995) built on Tim's early work with the intent of making it a viable publishing option. But when the World Wide Web (as it was adorably called back in the day) took the world by storm, browser developers, most notably Mosaic Netscape and later Microsoft Internet Explorer, didn't wait for any stinkin' standards. They gave the people what they wanted by creating a slew of browser-specific elements for improving the look of pages on their respective browsers. This divisive one-upping is what has come to be known as the Browser Wars. As a result, it became common in the late 1990s to create two separate versions of a site that targeted each of the Big Two browsers.

In 1996, the newly formed W3C put a stake in the ground and released its first Recommendation: HTML 3.2. It is a snapshot of all the HTML elements in common use at the time, and includes many presentational extensions to HTML that were the result of the Netscape/IE feud and the lack of a style sheet alternative. HTML 4.0 (1998) and HTML 4.01 (the slight revision that superseded it in 1999) aimed to get HTML back on track by emphasizing the separation of structure and presentation and improving accessibility. All matters of presentation were handed over to the newly minted Cascading Style Sheets standard that was gaining support.

NOTE

For a detailed history of the beginnings of the World Wide Web and HTML, read David Raggett's account from his book Raggett on HTML4 (Addison-Wesley, 1998), available on the W3C site (www.w3.org/People/Raggett/book4/ch02.html).

Enter XHTML

Around the same time that HTML 4.01 was in development, folks at the W3C became aware that one limited markup language wasn't going to cut it for describing all the sorts of information (chemical notation, mathematical equations, multimedia presentations, financial information, and so on) that might be shared over the Web. Their solution: XML (eXtensible Markup Language), a metalanguage for creating markup languages. XML was a simplification of SGML (Standardized Generalized Markup Language), the big kahuna of metalanguages that Tim Berners-Lee used to create his original HTML application. But SGML itself proved to be more complex than the Web required.

The W3C had a vision of an XML-based Web with many specialized markup languages working together—even within a single document. Of course, to pull that off, everyone would have to mark up documents very carefully, strictly abiding by XML syntax, to rule out potential confusion.

Their first step was to rewrite HTML according to the rules of XML so that it could play well with others. The result is XHTML (eXtensible HTML). The first version, XHTML 1.0, is nearly identical to HTML 4.01, sharing the same elements and attributes, but with stricter requirements for how markup must be done (see the XHTML Markup Requirements sidebar).

HTML 4.01, along with XHTML 1.0, its stricter XML-based sibling, became the cornerstone of the web standards movement (see the sidebar The Web Standards Project). They are still the most thoroughly and consistently supported standards as of this writing (although HTML5 is quickly gaining steam).

But the W3C didn't stop there. With a vision of an XML-based Web in mind, they began work on XHTML 2.0, an even bolder attempt to make things work "right" than HTML 4.01 had been. The problem was that it was not backward-compatible with old standards and browser behavior. The writing and approval process dragged on for years with no browser implementation. Without browser implementation, XHTML 2.0 was stuck.

XHTML Markup Requirements

- Element and attribute names must be lowercase. In HTML, element and attribute names are not case-sensitive.
- All elements must be closed (terminated). Empty elements are closed by adding a slash before the closing bracket (for example, `<br/>`).
- Attribute values must be in quotation marks. Single or double quotation marks are acceptable as long as they are used consistently. Furthermore, there should be no extra whitespace (character spaces or line returns) before or after the attribute value inside the quotation marks.
- All attributes must have explicit attribute values. XML (and therefore XHTML) does not support attribute minimization, the SGML practice in which certain attributes can be reduced to just the attribute value. So, while in HTML you can write `checked` to indicate that a form button be checked when the form loads, in XHTML you need to explicitly write out `checked="checked"`.
- Proper nesting of elements is strictly enforced. Some elements have new nesting restrictions.
- Special characters must always be represented by character entities (e.g., `&amp;` for the & symbol).
- Use `id` instead of `name` as an identifier.
- Scripts must be contained in a CDATA section so they will be treated as simple text characters and not parsed as XML markup. Here is an example of the syntax:

```
<script type="type/javascript">
  // <![CDATA[
  ... JavaScript goes here...
  // ]]>
</script>
```

The Web Standards Project

In 1998, at the height of the browser wars, a grassroots coalition called the Web Standards Project (WaSP for short) began to put pressure on browser creators (primarily Netscape and Microsoft at the time) to start sticking to the open standards as documented by the W3C. Not stopping there, they educated the web developer community on the many benefits of developing with standards. Their efforts revolutionized the way sites are created and supported. Now browsers (even Microsoft) brag of standards support while continuing to innovate. You can read their mission statement, history, and current efforts on the WaSP site (*webstandards.org*).

Hello HTML5!

HTML5 aims to make HTML more useful for creating web applications.

Meanwhile...

In 2004, members of Apple, Mozilla, and Opera formed the Web Hypertext Application Technology Working Group (WHATWG, *whatwg.org*), separate from the W3C. The goal of the WHATWG was to further the development of HTML to meet new demands in a way that was consistent with real-world authoring practices and browser behavior (in contrast to the start-from-scratch ideal that XHTML 2.0 described). Their initial documents, Web Applications 1.0 and Web Forms 1.0, were rolled together into HTML5, which is still in development under the guidance of an editor, Ian Hickson (currently of Google).

The W3C eventually established its own HTML5 Working Group (also led by Hickson) based on the work done by the WHATWG. As of this writing, work on the HTML5 specification is happening in both organizations in tandem, sometimes with conflicting results. It is not yet a formal Recommendation as of this writing, but that isn't stopping browsers from implementing it a little at a time.

NOTE

The WHATWG maintains what it calls the HTML "Living Standard" (meaning they aren't giving it a version number) at www.whatwg.org. It is nearly identical to HTML5, but it includes a few extra elements and attributes that the W3C isn't quite ready to adopt, and it has a slightly different lineup of APIs.

And XHTML 2.0? At the end of 2009, the W3C officially put it out of its misery, pulling the plug on the working group and putting its resources and efforts into HTML5.

So that's how we got here, and it's a whole lot of prelude to the meat of this chapter, which of course is the new features that HTML5 offers. I also encourage you to read the sidebar HTML5 Fun Facts for more juicy information on the specification itself. In this section, I'll introduce what's new in HTML5, including:

- A new DOCTYPE
- New elements and attributes
- Obsolete 4.01 elements
- APIs

HTML5 Fun Facts

HTML5 both builds on previous versions of HTML and introduces some significant departures. Here are some interesting tidbits about the HTML5 specification itself.

- HTML5 is based on HTML 4.01 Strict, the version of HTML that did not include any presentation-based or other deprecated elements and attributes. That means the vast majority of HTML5 is made up of the same elements we've been using for years, and browsers know what to do with them.
- HTML5 does not use a DTD (Document Type Definition), which is a document that defines all of the elements and attributes in a markup language. It is the way you document a language in SGML, and if you'll remember, HTML was originally crafted according to the rules of SGML. HTML 4.01 was defined by three separate DTDs: Transitional (including legacy elements that were marked as "deprecated," or soon to be obsolete), Strict (deprecated features stripped out, as noted earlier), and Frameset (for documents broken into individually scrolling frames, a technique that is now considered obsolete).
- HTML5 is the first HTML specification that includes detailed instructions for how browsers should handle malformed and legacy markup. It bases the instructions on legacy browser behavior, but for once, there is a standard protocol for browser makers to follow when browsers encounter incorrect or non-standard markup.
- HTML5 can also be written according to the stricter syntax of XML (called the XML serialization of HTML5). Some developers have come to prefer the tidiness of well-formed XHTML (lowercase element names, quoted attribute values, closing all elements, and so on), so that way of writing is still an option, although not required. In edge cases, an HTML5 document may be required to be served as XML in order to work with other XML applications, in which case it can use the XML syntax and be ready to go.
- In addition to markup, HTML5 defines a number of APIs (Application Programming Interface). APIs make it easier to communicate with web-based applications. They also move some common processes (such as audio and video players) into native browser functionality.

In the Markup Department

We'll start with a look at the markup aspects of HTML5, and then we'll move on to the APIs.

A minimal DOCTYPE

As we saw in Chapter 4, HTML documents should begin with a Document Type Declaration (DOCTYPE declaration) that identifies which version of HTML the document follows. The HTML5 declaration is short and sweet:

```
<! DOCTYPE html>
```

Compare that to a declaration for a Strict HTML 4.01 document:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/HTML4.01/strict.dtd">
```

Why so complicated? In HTML 4.01 and XHTML 1.0 and 1.1, the declaration must point to the public DTD (Document Type Definition), a document that defines all of the elements in a markup language as well as the rules for using them. HTML 4.01 was defined by three separate DTDs: Transitional (including legacy elements such as font and attributes such as **align** that were marked as "deprecated," or soon to be obsolete), Strict (deprecated features stripped out), and Frameset (for documents broken into individually scrolling frames, a technique that is now considered obsolete). HTML5 does not have a DTD, which is why we have the simple DOCTYPE declaration.

DTDs are a remnant of SGML and proved to be less helpful on the Web than originally thought, so the authors of HTML5 simply didn't use one.

Validators—software that checks that all the markup in a document is correct (see note)—use the DOCTYPE declaration to make sure the document abides by the rules of the specification it claims to follow. The sidebar HTML DOCTYPES lists all declarations in common use, should you need to write documents in HTML 4.01 or XHTML 1.0.

NOTE

To check whether your HTML document is valid, use the online validator at the W3C (validator.w3.org). An HTML5-specific validator is also available at html5.validator.nu. There is also a validator built into Adobe Dreamweaver that allows you to check your document against various specs as you work.

HTML DOCTYPES

The following lists all of the DOCTYPE declarations in common use.

HTML5

```
<!DOCTYPE html>
```

HTML 4.01 Transitional

The Transitional DTD includes deprecated elements and attributes:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/HTML4.01/loose.dtd">
```

HTML 4.01 Strict

The Strict DTD omits all deprecated elements and attributes:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/HTML4.01/strict.dtd">
```

HTML 4.01 Frameset

If your document contains frames—that is, it uses **`frameset`** instead of **`body`** for its content—then identify the Frameset DTD:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/HTML4.01/frameset.dtd">
```

XHTML 1.0 Strict

The same as HTML 4.01 Strict, but reformulated according to the syntax rules of XML:

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

XHTML 1.0 Transitional

The same as HTML 4.01 Transitional, but reformulated according to the syntax rules of XML:

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

XHTML 1.0 Frameset

The same as HTML 4.01 Frameset, but reformulated according to the syntax rules of XML:

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

Elements and attributes

HTML5 introduced a number of new elements. You'll find them sprinkled throughout this book, but Table 10-1 lists them all in one place.

NOTE

For a detailed list of all the ways HTML5 differs from HTML 4.01, see the W3C official document at www.w3.org/TR/html5-diff/.

Table 10-1. New elements in HTML5

article	datalist	header	output	source
aside	details	hgroup	progress	summary
audio	embed	keygen	rp	time
bdi	figcaption	mark	rt	track
canvas	figure	meter	ruby	video
command	footer	nav	section	wbr

New form input types

We covered the new form input control types in Chapter 9, but here they are at a glance: **color**, **date**, **datetime**, **datetime-local**, **email**, **month**, **number**, **range**, **search**, **tel**, **time**, **url**, and **week**.

New global attributes

Global attributes are attributes that can be applied to any element. The number of global attributes was expanded in HTML5, and many of them are brand new (as noted in Table 10-2). The W3C is still adding and removing attributes as of this writing, so it's worth checking in with the spec for the latest (*dev.w3.org/html5/spec/global-attributes.html#global-attributes*).

Table 10-2. Global attributes in HTML5

Attribute	Values	Description
accesskey	Single text character	Assigns an access key (shortcut key command) to the link. Access keys are also used for form fields. Users may access the element by hitting Alt-<*key*> (PC) or Ctrl-<*key*> (Mac).
aria-*	One of the standardized state or property keywords in WAI-ARIA (*www.w3.org/TR/wai-aria/states_and_properties*)	WAI-ARIA (Accessibile Rich Internet Applications) defines a way to make web content and applications more accessible to users with assistive devices. HTML5 allows any of the ARIA properties and roles to be added to elements. For example, a **div** used for a pop-up menu could include the attribute **aria-haspopup** to make that property clear to a user without a visual browser. See also the related **role** global attribute.
class	Text string	Assigns one or more classification names to the element.
contenteditable	**true** | **false**	NEW IN HTML5 Indicates the user can edit the element. This attribute is already well supported in current browser versions.

Table 10-2. Global attributes in HTML5

Attribute	Values	Description
contextmenu	**id** of the **menu** element	NEW IN HTML5 Specifies a context menu that applies to the element. The context menu must be requested by the user, for example, by a right-click.
data-*	Text string or numerical data	NEW IN HTML5 Enables authors to create custom data-related attributes (the "*" is a symbol that means "anything"), for example, **data-length**, **data-duration**, **data-speed**, etc. so that the data can be used by a custom application or scripts.
dir	**ltr** | **rtl**	Specifies the direction of the element ("left to right" or "right to left").
draggable	**true** | **false**	NEW IN HTML5 A **true** value indicates the element is draggable, meaning it can be moved my clicking and holding on it, then moving it to a new position in the window.
dropzone	**copy** | **link** | **move** | **s:text/plain** | **f:***file-type* (for example, **f:image/jpg**)	NEW IN HTML5 Indicates the element can accept dragged and dropped text or file data. The values are a space-separated list that includes what type of data it accepts (**s:text/plain** for text strings; **f:***file-type* for file types) and a keyword that indicates what to do with the dropped content: **copy** results in a copy of the dragged data; **move** moves it to the new location; and **link** results in a link to the original data.
hidden	No value for HTML documents In XHTML, set a value **hidden="hidden"**	NEW IN HTML5 Prevents the element and its descendants from being rendered in the user agent (browser). Any scripts or form controls in hidden sections will still execute, but they will not be presented to the user.
id	Text string (may not begin with an number)	Assigns a unique identifying name to the element.
lang	Two-letter language code (see *www.loc.gov/standards/iso639-2/php/code_list.php*)	Specifies the language for the element by its language code.
role	One of the standard role keywords in WAI-ARIA (see *www.w3.org/TR/wai-aria/roles*)	NEW IN HTML5 Assigns one of the standardized WAI-ARIA roles to an element to make its purpose clearer to users with disabilities. For example, a **div** with contents that will display as a pop-up menu on visual browsers could be marked with **role="menu"** for clarity on screen readers.
spellcheck	**true** | **false**	NEW IN HTML5 Indicates the element is to have its spelling and grammar checked.
style	Semicolon-separated list of style rules (**property: value** pairs)	Associates style information with an element. For example: `<h1 style="color: red; border: 1px solid">Heading</h1>`
tabindex	Number	Specifies the position of the current element in the tabbing order for the current document. The value must be between 0 and 32,767. It is used for tabbing through links on a page or fields in a form and is useful for assistive browsing devices. A value of –1 is allowable to remove elements from the tabbing flow and make them focusable only by JavaScript.
title	Text string	Provides a title or advisory information about the element, typically displayed as a tooltip.

Obsolete HTML 4.01 Markup

HTML5 also declared a number of elements in HTML 4.01 to be "obsolete" because they are presentational, antiquated, or poorly supported (Table 10-3). If you use them, browsers will support them, but I strongly recommend leaving them in the dust.

Table 10-3. HTML 4 elements that are now obsolete in HTML5

acronym	dir	noframes
applet	font	strike
basefont	frame	tt
big	frameset	
center	isindex	

Are you still with me? I know, this stuff gets pretty dry. That's why I've included Figure 10-1. It has nothing at all to do with HTML5, but I thought we could all use a little breather before taking on APIs.

Figure 10-1. This adorable baby red panda has nothing to do with HTML5. (Photo by Tara Menne)

Meet the APIs

HTML specifications prior to HTML5 included only documentation of the elements, attributes, and values permitted in the language. That's fine for simple text documents, but the creators of HTML5 had their minds set on

making it easier to create web-based applications that require scripting and programming. For that reason, HTML5 also defines a number of new APIs for making it easier to communicate with an application.

An API (Application Programming Interface) is a documented set of commands, data names, and so on, that lets one software application communicate with another. For example, the developers of Twitter documented the names of each data type (users, tweets, timestamps, and so on) and the methods for accessing them in an API document (*dev.twitter.com/docs*) that lets other developers include Twitter feeds and elements in their programs. That is why there are so many Twitter programs and widgets available. Amazon.com also opens up its product data via an API. In fact, publishers of all ilks are recognizing the power of having their content available via an API. You could say that APIs are hot right now.

But let's bring it back to HTML5, which includes APIs for tasks that traditionally required proprietary plug-ins (like Flash) or custom programming. The idea is that if browsers offer those features natively—with standardized sets of hooks for accessing them—developers can do all sorts of nifty things and count on it working in all browsers, just as we count on the ability to embed an image on a page today. Of course, we have a way to go before there is ubiquitous support of these cutting-edge features, but we're getting there steadily. Some APIs have a markup component, such as embedding multimedia with the new HTML5 **`video`** and **`audio`** elements. Others happen entirely behind the scenes with JavaScript or server-side components, such as creating web applications that work even when there is no Internet connection (Offline Web Application API).

NOTE

For a list of all the APIs, see the article "HTML Landscape Overview" by Erik Wilde (dret.typepad.com/dretblog/html5-api-overview.html). The W3C lists all the documents they maintain, many of which are APIs, at www.w3.org/TR/tr-title-all.

The W3C and WHATWG are working on *lots and lots* of APIs for use with web applications, all in varying stages of completion and implementation. Most have their own specifications, separate from the HTML5 spec itself, but they are generally included under the wide HTML5 umbrella that covers web-based applications. HTML5 includes specifications for these APIs:

Media Player API

For controlling audio and video players embedded on a web page, used with the new **`video`** and **`audio`** elements. We will take a closer look at audio and video later in this chapter.

Session History API

Exposes the browser history for better control over the Back button.

Offline Web Application API

Makes it possible for a web application to work even when there is no Internet connection. It does it by including a manifest document that lists all of the files and resources that should be downloaded into the browser's cache in order for the application to work. When a connection is available, it checks to see whether any of the documents have changed, then updates those documents.

Editing API

Provides a set of commands that could be used to create in-browser text editors, allowing users to insert and delete text, format text as bold, italic, or as a hypertext link, and more. In addition, there is a new `contenteditable` attribute that allows any content element to be editable right on the page.

Drag and Drop API

Adds the ability to drag a text selection or file to a target area on the page or another web page. The `draggable` attribute indicates the element can be selected and dragged. The `dropzone` attribute is used on the target area and defines what type of content it can accept (text or file type) and what to do with it when it gets there (`copy`, `link`, `move`).

The following are just a handful of the APIs in development at the W3C with specifications of their own (outside HTML5):

Canvas API

The `canvas` element adds a dynamic, two-dimensional drawing space to a page. We'll take a look at it at the end of this chapter.

Web Storage API

Allows data to be stored in the browser's cache so that an application can use it later. Traditionally, that has been done with "cookies," but the Web Storage API allows more data to be stored. It also controls whether the data is limited to one session (`sessionStorage`: when the window is closed, the data is cleared) or based on domain (`localStorage`: all open windows pointed to that domain have access to the data).

Geolocation API

Lets users share their geographical location (longitude and latitude) so that it is accessible to scripts in a web application. This allows the app to provide location-aware features such as suggesting a nearby restaurant or finding other users in your area.

Web Workers API

Provides a way to run computationally complicated scripts in the background. This allows the browser to keep the web page interface quick and responsive to user actions while working on processor-intensive scripts at the same time. The Web Workers API is part of the HTML5 spec at the WHATWG, but at the W3C, it's been moved into a separate document.

Web Sockets API

Creates a "socket," which is an open connection between the browser client and the server. This allows information to flow between the client and the server in real time, with no lags for the traditional HTTP requests. It is useful for multiplayer games, chat, or data streams that update constantly, such as sports or stock tickers or social media streams.

NOTE

You can think of a web socket as an ongoing telephone call between the browser and server compared to the walkie-talkie, one-at-a-time style of traditional browser/server communication. (A hat tip to Jen Simmons for this analogy.)

Some APIs have correlating HTML elements, such as the **audio** and **video** elements for embedding media players on a page, and the **canvas** element for adding a dynamic drawing area. In the following sections, we'll take a brief look at how those elements are put to use.

Video and Audio

In the earliest days of the World Wide Web (I know, I was there), it was possible to add a MIDI file to a web page for a little beep-boopy soundtrack (think early video games). It wasn't long before better options came along, including RealMedia and Windows Media, that allowed all sorts of audio and video formats to be embedded in a web page. In the end, Flash became the *de facto* embedded multimedia player thanks in part to its use by YouTube and similar video services.

What all of these technologies have in common is that they require third-party, proprietary plug-ins to be downloaded and installed in order to play the media files. Until recently, browsers did not have built-in capabilities for handling sound or video, so the plug-ins filled in the gap. With the development of the Web as an open standards platform, it seemed like time to make multimedia support part of browsers' out-of-the-box capabilities. Enter the new **audio** and **video** elements and their respective APIs.

Farewell Flash?

Apple's announcement that it would not support Flash on its iOS devices, *ever*, gave HTML5 an enormous push forward and eventually led to Adobe stopping development on its mobile Flash products. Not long after, Microsoft announced that it was discontinuing its Silverlight media player in lieu of HTML5 alternatives. As of this writing, HTML5 is a long way from being able to reproduce the vast features and functionality of Flash, but it's getting there gradually. That means we are likely to see Flash and Silverlight players on the desktop for years to come, but the trajectory away from plug-ins and toward web standards technologies seems clear.

The good news and the bad news

The good news is that the **audio** and **video** elements are well supported in modern browsers, including IE 9+, Safari 3+, Chrome, Opera, and Firefox 3.5+ for the desktop and iOS Safari 4+, Android 2.3+, and Opera Mobile (however, not Opera Mini).

But lest you envision a perfect world where all browsers are supporting audio and video in perfect harmony, I am afraid that it is not that simple. Although they have all lined up on the markup and JavaScript for embedding and controlling media players, unfortunately they have not agreed on which formats to support. Let's take a brief journey through the land of media file formats. If you want to add audio or video to your page, this stuff is important to understand.

How media formats work

When you prepare audio or video content for web delivery, there are two format decisions to make. The first is how the media is encoded (the algorithms used to convert the source to 1s and 0s and how they are compressed). The method used for encoding is called the codec, which is short for "code/decode" or "compress/decompress." There are a bazillion codecs out there (that's an estimate). Some probably sound familiar, like MP3; others might

sound new, such as H.264, Vorbis, Theora, VP8, and AAC. Fortunately, only a few are appropriate for the Web, and we'll review them in a moment.

Second, you need to choose the container format for the media...you can think of it as a ZIP file that holds the compressed media and its metadata together in a package. Usually a container format can hold more than one codec type, and the full story is complicated. Because space is limited in this chapter, I'm going to cut to the chase and introduce the most common container/codec combinations for the Web. If you are going to add video or audio to your site, I encourage you to get more familiar with all of these formats. The books in the For Further Reading: HTML5 Media sidebar are a great first step.

Meet the video formats

For video, the most common options are:

- **Ogg container + Theora video codec + Vorbis audio codec.** This is typically called "Ogg Theora," and the file should have a *.ogv* suffix. All of the codecs and the container in this option are open source and unencumbered by patents or royalty restrictions, which makes them ideal for web distribution, but some say the quality is inferior to other options.
- **MPEG-4 container + H.264 video codec + AAC audio codec.** This combination is generally referred to as "MPEG-4," and it takes the *.mp4* or *.m4v* file suffix. H.264 is a high-quality and flexible video codec, but it is patented and must be licensed for a fee. The royalty requirement has been a deal-breaker for browsers that refuse to support it.
- **WebM container + VP8 video codec + Vorbis audio codec.** "WebM" is the newest container format and uses the *.webm* file extension. It is designed to work with VP8 and Vorbis exclusively, and has the advantage of being open source and royalty-free.

Of course, the problem that I referred to earlier is that browser makers have not agreed on a single format to support. Some go with open source, royalty-free options like Ogg Theora or WebM. Others are sticking with the better quality of H.264 despite the royalty requirements. What that means is that we web developers need to make multiple versions of videos to ensure support across all browsers. Table 10-4 lists which browsers support the various video options.

For Further Reading: HTML5 Media

I recommend these books when you are ready to learn more about HTML5 media:

- *HTML5 Media*, by Shelley Powers (O'Reilly Media)
- *HTML5, Up and Running*, by Mark Pilgrim (O'Reilly Media) includes a helpful section on HTML5 video.
- *The Definitive Guide to HTML5 Video*, by Sylvia Pfeiffer (Apress)

Table 10-4. Video support in current browsers (as of mid-2012)

Format	Type	IE	Chrome	Firefox	Safari	Opera Mobile	Mobile Safari	Android
Ogg Theora	video/ogg	–	5.0+	3.5+	–	10.5+	–	–
MP4/H.264	video/mp4	9.0+	–	–	3.1+	–	3.0+	2.0+
WebM	video/webm	9.0+	6.0+	4.0+	–	11+	–	2.3.3+

Meet the audio formats

The landscape looks similar for audio formats: several to choose from, but no format that is supported by all browsers (Table 10-5).

- **MP3.** The MP3 format is a codec and container in one, with the file extension.*mp3*. It has become ubiquitous as a music download format. The MP3 (short for MPEG-1 Audio Layer 3) is patented and requires license fees paid by hardware and software companies (not media creators).
- **WAV.** The WAV format (*.wav*) is also a codec and container in one.
- **Ogg container + Vorbis audio codec.** This is usually referred to as "Ogg Vorbis" and is served with the *.ogg* or *.oga* file extension.
- **MPEG 4 container + AAC audio codec.** "MPEG4 audio" (*.m4a*) is less common than MP3.
- **WebM container + Vorbis audio codec.** The WebM (*.webm*) format can also contain audio only.

Table 10-5. Audio support in current browsers (as of 2012)

Format	Type	IE	Chrome	Firefox	Safari	Opera Mobile	Mobile Safari	Android
MP3	audio/mpeg	9.0+	5.0+	–	4+	–	3.0+	2.0+
WAV	audio/wav or audio/wave	–	5.0+	3.5+	4+	10.5+	3.0+	2.0+
Ogg Vorbis	audio/ogg	–	5.0+	3.5+	–	10.5+	–	2.0+
MPEG-4/AAC	audio/mp4	9.0+	5.0+	–	4+	–	3.0+	2.0+
WebM	audio/webm	9.0+	6.0+	4.0+	–	11+	–	2.3.3+

Video and Audio Encoding Tools

There are scores of options for editing and encoding video and audio files, so I can't cover them all here, but the following tools are free and get the job done.

Video conversion

- Miro Video Converter (*www.mirovideoconverter.com*) is a free tool that converts any video to H.264, Ogg Theora, or WebM format optimized for mobile devices or the desktop with a simple drag-and-drop interface. It is available for OS X and Windows.
- Handbrake (*handbrake.fr*) is a popular open source tool for getting better control over H.264 settings. It is available for Windows, OS X, and Linux.
- Firefogg (*firefogg.org*) is an extension to Firefox for converting video to the Ogg Theora format. Simply install the Firefogg extension to Firefox 3.5+, then visit the Firefogg site and convert video using their online interface.

Audio conversion

- MP3/WMA/Ogg Converter (*www.freemp3wmaconverter.com*) is a free tool that converts the following audio formats: MP3, WAV, WMA, OGG, AAC, and more. Sorry, Mac users; it is Windows only.
- On the Mac, try Max, an open source audio converter available at *sbooth.org/Max/*. Audacity (*audacity.sourceforge.net/*) also has some basic conversion tools in addition to being a recording tool.

Adding a video to a page

`<video>...</video>`

Adds a video player to the page

NEW IN HTML5

I guess it's about time we got to the markup for adding a video to a web page (this is the HTML part of the book, after all). Let's start with an example that assumes you are designing for an environment where you know exactly what browser your user will be using. When this is the case, you can provide only one video format using the **src** attribute in the video tag (just as you do for an **img**). Figure 10-2 shows a movie with the default player in the Chrome browser. We'll look at the other attributes after the example.

```
<video src="highlight_reel.mp4" width="640" height="480"
poster="highlight_still.jpg" controls autoplay>
</video>
```

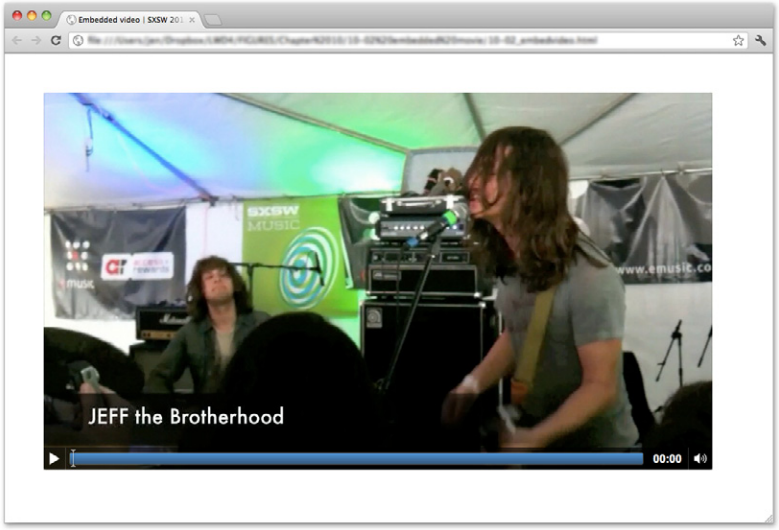


Figure 10-2. An embedded movie using the video element (shown in Chrome on Mac).

There are some juicy attributes in that example worth looking at in detail.

width="*pixel measurement***"**
height="*pixel measurement***"**

Specifies the size of the box the embedded media player takes up on the screen. Generally, it is best to set the dimensions to exactly match the pixel dimensions of the movie. The movie will resize to match the dimensions set here.

poster="*url of image***"**

Provides the location of a still image to use in place of the video before it plays.

controls

Adding the **controls** attribute prompts the browser to display its built-in media controls, generally a play/pause button, a "seeker" that lets you move to a position within the video, and volume controls. It is possible to

WARNING

iOS3 devices will not play a video that includes the **poster** *attribute, so avoid using it if you need to support old iPhones and iPads.*

create your own custom player interface using CSS and JavaScript if you want more consistency across browsers. How to do that is beyond the scope of this chapter, but is explained in the resources listed in the For Further Reading: HTML5 Media sidebar. In many instances, the default controls are just fine.

`autoplay`
: Makes the video start playing automatically once it has downloaded enough of the media file to play through without stopping. In general, use of **`autoplay`** should be avoided in favor of letting the user decide when the video should start.

In addition, the **`video`** (and **`audio`**) element can use the **`loop`** attribute to make the video play again once it has finished (ad infinitum), **`muted`** for playing the video track without the audio, **`mediagroup`** for making a **`video`** element part of a group of related media elements (such as a `video` and a synced sign language translation), and **`preload`** for suggesting to the browser whether the video data should be fetched as soon as the page loads (**`preload="auto"`**) or wait until the user presses the play button (**`preload="none"`**). Setting **`preload="metadata"`** loads information about the media file, but not the media itself. A device can decide how to best handle the **`auto`** setting; for example, a browser in a smartphone may protect a user's data usage by not preloading media, even when it is set to **`auto`**.

object and embed

The **`object`** element is the generic way to embed media such as a movie, Flash movie, applet, even images in a web page. It contains a number of **`param`** (for parameters) elements that provide instructions or resources that the object needs to display. You can also put fallback content inside the **`object`** element that is used if the media is not supported. The attributes and parameters vary by object type and are sometimes specific to the third-party plugin displaying the media.

The **`object`**'s poor cousin, **`embed`**, also embeds media on web pages. It has been a non-standard, but widely supported, element until it was finally made official in HTML5. Some media require the use of **`embed`**, which is often used as a fallback in an **`object`** element to appease all browsers.

You can see an example of the **`object`** and **`param`** elements in the "Video for Everybody" code example on the following page.

Video for all!

But wait a minute! We already know that one video format isn't going to cut it in the real world. At the very least, you need to make two versions of your video: Ogg Theora and MPEG-4 (H.264 video). Some developers prefer WebM instead of Ogg because browser support is nearly as good and the files are smaller. As a fallback for users with browsers that don't support HTML5 video, you can embed a Flash player on the page or use a service like YouTube or Vimeo, in which case you let them handle the conversion, and you just copy the embed code.

In the markup, a series of **`source`** elements inside the **`video`** element point to each video file. Browsers look down the list until they find one they support and download only that version. The Flash fallback gets added with the traditional **`object`** and **`embed`** elements, so if a browser can't make head or tails of **`video`** and **`source`**, chances are high it can play it in Flash. Finally, to ensure accessibility for all, it is highly recommended that you add some simple links to download the videos so they can be played in whatever media player is available, should all of the above fail.

Without further ado, here is one (very thorough) code example for embedding video that should serve all users, including those on mobile devices. You may choose not to provide all these formats, so adapt it accordingly.

The following example is based on the code in Kroc Camen's article "Video for Everybody" (*camendesign.com/code/video_for_everybody*). I highly recommend checking that page for updates, instructions for modifying the code, and many more technical details. We'll look at each part following the example.

```
<video id="yourmovieid" width="640" height="360" poster="yourmovie_
still.jpg" controls preload="auto">
    <source src="yourmovie-baseline.mp4" type='video/mp4;
codecs="avc1.42E01E, mp4a.40.2"'>
    <source src="yourmovie.webm" type='video/webm; codecs="VP8,
vorbis"'>
    <source src="yourmovie.ogv" type='video/ogg; codecs="theora,
vorbis"'>
<!--Flash fallback -->
    <object width="640" height="360" type="application/x-shockwave-
flash" data="your_flash_player.swf">
        <param name="movie" value="your_flash_player.swf">
        <param name="flashvars" value="controlbar=over&amp;image=poster.
jpg&amp;file=yourmovie-main.mp4">
        <img src="poster.jpg" width="640" height="360" alt=""
        title="No video playback capabilities, please download the video
below">
    </object>
</video>
<p>Download the Highlights Reel:</p>
<ul>
    <li><a href="yourmovie.mp4">MPEG-4 format</a></li>
    <li><a href="yourmovie.ogv">Ogg Theora format</a></li>
</ul>
```

NOTE

If you look carefully, you'll see that single quotation marks (') were used to enclose the long string of values for the type attribute in the source element. That is because the codecs must be enclosed in double quotation marks, so the whole attribute requires a different quotation mark type.

Each **source** element contains the location of the media file (**src**) and information about its file type (**type**). In addition to listing the MIME type of the file container (e.g., **video/ogg**), it is helpful to also list the codecs that were used (see the note). This is especially important for MPEG-4 video because the H.264 codec has a number of different profiles, such as baseline (used by mobile devices), main (used by desktop Safari and IE9+), extended, and high (these two are generally not used for web video). Each profile has its own profile ID, as you see in the first **source** element in the example.

Technically, the order of the **source** elements doesn't matter, but to compensate for a bug on early iPads, it is best to put the baseline MPEG-4 first in the list. iPads running iOS 3 won't find it if it's further down, and it won't hurt any other browsers.

After the **source** elements, an **object** element is used to embed a Flash player that will play the MPEG-4 video for browsers that have the Flash plug-in. There are many Flash players available, but Kroc Camen (of "Video for Everybody" fame) recommends JW Player, which is easy to install (just put a JavaScript *.js* file and the Flash *.swf* file on your server). Download the JW Player and instructions for installing and configuring it at *www.longtailvideo.com/players/jw-flv-player/*. If you use the JW Player, replace *your_flash_player.swf* in the example with **player.swf**.

NOTE

In this example, the MPEG-4 video is provided at "baseline" quality in order to play on iOS 3 devices. If iOS3 is obsolete when you are reading this or does not appear in your traffic data, you can provide the higher-quality "main" profile version instead:

```
<source src="yourmovie-
main.mp4" type='video/mp4;
codecs="avc1.4D401E, mp4a.40.2"'>
```

It is important to note that the Flash fallback is for browsers that do not recognize the **video** element. If a browser does support **video** but simply does not support one of the media file formats, it will *not* display the Flash version. It shows nothing. That's why it is a good idea to have direct links (**a**) to the video options outside the **video** element for maximum accessibility.

Finally, if you want the video to start playing automatically, add the **autoplay** attribute to the **video** element and **autostart=true** to the Flash **param** element like this:

```
<video src="movie.mp4" width="640" height="480" autoplay>

<param name="flashvars" value="autostart=true&amp;controlbar=over&amp;
image=poster.jpg&amp; file=yourmovie-main.mp4">
```

Keep in mind that videos will not play automatically on iOS devices, even if you set it in the code. Apple disables **autoplay** on its mobile devices to prevent unintended data transfer.

WARNING

If your server is not configured to properly report the video type (its MIME type) of your video and audio files, some browsers will not play them. The MIME types for each format are listed in the "Type" column in Tables 10-4 and 10-5. So be sure to notify your server administrator or hosting company's technical help if you intend to serve media files and get the MIME types set up correctly.

Adding audio to a page

`<audio>...</audio>`

Adds an audio file to the page

NEW IN HTML5

If you've wrapped your head around the video markup example, you already know how to add audio to a page. The **audio** element uses the same attributes as the **video** element, with the exception of **width**, **height**, and **poster** (because there is nothing to display). Just like the **video** element, you can provide a stack of audio format options using the **source** element, as shown in the example here.

```
<audio id="soundtrack" controls preload="auto">
  <source src="soundtrack.mp3" type="audio/mp3">
  <source src="soundtrack.ogg" type="audio/ogg">
  <source src="soundtrack.webm" type="audio/webm">
</audio>
<p>Download the Soundtrack song:</p>
<ul>
  <li><a href="soundtrack.mp3">MP3</a></li>
  <li><a href="soundtrack.ogg">Ogg Vorbis</a></li>
</ul>
```

If you want to be evil, you could embed audio in a page, set it to play automatically and then loop, and not provide any controls to stop it like this:

```
<audio src="soundtrack.mp3" autoplay loop></audio>
```

WARNING

Firefox versions 7 and earlier do not support the **loop** *attribute.*

But you would never, *ever* do something like that, right? *Right?!* Of course you wouldn't.

Canvas

Another cool, "Look Ma, no plug-ins!" addition in HTML5 is the **canvas** element and the associated Canvas API. The **canvas** element creates an area on a web page that you can draw on using a set of JavaScript functions for creating lines, shapes, fills, text, animations, and so on. You could use it to

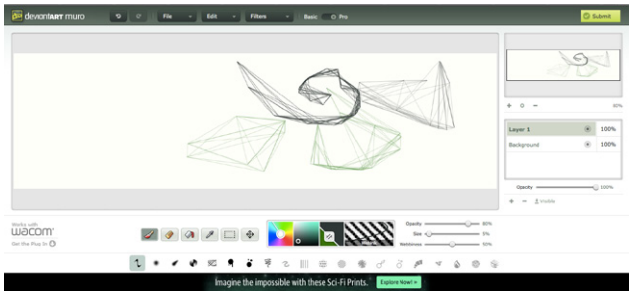
display an illustration, but what gives the **canvas** element so much potential (and has all the web development world so delighted) is that it's all generated with scripting. That means it is dynamic and can draw things on the fly and respond to user input. This makes it a nifty platform for creating animations, games, and even whole applications...all using the native browser behavior and without proprietary plug-ins like Flash.

The good news is that Canvas is supported by every current browser as of this writing, with the exception of Internet Explorer 8 and earlier. Fortunately, the FlashCanvas JavaScript library (*flashcanvas.net*) can add Canvas support to those browsers using the Flash drawing API. So Canvas is definitely ready for prime time.

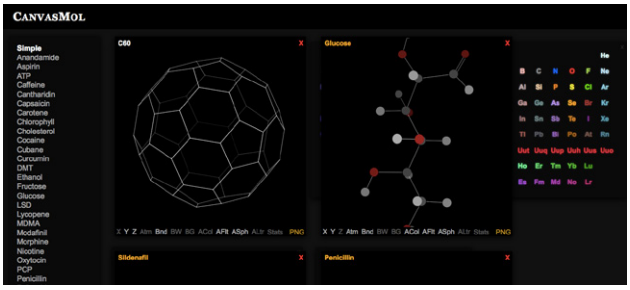
Figure 10-3 shows a few examples of the **canvas** element used to create games, drawing programs, an interactive molecule structure tool, and an asteroid animation. You can find more examples at *Canvasdemos.com*.

ie.microsoft.com/testdrive/Performance/AsteroidBelt/#

www.relfind.com/game/magician.html



muro.deviantart.com



alteredqualia.com/canvasmol/

Figure 10-3. *A few examples of the* canvas *element used for games, animations, and applications.*

Mastering the **canvas** element is more than we can take on here, particularly without any JavaScript experience under our belts, but I will give you a taste of what it is like to draw with JavaScript. That should give you a good idea of how it works, and also a new appreciation for the complexity of some of those examples.

The canvas element

<canvas>...</canvas>

Adds a 2-D dynamic drawing area

NEW IN HTML5

You add a canvas space to the page with the **canvas** element and specify the dimensions with the **width** and **height** attributes. And that's really all there is to the markup. For browsers that don't support the **canvas** element, you can provide some fallback content (a message, image, or whatever seems appropriate) inside the tags.

```
<canvas width="600" height="400" id="my_first_canvas">
  Your browser does not support HTML5 canvas. Try using Chrome,
Firefox, Safari or Internet Explorer 9.
</canvas>
```

The markup just clears a space on which the drawing will happen.

Drawing with JavaScript

The Canvas API includes functions for creating basic shapes (such as **strokeRect()** for drawing a rectangular outline and **beginPath()** for starting a line drawing) and moving things around (such as **rotate()** and **scale()**), plus attributes for applying styles (for example, **lineWidth**, **strokeStyle**, **fillStyle**, and **font**).

The following example was created by my O'Reilly Media colleague Sanders Kleinfeld for his book *HTML5 for Publishers* (O'Reilly). He was kind enough to allow me to use it in this book.

Figure 10-4. *The finished product of our "Hello Canvas" canvas example. See the original at examples.oreilly.com/0636920022473/my_first_canvas/my_first_canvas.html.*

Figure 10-4 shows the simple smiley face we'll be creating with the Canvas API.

And here is the script that created it. Don't worry that you don't know any JavaScript yet. Just skim through the script and pay attention to the little notes. I'll also describe some of the functions in use at the end. I bet you'll get the gist of it just fine.

```
<script type="text/javascript">
window.addEventListener('load', eventWindowLoaded, false);
function eventWindowLoaded() {
    canvasApp();
}

function canvasApp(){
var theCanvas = document.getElementById('my_first_canvas');
var my_canvas = theCanvas.getContext('2d');
my_canvas.strokeRect(0,0,200,225)
    // to start, draw a border around the canvas

    //draw face
my_canvas.beginPath();
my_canvas.arc(100, 100, 75, (Math.PI/180)*0, (Math.PI/180)*360, false);
    // circle dimensions
my_canvas.strokeStyle = "black"; // circle outline is black
my_canvas.lineWidth = 3; // outline is three pixels wide
my_canvas.fillStyle = "yellow"; // fill circle with yellow
my_canvas.stroke(); // draw circle
my_canvas.fill(); // fill in circle
```

```
my_canvas.closePath();

    // now, draw left eye
my_canvas.fillStyle = "black"; // switch to black for the fill
my_canvas.beginPath();
my_canvas.arc(65, 70, 10, (Math.PI/180)*0, (Math.PI/180)*360, false);
    // circle dimensions
my_canvas.stroke(); // draw circle
my_canvas.fill(); // fill in circle
my_canvas.closePath();

    // now, draw right eye
my_canvas.beginPath();
my_canvas.arc(135, 70, 10, (Math.PI/180)*0, (Math.PI/180)*360, false);
    // circle dimensions
my_canvas.stroke(); // draw circle
my_canvas.fill(); // fill in circle
my_canvas.closePath();

    // draw smile
my_canvas.lineWidth = 6; // switch to six pixels wide for outline
my_canvas.beginPath();
my_canvas.arc(99, 120, 35, (Math.PI/180)*0, (Math.PI/180)*-180, false);
    // semicircle dimensions
my_canvas.stroke();
my_canvas.closePath();

    // Smiley Speaks!
my_canvas.fillStyle = "black"; // switch to black for text fill
my_canvas.font      = '20px _sans'; // use 20 pixel sans serif font
my_canvas.fillText  ("Hello Canvas!", 45, 200); // write text
}
</script>
```

Finally, here is a little more information on the Canvas API functions used in the example:

strokeRect(*x1, y1, x2, y2*)
: Draws a rectangular outline from the point (x1, y1) to (x2, y2). By default, the origin of the Canvas (0,0) is the top-left corner, and *x* and *y* coordinates are measured to the right and down.

beginPath()
: Starts a line drawing.

closePath()
: Ends a line drawing that was started with **beginPath()**.

arc(*x, y, arc_radius, angle_radians_beg, angle_radians_end*)
: Draws an arc where (x,y) is the center of the circle, **arc_radius** is the length of the radius of the circle, and **angle_radians_beg** and **_end** indicate the beginning and end of the arc angle.

stroke()
: Draws the line defined by the path. If you don't include this, the path won't appear on the canvas.

`fill()`
: Fills in the path specified with **`beginPath()`** and **`endPath()`**.

`fillText(`*`your_text`*`, `*`x1`*`, `*`y1`*`)`
: Adds text to the canvas starting at the (x,y) coordinate specified.

In addition, the following attributes were used to specify colors and styles:

`lineWidth`
: Width of the border of the path.

`strokeStyle`
: Color of the border.

`fillStyle`
: Color of the fill (interior) of the shape created with the path.

`font`
: The font and size of the text.

Of course, the Canvas API includes many more functions and attributes than we've used here. For a complete list, see the W3C's HTML5 Canvas 2D Context specification at *dev.w3.org/html5/2dcontext/*. A web search will turn up lots of canvas tutorials should you be ready to learn more. In addition, I can recommend these resources:

- The book *HTML5 Canvas* by Steve Fulton and Jeff Fulton (O'Reilly Media)
- Or if watching a video is more your speed, try this tutorial: *Client-side Graphics with HTML5 Canvases: An O'Reilly Breakdown* (*shop.oreilly.com/product/0636920016502.do*)

Final Word

By now you should have a good idea of what's up with HTML5. We've looked at new elements for adding improved semantics to documents. You got a whirlwind tour of the various APIs in development that will move some useful functionality into the native browser behavior. You learned how to use the **`video`** and **`audio`** elements to embed media on the page (plus a primer on media formats). And finally, you got a peek at the **`canvas`** element.

In the next part of this book, CSS for Presentation, you'll learn how to write style sheets that customize the look of the page, including text styles, colors, backgrounds, and even page layout. Goodbye, default browser styles!

Test Yourself

Let's see if you were paying attention. These questions should test whether you got the important highlights of this chapter. Good luck! And as always, the answers are in Appendix A.

1. What is the difference between HTML and XHTML?

2. Using the XHTML Markup Requirements sidebar as a guide, rewrite these HTML elements in XHTML syntax.

   a. `<H1> … </H1>`

   b. `<img src=image.png>`

   c. `<input type="radio" checked>`

   d. `<hr>`

   e. `<title>Sifl & Olly</title>`

   f.
   ```
   <ul>
    <li>popcorn
    <li>butter
    <li>salt
   </ul>
   ```

3. What is a DTD?

4. Name at least three ways that HTML5 is unique as a specification.

5. What is a "global attribute"?

6. Match the API with its function:

Web Workers _____	a. Makes longitude and latitude information available
Editing API _____	b. Holds a line of communication open between the server and browser
Geolocation API _____	c. Makes web apps work even when there is no Internet connection
Web Socket _____	d. Runs processor-intensive scripts in the background
Offline Applications _____	e. Provides a set of commands for copying, pasting, and text formatting

7. Identify each of the following as a container format, video codec, or audio codec.

Ogg	_____
H.264	_____
VP8	_____
Vorbis	_____
WebM	_____
Theora	_____
AAC	_____
MPEG-4	_____

8. List the two Canvas API functions for drawing a rectangle and filling it with red. You don't need to write the whole script.

CASCADING STYLE SHEETS ORIENTATION

IN THIS CHAPTER

The benefits and power of Cascading Style Sheets (CSS)

How HTML markup creates a document structure

Writing CSS style rules

Attaching styles to the HTML document

The big CSS concepts of inheritance, the cascade, specificity, rule order, and the box model

You've heard style sheets mentioned quite a bit already, and now we'll finally put them to work and start giving our pages some much needed style. Cascading Style Sheets (CSS) is the W3C standard for defining the presentation of documents written in HTML, and in fact, any XML language. Presentation, again, refers to the way the document is displayed or delivered to the user, whether on a computer screen, a cell phone display, printed on paper, or read aloud by a screen reader. With style sheets handling the presentation, HTML can handle the business of defining document structure and meaning, as intended.

CSS is a separate language with its own syntax. This chapter covers CSS terminology and fundamental concepts that will help you get your bearings for the upcoming chapters, where you'll learn how to change text and font styles, add colors and backgrounds, and even do basic page layout. By the end of Part III I aim to give you a solid foundation for further reading on your own and lots of practice.

The Benefits of CSS

Not that you need further convincing that style sheets are the way to go, but here is a quick rundown of the benefits of using style sheets.

- **Precise type and layout controls.** You can achieve print-like precision using CSS. There is even a set of properties aimed specifically at the printed page (but we won't be covering them in this book).
- **Less work.** You can change the appearance of an entire site by editing one style sheet.
- **More accessible sites.** When all matters of presentation are handled by CSS, you can mark up your content meaningfully, making it more accessible for non-visual or mobile devices.
- **Reliable browser support.** Every browser in current use supports CSS Level 2 and many cool parts of CSS Level 3. (See the sidebar A Quick History of CSS at the end of this chapter for what is meant by CSS "levels.")

Come to think of it, there really aren't any disadvantages to using style sheets. There are some lingering hassles from browser inconsistencies, but they can either be avoided or worked around if you know where to look for them.

The power of CSS

We're not talking about minor visual tweaks here, like changing the color of headlines or adding text indents. When used to its full potential, CSS is a robust and powerful design tool. My eyes were first opened to the possibilities of using CSS for design by the variety and richness of the designs at CSS Zen Garden (*www.csszengarden.com*).

In the misty days of yore, when developers were still hesitant to give up their table-based layouts for CSS, David Shea's CSS Zen Garden site demonstrated exactly what could be accomplished using CSS alone. David posted an HTML document and invited designers to contribute their own style sheets that gave the document a visual design. Figure 11-1 shows just a few of my favorites. All of these designs use the *exact same* HTML source document.

CSS Zen Dragen
by Matthew Buchanan

Shaolin Yokobue
by Javier Cabrera

By the Pier
by Peter OngKelmscott

Organica Creativa
by Eduardo Cesario

Figure 11-1. These pages from the CSS Zen Garden use the same XHTML source document, but the design is changed using exclusively CSS (used with permission of CSS Zen Garden and the individual designers).

Not only that, it doesn't include a single `img` element (all of the images are used as backgrounds). But look at how different each page looks—and how sophisticated. That's all done with style sheets. It was proof of the power in keeping CSS separate from HTML, and presentation separate from structure.

The CSS Zen Garden is no longer being updated and now is considered a historical document of a turning point in the adoption of web standards. Despite its age, I still find it to be a nice one-stop lesson for demonstrating exactly what CSS can do.

Granted, it takes a lot of practice to be able to create CSS layouts like those shown in Figure 11-1. Killer graphic design skills help too (unfortunately, you won't get those in this book). I'm showing this to you up front because I want you to be aware of the potential of CSS-based design, particularly because the examples in this beginners' book tend to be simple and straightforward. Take your time learning, but keep your eye on the prize.

How Style Sheets Work

It's as easy as 1-2-3!

1. Start with a document that has been marked up in HTML.
2. Write style rules for how you'd like certain elements to look.
3. Attach the style rules to the document. When the browser displays the document, it follows your rules for rendering elements (unless the user has applied some mandatory styles, but we'll get to that later).

OK, so there's a bit more to it than that, of course. Let's give each of these steps a little more consideration.

1. Marking up the document

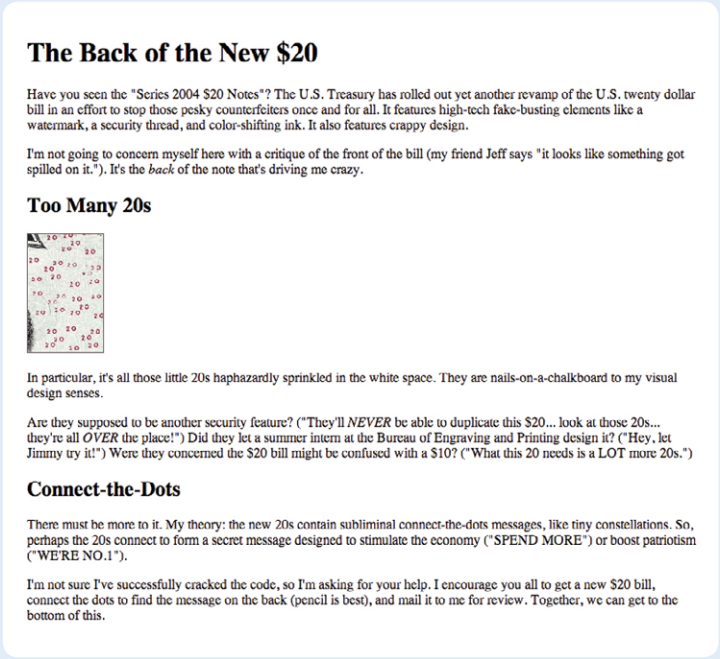
You know a lot about marking up content from the previous chapters. For example, you know that it is important to choose elements that accurately describe the meaning of the content. You've also heard me say that the markup creates the structure of the document, sometimes called the structural layer, upon which the presentation layer can be applied.

In this and the upcoming chapters, you'll see that having an understanding of your document's structure and the relationships between elements is central to your work as a style sheet author.

To get a feel for how simple it is to change the look of a document with style sheets, try your hand at Exercise 11-1. The good news is that I've whipped up a little HTML document for you to play with.

exercise 11-1 | Your first style sheet

In this exercise, we'll add a few simple styles to a short article. The document, *twenties.html*, and its associated image, *twenty_20s.jpg*, are available at *www.learningwebdesign.com/4e/materials/*. First, open the document in a browser to see how it looks by default (it should look something like Figure 11-2). You can also open the document in a text editor and get ready to follow along when this exercise continues in the next section.



The Back of the New $20

Have you seen the "Series 2004 $20 Notes"? The U.S. Treasury has rolled out yet another revamp of the U.S. twenty dollar bill in an effort to stop those pesky counterfeiters once and for all. It features high-tech fake-busting elements like a watermark, a security thread, and color-shifting ink. It also features crappy design.

I'm not going to concern myself here with a critique of the front of the bill (my friend Jeff says "it looks like something got spilled on it."). It's the *back* of the note that's driving me crazy.

Too Many 20s

In particular, it's all those little 20s haphazardly sprinkled in the white space. They are nails-on-a-chalkboard to my visual design senses.

Are they supposed to be another security feature? ("They'll *NEVER* be able to duplicate this $20... look at those 20s... they're all *OVER* the place!") Did they let a summer intern at the Bureau of Engraving and Printing design it? ("Hey, let Jimmy try it!") Were they concerned the $20 bill might be confused with a $10? ("What this 20 needs is a LOT more 20s.")

Connect-the-Dots

There must be more to it. My theory: the new 20s contain subliminal connect-the-dots messages, like tiny constellations. So, perhaps the 20s connect to form a secret message designed to stimulate the economy ("SPEND MORE") or boost patriotism ("WE'RE NO.1").

I'm not sure I've successfully cracked the code, so I'm asking for your help. I encourage you all to get a new $20 bill, connect the dots to find the message on the back (pencil is best), and mail it to me for review. Together, we can get to the bottom of this.

Figure 11-2. This what the article looks like without any style sheet instructions. Although we won't be making it beautiful, you will get a feel for how styles work.

2. Writing the rules

A style sheet is made up of one or more style instructions (called rules or rule sets) that describe how an element or group of elements should be displayed. The first step in learning CSS is to get familiar with the parts of a rule. As you'll see, they're fairly intuitive to follow. Each rule *selects* an element and *declares* how it should look.

The following example contains two rules. The first makes all the **h1** elements in the document green; the second specifies that the paragraphs should be in a small, sans-serif font.

```
h1 { color: green; }
p  { font-size: small; font-family: sans-serif; }
```

In CSS terminology, the two main sections of a rule are the selector that identifies the element or elements to be affected, and the declaration that provides the rendering instructions. The declaration, in turn, is made up of a property (such as **color**) and its value (**green**), separated by a colon and a space. One or more declarations are placed inside curly brackets, as shown in Figure 11-3.

NOTE

Sans-serif fonts do not have a little slab (a serif) at the ends of strokes and tend to look more sleek and modern. We'll talk a lot more about font families in Chapter 12, Formatting Text.

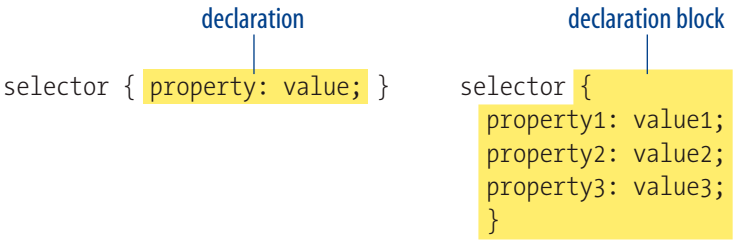


Figure 11-3. The parts of a style sheet rule.

Selectors

In the previous small style sheet example, the **h1** and **p** elements are used as selectors. This is called an element type selector, and it is the most basic type of selector. The properties defined for each rule will apply to every **h1** and **p** element in the document, respectively. In upcoming chapters, I'll introduce you to more sophisticated selectors that you can use to target elements, including ways to select groups of elements and elements that appear in a particular context.

Mastering selectors—that is, choosing the best type of selector and using it strategically—is an important step in becoming a CSS Jedi Master.

Declarations

The declaration is made up of a property/value pair. There can be more than one declaration in a single rule; for example, the rule for the **p** element shown earlier in the code example has both the **font-size** and **font-family** properties. Each declaration must end with a semicolon to keep it separate from the following declaration (see note). If you omit the semicolon, the declaration and the one following it will be ignored. The curly brackets and the declarations they contain are often referred to as the declaration block (Figure 11-3).

Because CSS ignores whitespace and line returns within the declaration block, authors typically write each declaration in the block on its own line, as shown in the following example. This makes it easier to find the properties applied to the selector and to tell when the style rule ends.

```
p  {
  font-size: small;
  font-family: sans-serif;
}
```

Note that nothing has really changed here—there is still one set of curly brackets, semicolons after each declaration, etc. The only difference is the insertion of line returns and some character spaces for alignment.

The heart of style sheets lies in the collection of standard properties that can be applied to selected elements. The complete CSS specification defines dozens of properties for everything from text indents to how table headers

NOTE

Technically, the semicolon is not required after the last declaration in the block, but it is recommended that you get into the habit of always ending declarations with a semicolon. It will make adding declarations to the rule later that much easier.

Providing Measurement Values

When providing measurement values, the unit must immediately follow the number, like this:

```
margin: 2em;
```

Adding a space before the unit will cause the property not to work.

INCORRECT: `margin: 2 em;`

It is acceptable to omit the unit of measurement for zero values:

```
margin: 0;
```

should be read aloud. This book covers the most common and best-supported properties that you can begin using right away.

Values are dependent on the property. Some properties take length measurements, some take color values, and others have a predefined list of keywords. When using a property, it is important to know which values it accepts; however, in many cases, simple common sense will serve you well.

Before we move on, why not get a little practice writing style rules yourself in the continuation of Exercise 11-1?

exercise 11-1 | Your first style sheet (continued)

Open *twenties.html* in a text editor. In the **head** of the document you will find that I have set up a **style** element for you to type the rules into. The **style** element is used to embed a style sheet in an HTML document.

To begin, we'll simply add the small style sheet that we just looked at in this section. Type the following rules into the document, just as you see them here:

```
<style type="text/css">
h1 {
  color: green;
}
p {
  font-size: small;
  font-family: sans-serif;
}
</style>
```

Save the file, and take a look at it in the browser. You should notice some changes (if your browser already uses a sans-serif font, you may only see a size change). If not, go back and check that you included both the opening and closing curly bracket and semicolons. It's easy to accidentally omit these characters, causing the style sheet not to work.

Now we'll change and add to the style sheet to see how easy it is to write rules and see the effects of the changes. Here are a few things to try (remember that you need to save the document after each change in order for the changes to be visible when you reload it in the browser).

- Make the **h1** element "gray" and take a look at it in the browser. Then make it "blue". Finally, make it "red". (We'll run through the complete list of available color names in Chapter 13, Colors and Backgrounds.)
- Add a new rule that makes the **h2** elements red as well.
- Add a 100-pixel left margin to paragraph (**p**) elements using this declaration:

```
margin-left: 100px;
```

Remember that you can add this new declaration to the existing rule for **p** elements.

- Add a 100-pixel left margin to the **h2** headings as well.
- Add a red, 1-pixel border to the bottom of the **h1** element using this declaration:

```
border-bottom: 1px solid red;
```

- Move the image to the right margin, and allow text to flow around it with the **float** property. The shorthand **margin** property shown in this rule adds zero pixels of space on the top and bottom of the image and 12 pixels of space on the left and right of the image (the values are mirrored in a manner explained in Chapter 14, Thinking Inside the Box).

```
img {
  float: right;
  margin: 0 12px;
}
```

When you are done, the document should look something like the one shown in Figure 11-4.

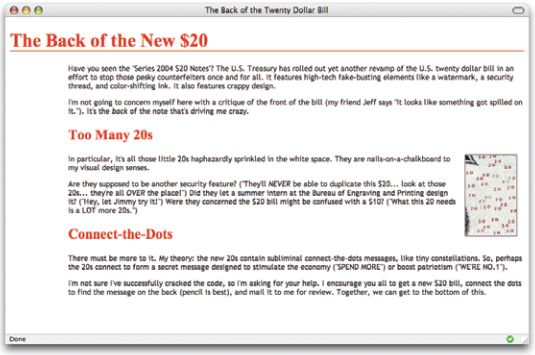


Figure 11-4. *The article after adding the small style sheet from the example. As I said, not beautiful; just different.*

3. Attaching the styles to the document

In the previous exercise, we embedded the style sheet right in the document using the **style** element. That is just one of three ways that style information can be applied to an HTML document. You'll get to try out each of these soon, but it is helpful to have an overview of the methods and terminology up front.

External style sheets. An external style sheet is a separate, text-only document that contains a number of style rules. It must be named with the *.css* suffix. The *.css* document is then linked to or imported into one or more HTML documents (we'll discuss how in Chapter 13). In this way, all the files in a website may share the same style sheet. This is the most powerful and preferred method for attaching style sheets to content.

Embedded style sheets. This is the type of style sheet we worked with in the exercise. It is placed in a document using the **style** element, and its rules apply only to that document. The **style** element must be placed in the **head** of the document. This example also includes a comment (see the Comments in Style Sheets sidebar).

```
<head>
   <title>Required document title here</title>
   <style>
     /* style rules go here */
   </style>
</head>
```

Comments in Style Sheets

Sometimes it is helpful to leave yourself or your collaborators comments in a style sheet. CSS has its own comment syntax, shown here:

```
/* comment goes here */
```

Content between the **/*** and ***/** will be ignored when the style sheet is parsed, which means you can leave comments anywhere in a style sheet, even within a rule.

```
body {
  font-size: small;
 /* font-size:large; */
}
```

One use for comments is to label sections of the style sheet to make things easier to find later, for example:

```
/* Layout styles */
```

or:

```
/* FOOTER STYLES */
```

CSS comments are also useful for temporarily hiding style declarations in the design process. When I am trying out a number of styles, I can quickly switch styles off by enclosing them in **/*** and ***/**, check it in a browser, then remove the comment characters to make the style appear again. It's much faster than retyping the entire thing.

NOTE

In HTML 4.01 and XHTML 1.0/1.1, the `style` *element must contain a* `type` *attribute that identifies the content of the* `style` *element:*

```
<style type="text/css">
```

In HTML5, the `type` *attribute is no longer required.*

The `style` *element may also include the* `media` *attribute used to target specific media such as screen, print, or handheld devices. These are discussed in Chapter 13 as well.*

Inline styles. You can apply properties and values to a single element using the **style** attribute in the element itself, as shown here:

```
<h1 style="color: red">Introduction</h1>
```

To add multiple properties, just separate them with semicolons, like this:

```
<h1 style="color: red; margin-top: 2em">Introduction</h1>
```

Inline styles apply only to the particular element in which they appear. Inline styles should be avoided, unless it is absolutely necessary to override styles from an embedded or external style sheet. Inline styles are problematic in that they intersperse presentation information into the structural markup. They also make it more difficult to make changes because every **style** attribute must be hunted down in the source.

Exercise 11-2 gives you an opportunity to write an inline style and see how it works. We won't be working with inline styles after this point for the reasons listed earlier, so here's your chance.

exercise 11-2 | Applying an inline style

Open the article *twenties.html* in whatever state you last left it in Exercise 11-1. If you worked to the end of the exercise, you will have a rule that makes the **h2** elements red.

Write an inline style that makes the second **h2** gray. We'll do that right in the opening **h2** tag using the **style** attribute, as shown here:

```
<h2 style="color: gray">
Connect-the-Dots</h2>
```

Save the file and open it in a browser. Now the second heading is gray, overriding the red color set in the embedded style sheet. The other **h2** heading is unaffected.

The Big Concepts

There are a few big ideas that you need to get your head around to be comfortable with how Cascading Style Sheets behave. I'm going to introduce you to these concepts now so we don't have to slow down for a lecture once we're rolling through the style properties. Each of these ideas will certainly be revisited and illustrated in more detail in the upcoming chapters.

Inheritance

Are your eyes the same color as your parents'? Did you inherit their hair color? Your unique smile? Well, just as parents pass down traits to their children, styled HTML elements pass down certain style properties to the elements they contain. Notice in Exercise 11-1, when we styled the **p** elements in a small, sans-serif font, the **em** element in the second paragraph became small and sans-serif as well, even though we didn't write a rule for it specifically (Figure 11-5). That is because it inherited the styles from the paragraph it is in.

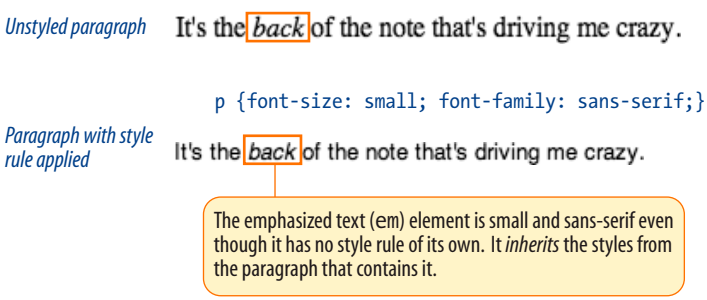


Figure 11-5. The em *element inherits styles that were applied to the paragraph.*

Document structure

This is where an understanding of your document's structure becomes important. As I've noted before, HTML documents have an implicit structure or hierarchy. For example, the sample article we've been playing with has an **html** root element that contains a **head** and a **body**, and the **body** contains heading and paragraph elements. A few of the paragraphs, in turn, contain inline elements such as images (**img**) and emphasized text (**em**). You can visualize the structure as an upside-down tree, branching out from the root, as shown in Figure 11-6.

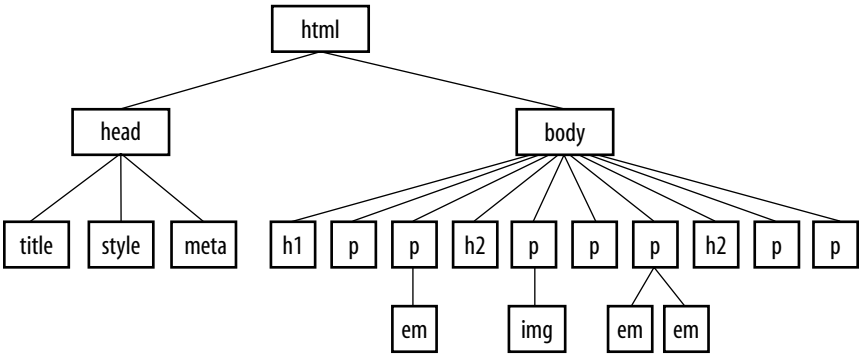


Figure 11-6. The document tree structure of the sample document, twenties.html.

Parents and children

The document tree becomes a family tree when it comes to referring to the relationship between elements. All the elements contained within a given element are said to be its descendants. For example, the **h1**, **h2**, **p**, **em**, and **img** elements in the document in Figure 11-6 are all descendants of the **body** element.

An element that is directly contained within another element (with no intervening hierarchical levels) is said to be the child of that element. Conversely, the containing element is the parent. For example, the **em** element is the child of the **p** element, and the **p** element is its parent.

All of the elements higher than a particular element in the hierarchy are its ancestors. Two elements with the same parent are siblings. We don't refer to "aunts" or "cousins," so the analogy stops there. This may all seem academic, but it will come in handy when writing CSS selectors.

Pass it on

When you write a font-related style rule using the **p** element as a selector, the rule applies to all of the paragraphs in the document as well as the inline text elements they contain. We've seen the evidence of the **em** element inheriting the style properties applied to its parent (**p**) back in Figure 11-5. Figure 11-7 demonstrates what's happening in terms of the document structure diagram. Note that the **img** element is excluded because font-related properties do not apply to images.

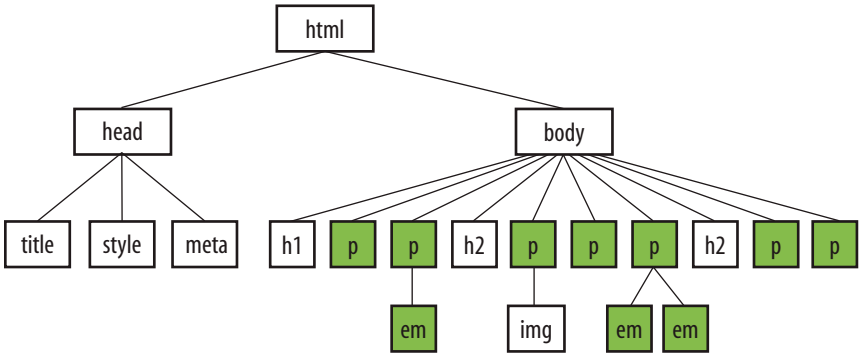


Figure 11-7. Certain properties applied to the **p** *element are inherited by their children.*

CSS TIP

When you learn a new property, it is a good idea to note whether it inherits. Inheritance is noted for every property listing in this book. For the most part, inheritance follows your expectations.

Notice that I've been saying "certain" properties are inherited. It's important to note that some style sheet properties inherit and others do not. In general, properties related to the styling of text—font size, color, style, etc.—are passed down. Properties such as borders, margins, backgrounds, and so on, that affect the boxed area around the element tend not to be passed down. This makes sense when you think about it. For example, if you put a border around a paragraph, you wouldn't want a border around every inline element (such as **em**, **strong**, or **a**) it contains as well.

WARNING

The browser's style sheet may override styles set on the **body**, *so be on the lookout for unexpected styling.*

You can use inheritance to your advantage when writing style sheets. For example, if you want all text elements to be rendered in the Verdana font face, you could write separate style rules for every element in the document and set the **font-face** to Verdana. A *better* way would be to write a single style rule that applies the **font-face** property to the **body** element, and let all the text elements contained in the **body** inherit that style (Figure 11-8).

Any property applied to a specific element will override the inherited values for that property. Going back to the article example, we could specify that the **em** element should appear in a serif font, and that would override the inherited sans-serif setting.

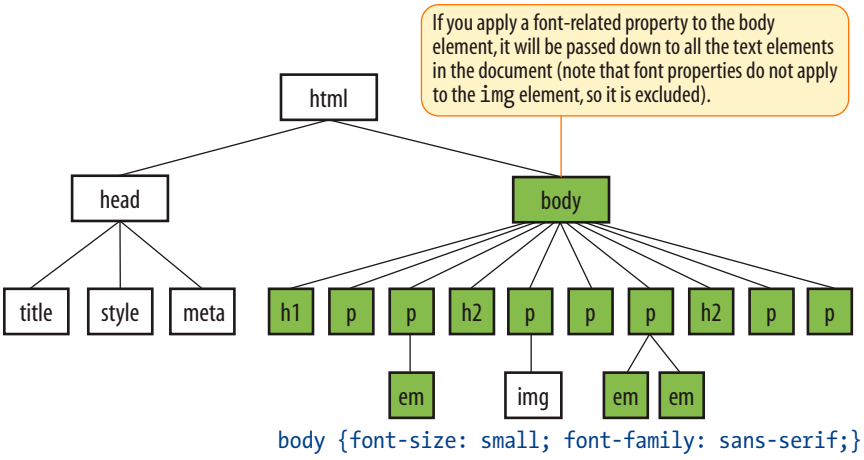


Figure 11-8. All the elements in the document inherit certain properties applied to the **body** *element.*

Conflicting styles: the cascade

Ever wonder why they are called "cascading" style sheets? CSS allows you to apply several style sheets to the same document, which means there are bound to be conflicts. For example, what should the browser do if a document's imported style sheet says that **h1** elements should be red, but its embedded style sheet has a rule that makes **h1**s purple?

The folks who wrote the style sheet specification anticipated this problem and devised a hierarchical system that assigns different weights to the various sources of style information. The cascade refers to what happens when several sources of style information vie for control of the elements on a page: style information is passed down ("cascades" down) until it is overridden by a style command with more weight.

For example, if you don't apply any style information to a web page, it will be rendered according to the browser's internal style sheet (we've been calling this the default rendering; the W3C calls it the user agent style sheet). Individual users can apply their own styles as well (the user style sheet), which overrides the default styles in their browser. However, if the author of the web page has attached a style sheet (the author style sheet), that overrides both the user and the user agent styles. The only exception is if the user has identified a style as "important," in which case that style will trump all (see the Assigning Importance sidebar).

The style sheet source is one hierarchy that determines which style wins. As we've learned, there are three ways to attach style information to the source document, and they have a cascading order as well. Generally speaking, the closer the style sheet is to the content, the more weight it is given. Embedded style sheets that appear right in the document in the **style** element have more weight than external style sheets. In the example that started this section, the **h1** elements would end up purple as specified in the embedded style sheet, not red as specified in the external *.css* file that has less weight. Inline styles have more weight than embedded style sheets because you can't get any closer to the content than a style right in the element's opening tag. That's the effect we witnessed in Exercise 11-2.

To prevent a specific rule from being overridden, you can assign it "importance" with the **!important** indicator, as explained in the Assigning Importance sidebar. The sidebar Style Sheet Hierarchy provides an overview of the cascading order from general to specific.

Specificity

When two rules in a single style sheet conflict, the type of selector is used to determine the winner.

Once the applicable style sheet has been chosen, there may still be conflicts; therefore, the cascade continues at the rule level. When two rules in a single style sheet conflict, the type of selector is used to determine the winner. The more specific the selector, the more weight it is given to override conflicting declarations.

It's a little soon to be discussing specificity because we've only looked at one type of selector (and the least specific type, at that). For now, put the term specificity and the concept of some selectors overriding others on your radar. We will revisit it in Chapter 12 when you have more selector types under your belt.

Assigning Importance

If you want a rule not to be overridden by a subsequent conflicting rule, include the **!important** indicator just after the property value and before the semicolon for that rule. For example, to make paragraph text blue always, use the following rule:

```
p {color: blue !important;}
```

Even if the browser encounters an inline style later in the document (which should override a document-wide style sheet), like this one:

```
<p style="color: red">
```

that paragraph will still be blue because the rule with the **!important** indicator cannot be overridden by other styles in the author's style sheet.

The only way an **!important** rule may be overridden is by a conflicting rule in a reader (user) style sheet that has also been marked **!important**. This is to ensure that special reader requirements, such as large type for the visually impaired, are never overridden.

Based on the previous examples, if the reader's style sheet includes this rule:

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

the text would still be blue because all author styles (even those not marked **!important**) take precedence over the reader's styles. However, if the conflicting reader's style is marked **!important**, like this:

```
p {color: black !important;}
```

the paragraphs will be black and cannot be overridden by any author-provided style.

Style Sheet Hierarchy

Style information can come from various sources, listed here from general to specific. Items lower in the list will override items above them:

- Browser default settings
- User style settings (set in a browser as a "reader style sheet")
- Linked external style sheet (added with the **`link`** element)
- Imported style sheets (added with the **`@import`** function)
- Embedded style sheets (added with the **`style`** element)
- Inline style information (added with the **`style`** attribute in an opening tag)
- Any style rule marked **`!important`** by the author
- Any style rule marked **`!important`** by the reader (user)

Rule order

Finally, if there are conflicts within style rules of identical weight, whichever one comes last in the list "wins." Take these three rules, for example:

```
<style>
  p { color: red; }
  p { color: blue; }
  p { color: green; }
</style>
```

In this scenario, paragraph text will be green because the last rule in the style sheet—that is, the one closest to the content in the document—overrides the earlier ones. The same thing happens when conflicting styles occur within a single declaration stack:

```
<style>
  p { color: red;
      color: blue;
      color: green; }
</style>
```

The resulting color will be green because the last declaration overrides the previous two. It is easy to accidentally override previous declarations within a rule when you get into compound properties, so this is an important behavior to keep in mind.

NOTE

This "last one listed wins" rule applies in other contexts in CSS as well. For example, external style sheets listed later in the source will be given precedence over those listed above them.

The box model

As long as we're talking about "big CSS concepts," it is only appropriate to introduce the cornerstone of the CSS visual formatting system: the box model. The easiest way to think of the box model is that browsers see every element on the page (both block and inline) as being contained in a little rectangular box. You can apply properties such as borders, margins, padding, and backgrounds to these boxes, and even reposition them on the page.

We're going to go into a lot more detail about the box model in Chapter 14, but having a general feel for the box model will benefit you even as we discuss text and backgrounds in the following two chapters.

To see the elements roughly the way the browser sees them, I've written style rules that add borders around every content element in our sample article.

```
h1 { border: 1px solid blue; }
h2 { border: 1px solid blue; }
p { border: 1px solid blue; }
em { border: 1px solid blue; }
img { border: 1px solid blue; }
```

Figure 11-9 shows the results. The borders reveal the shape of each block element box. There are boxes around the inline elements (`em` and `img`) as well. Notice that the block element boxes expand to fill the available width of the browser window, which is the nature of block elements in the normal document flow. Inline boxes encompass just the characters or image they contain.

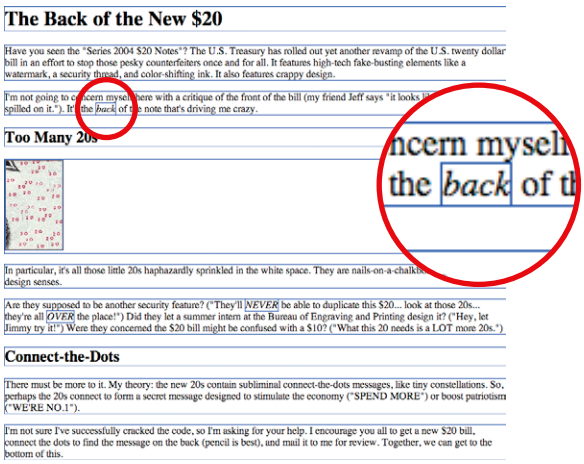


Figure 11-9. Rules around all the elements reveal their element boxes.

Grouped selectors

Hey! This is a good opportunity to show you a handy style rule shortcut. If you ever need to apply the same style property to a number of elements, you can group the selectors into one rule by separating them with commas. This one rule has the same effect as the five rules listed previously. Grouping them makes future edits more efficient and results in a smaller file size.

```
h1, h2, p, em, img { border: 1px solid blue; }
```

Now you have two selector types in your toolbox: a simple element selector and grouped selectors.

Pop Quiz

Can you guess why I didn't just add the **`border`** property to the **`body`** element and let it inherit to all the elements in the grouped selector?

Answer:

Because **`border`** is one of the properties that is not inherited.

A Quick History of CSS

The first official version of CSS (the CSS Level 1 Recommendation, a.k.a CSS1) was officially released in 1996, and included properties for adding font, color, and spacing instructions to page elements. Unfortunately, lack of dependable browser support prevented the widespread adoption of CSS for several years.

CSS Level 2 (CSS2) was released in 1998. It most notably added properties for positioning that allowed CSS to be used for page layout. It also introduced styles for other media types (such as print, handheld, and aural) and more sophisticated methods for selecting elements for styling. CSS Level 2, Revision 1 (CSS2.1) makes some minor adjustments to CSS2 and became a full Recommendation in 2011.

CSS Level 3 (CSS3) is different from prior versions in that it has been divided into many individual modules, each addressing a feature such as animation, multiple column layouts, or borders. While some modules are being standardized, others remain experimental. In that way, browser developers can begin implementing (and we can begin using!) one feature at a time instead of waiting for an entire specification to be "ready." In fact, many developers use enhanced CSS3 features even though they aren't universally supported as long as the fallback is usable and no content is lost. They can be used as "frosting" on an otherwise stable design (or in other words, as an enhancement).

To keep up to date with the various CSS features in the works, see the W3C's CSS Current work page at *www.w3.org/Style/CSS/current-work*.

Moving Forward with CSS

This chapter covered all the fundamentals of Cascading Style Sheets, including rule syntax, ways to apply styles to a document, and the central concepts of inheritance, the cascade, and the box model. Style sheets should no longer be a mystery, and from this point on, we'll merely be building on this foundation by adding properties and selectors to your arsenal as well as expanding on the concepts introduced here.

CSS is a vast topic, well beyond the scope of this book. Bookstores and the Web are loaded with information about style sheets for all skill levels. I've compiled a list of the resources I've found the most useful during my learning process. I've also provided a list of popular tools that assist in writing style sheets.

Books

There is no shortage of good books on CSS out there, but these are the ones that taught me, and I feel good recommending them.

Cascading Style Sheets: The Definitive Guide, by Eric Meyer (O'Reilly Media)

CSS: The Missing Manual, by David Sawyer McFarland (O'Reilly Media)

Handcrafted CSS: More Bulletproof Web Design, by Dan Cederholm (New Riders)

CSS Cookbook: Quick Solutions to Common CSS Problems, by Christopher Schmitt (O'Reilly Media)

Online resources

The sites listed here are good starting points for online exploration of style sheets.

World Wide Web Consortium (*www.w3.org/Style/CSS*). The World Wide Web Consortium oversees the development of web technologies, including CSS.

A List Apart (*www.alistapart.com/topics/code/css/*). This online magazine features some of the best thinking and writing on cutting-edge, standards-based web design. It was founded in 1998 by Jeffrey Zeldman and Brian Platz.

CSS-tricks (*css-tricks.com*). The is the blog of CSS whiz kid Chris Coyier. Chris *loves* CSS and enthusiastically shares his research and tinkering on his site.

CSS tools

Here are a couple of tools that I can personally recommend.

Web Developer extension

Web developers are raving about the Web Developer extension written by Chris Pederick. The extension adds a toolbar to the browser with tools that enable you to analyze and manipulate any page in the window. You can edit the style sheet for the page you are viewing as well as get information about the HTML and graphics. It also validates the CSS, HTML, and accessibility of the page. It is available for Chrome and Firefox/Mozilla browsers. Get it at *chrispederick.com/work/web-developer*. Note that Safari has a similar built-in inspector (go to Develop → Show Web Inspector).

Web authoring programs

Current WYSIWYG authoring programs such as Adobe Dreamweaver and Microsoft Expression Web can be configured to write a style sheet for you automatically as you design the page. The downside is that they are not always written in the most efficient manner (for example, they tend to overuse the `class` attribute to create style rules). Still, they may give you a good head start on the style sheet that you can then edit manually.

Test Yourself

Here are a few questions to test your knowledge of the CSS basics. Answers are provided in Appendix A.

1. Identify the various parts of this style rule:

```
blockquote { line-height: 1.5; }
```

selector: _____ value: _____

property: _____ declaration: _____

2. What color will paragraphs be when this embedded style sheet is applied to a document? Why?

```
<style type="text/css">
   p { color: purple; }
   p { color: green; }
   p { color: gray; }
</style>
```

3. Rewrite each of these CSS examples. Some of them are completely incorrect, and some could just be written more efficiently.

   a.
```
p {font-family: sans-serif;}
p {font-size: 1em;}
p {line-height: 1.2em;}
```

   b.
```
blockquote {
 font-size: 1em
 line-height: 150%
 color: gray }
```

   c.
```
body
 {background-color: black;}
 {color: #666;}
 {margin-left: 12em;}
 {margin-right: 12em;}
```

   d.
```
p {color: white;}
blockquote {color: white;}
li {color: white;}
```

   e.
```
<strong style="red">Act now!</strong>
```

4. Circle all the elements in the diagram that you would expect to appear in red when the following style rule is applied to a document with the structure diagrammed in Figure 11-10. This rule uses a type of selector you haven't seen yet, but common sense should serve you well.

```
div#intro { color: red;}
```

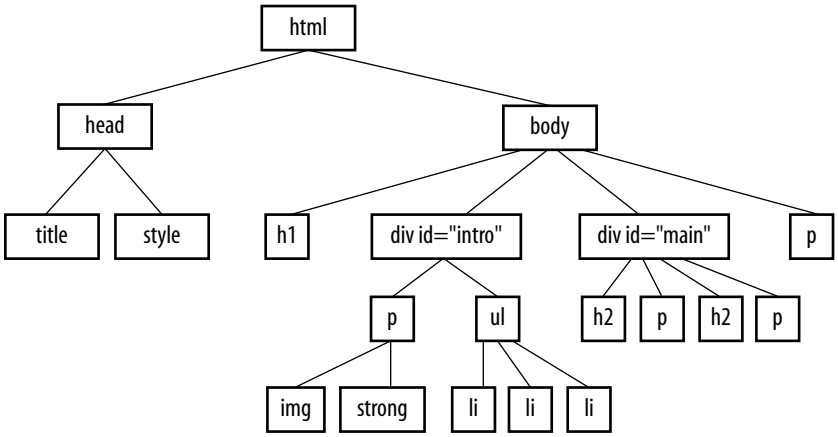


Figure 11-10. The document structure of a sample document.

CHAPTER 19

INTRODUCTION TO JAVASCRIPT

by Mat Marquis

IN THIS CHAPTER

In this chapter, I'm going to introduce you to JavaScript. Now, it's possible you've just recoiled a little bit, and I understand. We're into full-blown "programming language" territory now, and that can be a little intimidating. I promise, it's not so bad!

We'll start by going over what JavaScript is—and what it isn't—and discuss some of the ways it is used. The majority of the chapter is made up of an introduction to JavaScript syntax—variables, functions, operators, loops, stuff like that. Will you be coding by the end of the chapter? Probably not. But you will have a good head start toward understanding what's going on in a script when you see one. I'll finish up with a look at some of the ways you can manipulate the browser window and tie scripts to user actions such as clicking or submitting a form.

What Is JavaScript?

If you've made it this far in the book, you no doubt already know that JavaScript is the programming language that adds interactivity and custom behaviors to our sites. It is a client-side scripting language, which means it runs on the user's machine and not on the server, as other web programming languages such as PHP and Ruby do. That means JavaScript (and the way we use it) is reliant on the browser's capabilities and settings. It may not even be available at all, either because the user has chosen to turn it off or because the device doesn't support it, which good developers keep in mind and plan for. JavaScript is also what is known as a dynamic and loosely typed programming language. Don't sweat this description too much; I'll explain what all that means later.

First, I want to establish that JavaScript is kind of misunderstood.

What it isn't

Right off the bat, the name is pretty confusing. Despite its name, JavaScript has nothing to do with Java. It was created by Brendan Eich at Netscape in 1995 and originally named "LiveScript." But Java was all the rage around that time, so for the sake of marketing, "LiveScript" became "JavaScript." Or just "JS," if you want to sound as cool as one possibly can while talking about JavaScript.

JS also has something of a bad reputation. For a while it was synonymous with all sorts of unscrupulous Internet shenanigans—unwanted redirects, obnoxious pop-up windows, and a host of nebulous "security vulnerabilities," just to name a few. There was a time when JavaScript allowed less reputable developers to do all these things (and worse), but modern browsers have largely caught on to the darker side of JavaScript development and locked it down. We shouldn't fault JavaScript itself for that era, though. As the not-so-old cliché goes: "with great power comes great responsibility." JavaScript has always allowed developers a tremendous amount of control over how pages are rendered and how our browsers behave, and it's up to us to use that control in responsible ways.

What it is

NOTE

JavaScript was standardized in 1996 by the European Computer Manufacturer's Association (ECMA), which is why you sometimes hear it called ECMAScript.

Now we know what JavaScript isn't: it isn't related to Java, and it isn't a mustachioed villain lurking within your browser, wringing its hands and waiting to alert you to "hot singles in your area." Let's talk more about what JavaScript *is*.

JavaScript is a lightweight but incredibly powerful scripting language. We most frequently encounter it through our browsers, but JavaScript has snuck into everything from native applications to PDFs to ebooks. Even web servers themselves can be powered by JavaScript.

As a dynamic programming language, JavaScript doesn't need to be run through any form of compiler that interprets our human-readable code into something the browser can understand. The browser effectively reads the code the same way we do and interprets it on the fly.

JavaScript is also loosely typed. All this means is that we don't necessarily have to tell JavaScript what a variable is. If we're setting a variable to a value of 5, we don't have to programmatically specify that variable as a number. As you may have noted, 5 is already a number, and JavaScript recognizes it as such.

Now, you don't necessarily need to memorize these terms to get started writing JS, mind you—to be honest, I didn't. Even now my eyes gloss over a little as I read them. This is just to introduce you to a few of the terms you'll hear

often while you're learning JavaScript, and they'll start making more and more sense as you go along. This is also to provide you with conversation material for your next cocktail party! "Oh, me? Well, I've been really into loosely typed dynamic scripting languages lately." People will just nod silently at you, which I think means you're doing well conversationally. I don't go to a lot of cocktail parties.

What JavaScript can do

Most commonly we'll encounter JavaScript as a way to add interactivity to a page. Where the "structural" layer of a page is our markup and the "presentational" layer of a page is made up of CSS, the third "behavioral" layer is made up of our JavaScript. All of the elements, attributes, and text on a web page can be accessed by scripts using the DOM (Document Object Model), which we'll be looking at in Chapter 20, Using JavaScript. We can also write scripts that react to user input, altering either the contents of the page, the CSS styles, or the browser's behavior on the fly.

You've likely seen this in action if you've ever attempted to register for a website, entered a username, and immediately received feedback that the username you've entered is already taken by someone else (Figure 19-1). The red border around the text input and the appearance of the "sorry, this username is already in use" message are examples of JavaScript altering the contents of the page, and blocking the form submission is an example of JavaScript altering the browser's default behavior.

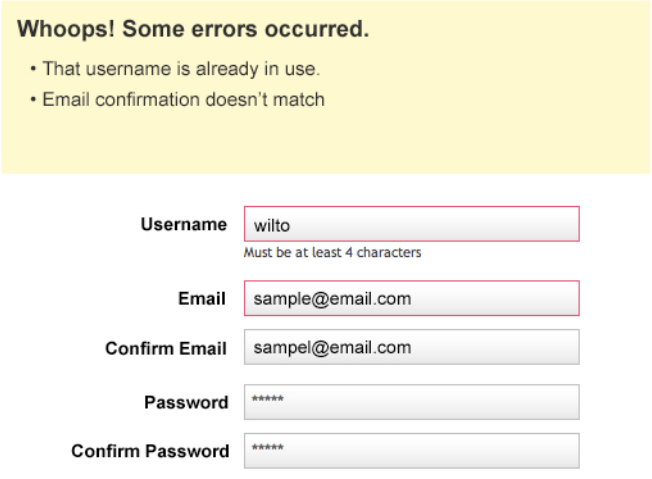


Figure 19-1. JavaScript detects that a username is not available and then inserts a message and alters styles to make the problem apparent.

In short, JavaScript allows you to create highly responsive interfaces that improve the user experience and provide dynamic functionality, without waiting for the server to load up a new page. For example, we can use JavaScript to do any of the following:

- Suggest the complete term a user might be entering in a search box as he types. You can see this in action on Google.com (Figure 19-2).

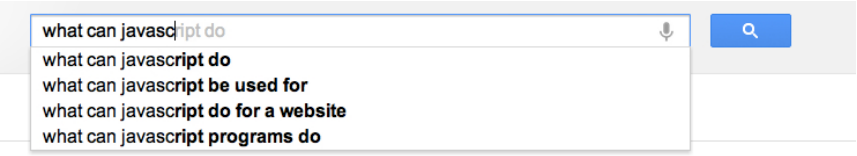


Figure 19-2. Google.com uses JavaScript to automatically complete a search term as it is typed in.

Figure 19-3. JavaScript can be used to reveal and hide portions of content.

- Request content and information from the server and inject it into the current document as needed, without reloading the entire page—this is commonly referred to as "Ajax."
- Show and hide content based on a user clicking on a link or heading, to create a "collapsible" content area (Figure 19-3).
- Test for browsers' individual features and capabilities. For example, one can test for the presence of "touch events," indicating that the user is interacting with the page through a mobile device's browser, and add more touch-friendly styles and interaction methods.
- Fill in gaps where a browser's built-in functionality falls short, or add some of the features found in newer browsers to older browsers. These kinds of scripts are usually called shims or polyfills.
- Load an image or content in a custom styled "lightbox"—isolated on the page using CSS—after a user clicks on a thumbnail version of the image (Figure 19-4).

This list is nowhere near exhaustive!

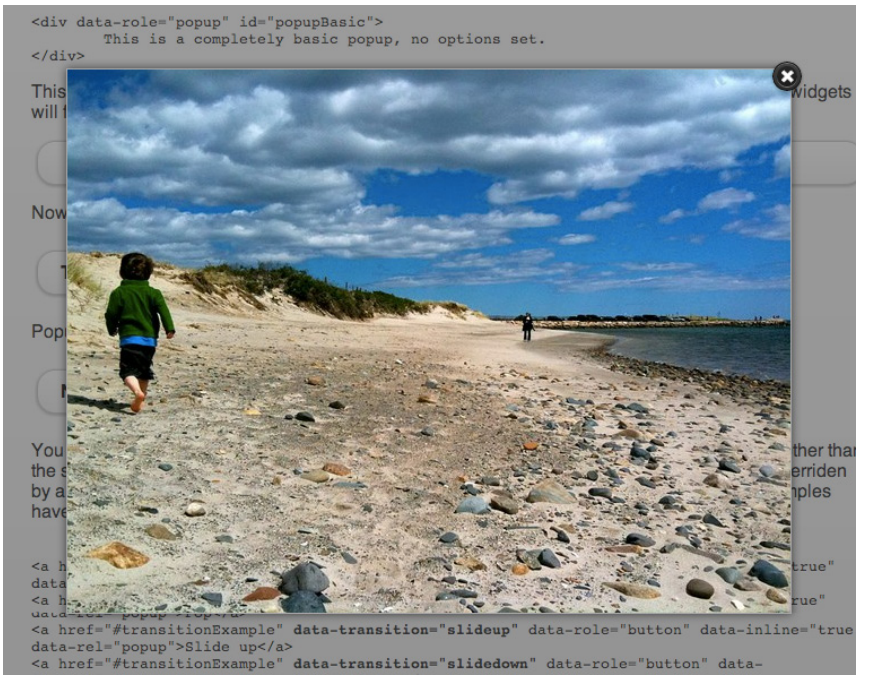


Figure 19-4. JavaScript can be used to load images into a lightbox-style gallery.

Adding JavaScript to a Page

Like CSS, you can embed a script right in a document or keep it in an external file and link it to the page. Both methods use the **script** element.

Embedded script

To embed a script on a page, just add the code as the content of a **script** element:

```
<script>
  … JavaScript code goes here
</script>
```

External scripts

The other method uses the **src** attribute to point to a script file (with a *.js* suffix) by its URL. In this case, the script element has no content.

```
<script src="my_script.js"></script>
```

The advantage to external scripts is that you can apply the same script to multiple pages (the same benefit external style sheets offer). The downside, of course, is that each external script requires an additional HTTP request of the server, which slows down performance.

Script placement

The **script** element go anywhere in the document, but the most common places for scripts are in the **head** of the document and at the very end of the **body**. It is recommended that you don't sprinkle them throughout the document, because they would be difficult to find and maintain.

For most scripts, the end of the document, just before the **</body>** tag, is the preferred placement because the browser will be done parsing the document and its DOM structure. Consequently, that information will be ready and available by the time it gets to the scripts and they can execute faster. In addition, the script download and execution blocks the rendering of the page, so moving the script to the bottom improves the perceived performance. However, in some cases, you might want your script to do something before the body completely loads, so putting it in the **head** will result in better performance.

NOTE

In HTML 4.01 the **script** *tag must include the* **type** *attribute in order to be valid:*

```
<script type="text/
    javascript">…</script>
```

For XHTML documents, you must identify the content of the **script** *element as CDATA the code in the following wrapper:*

```
<script type="text/javascript">
  // <![CDATA[
  …JavaScript code goes here
  // ]]>
</script>
```

The Anatomy of a Script

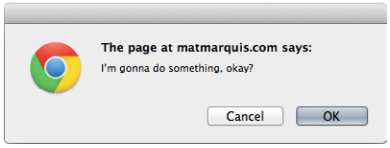
There's a reason why the book *JavaScript: The Definitive Guide* by David Flanagan (O'Reilly) is 1,100 pages long. There's a *lot* to say about JavaScript! In this section, we have only a few pages to make you familiar with the basic building blocks of JavaScript so you can begin to understand scripts when you encounter them. Many developers have taught themselves to program

by finding existing scripts and adapting them for their own needs. After some practice, they are ready to start writing their own from scratch. You may want to learn to write JavaScript yourself as well to round out your web designer skill set. Recognizing the parts of a script is the first step, so that's where we'll start.

Originally, JavaScript's functionality was mostly limited to crude methods of interaction with the user. We could use a few of JavaScript's built-in functions (Figure 19-5) to provide user feedback, such as **`alert()`** to push a notification to a user and **`confirm()`** to ask a user to approve or decline an action. To request the user's input, we were more or less limited to the built-in **`prompt()`** function. Although these methods still have their time and place today, they're jarring, obtrusive, and—in common opinion, at least—fairly obnoxious ways of interacting with users. As JavaScript has evolved over time, we've been afforded much more graceful ways of adding behavior to our pages, creating a more seamless experience for our users.

```
alert("Hi there");
```


```
confirm("I'm gonna do something, okay?");
```




```
prompt("What should I do?");
```

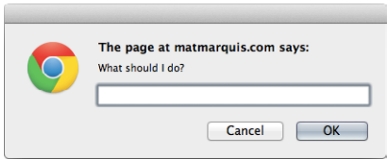


Figure 19-5. Built-in JavaScript functions: **`alert()`** *(top),* **`confirm()`** *(middle), and* **`prompt()`** *(bottom).*

In order to take advantage of these interaction methods, we have to first understand the underlying logic that goes into scripting. These are logic patterns common to all manner of programming languages, although the syntax may vary. To draw a parallel between programming languages and spoken languages, although the vocabulary may vary from one language to another, many grammar patterns are shared by the majority of them.

By the end of this section, you're going to know about variables, arrays, comparison operators, if/else statements, loops, functions, and more. Ready?

The basics

There are a few common syntactical rules that wind their way though all of JavaScript.

JavaScript is case-sensitive.

It is important to know that JavaScript is case-sensitive. A variable named "myVariable", a variable named "myvariable", and a variable named "MYVariable" will be treated as three different objects. Also, whitespace such as tabs and spaces are ignored, unless they're part of a string of text and enclosed in quotes.

Statements

A script is made up of a series of statements. A statement is a command that tells the browser what to do. Here is a simple statement that makes the browser display an alert with the phrase "Thank you."

```
alert("Thank you.");
```

The semicolon at the end of the statement tells JavaScript that it's the end of the command, just as a period ends a sentence. According to the JavaScript standard, a line break will also trigger the end of a command, but it is a best practice to end each statement with a semicolon.

Comments

JavaScript allows you to leave comments that will be ignored at the time the script is executed, so you can leave reminders and explanations throughout your code. This is especially helpful if this code is likely to be edited by another developer in the future.

There are two methods of using comments. For single-line comments, use two slash characters (**//**) at the beginning of the line. You can put single-line comments on the same line as a statement, as long as it comes after the statement. It does not need to be closed, as a line break effectively closes it.

```
// This is a single-line comment.
```

Multiple-line comments use the same syntax that you've seen in CSS. Everything within the **/* */** characters is ignored by the browser. You can use it to "comment out" notes and even chunks of the script when troubleshooting.

```
/* This is a multi-line comment.
Anything between these sets of characters will be
completely ignored when the script is executed.
This form of comment needs to be closed. */
```

I'll be using the single-line comment notation to add short explanations to example code, and we'll make use of the **alert()** function we saw earlier (Figure 19-5) so we can quickly view the results of our work.

Variables

If you're anything like me, the very term "variables" triggers nightmarish flashbacks to eighth grade math class. The premise is pretty much the same, though your teacher doesn't have a bad comb-over this time around.

A variable is like an information container.

A variable is like an information container. You give it a name and then assign it a value, which can a number, text string, an element in the DOM, or a function—anything, really. This gives us a convenient way to reference that value later by name. The value itself can be modified and reassigned in whatever way our scripts' logic dictates.

The following declaration creates a variable with the name "foo" and assigns it the value 5:

```
var foo = 5;
```

We start by declaring the variable using the **var** keyword. The single equals sign (**=**) indicates that we are assigning it a value. Because that's the end of our statement, we end the line with a semicolon. Variables can also be declared without the **var** keyword, which impacts what part of your script will have access to the information they contain. We'll discuss that further in the Variable Scope and the var keyword section later on in this chapter.

You can use anything you like as a variable name, but make sure it's a name that will make sense to you later on. You wouldn't want to name a variable

something like "data"; it should describe the information it contains. In our very specific example above, "numberFive" might be a more useful name than "foo." There are a few rules around variable naming:

- It must start with a letter or an underscore.
- It may contain letters, digits, and underscores in any combination.
- It may not contain character spaces. As an alternative, use underscores in place of spaces or close up the space and use camel case instead (for example, `my_variable` or `myVariable`).
- It may not contain special characters (`! . , / \ + * =` etc.).

You can change the value of a variable at any time by re-declaring it anywhere in your script. Remember: JavaScript is case-sensitive, and so are those variable names.

Data types

The values we assign to variables fall under a few distinct data types.

Undefined

The simplest of these data types is likely "undefined." If we declare a variable by giving it a name but no value, that variable contains a value of "undefined."

```
var foo;
alert(foo); // This will open a dialog containing "undefined".
```

Odds are you won't find a lot of use for this right away, but it's worth knowing for the sake of troubleshooting some of the errors you're likely to encounter early on in your JavaScript career. If a variable has a value of "undefined" when it shouldn't, you may want to double-check that it has been declared correctly or that there isn't a typo in the variable name. (We've all been there.)

Null

Similar to the above, assigning a variable of "null" (again, case-sensitive) simply says, "Define this variable, but give it no inherent value."

```
var foo = null;
alert(foo); // This will open a dialog containing "null".
```

Numbers

You can assign variables numeric values.

```
var foo = 5;
alert(foo); // This will open a dialog containing "5".
```

The word "foo" now means the exact same thing as the number five as far as JavaScript is concerned. Because JavaScript is "loosely typed," we don't have to tell our script to treat the variable `foo` as the *number* five.

The variable behaves the same as the number itself, so you can do things to it that you would do to any other number using classic mathematical notation: **+**, **-**, *****, and **/** for plus, minus, multiply, and divide, respectively. In this example, we use the plus sign (**+**) to add **foo** to itself (**foo + foo**).

```
var foo = 5;
alert(foo + foo); // This will alert "10".
```

Strings

Another type of data that can be saved to a variable is a string, which is basically a line of text. Enclosing characters in a set of single or double quotes indicates that it's a string, as shown here:

```
var foo = "five";
alert( foo ); // This will alert "five"
```

The variable **foo** is now treated exactly the same as the word "five". This applies to any combination of characters: letters, numbers, spaces, and so on. If the value is wrapped in quotation marks, it will be treated as a string of text. If we were to wrap the number five (5) in quotes and assign it to a variable, that variable wouldn't behave as a number; instead, it would behave as a string of text containing the character "5."

Earlier we saw the plus (**+**) sign used to add numbers. When the plus sign is used with strings, it sticks the strings together (called concatenation) into one long string, as shown in this example.

```
var foo = "bye"
alert (foo + foo);  // This will alert "byebye"
```

Notice what the alert returns in the following example when we define the value 5 in quotation marks, treating it as a string instead of a number.

```
var foo = "5";
alert( foo + foo ); // This will alert "55"
```

If we concatenate a string and a number, JavaScript will assume that the number should be treated as a string as well, since the math would be impossible.

```
var foo = "five";
var bar = 5;
alert( foo + bar ); // This will alert "five5"
```

Booleans

We can also assign a variable a "true" or "false" value. This is called a Boolean value, and it is the lynchpin for all manner of advanced logic. Boolean values use the **true** and **false** keywords built into JavaScript, so quotation marks are not necessary.

```
var foo = true; // The variable "foo" is now true
```

Just as with numbers, if we were to wrap the value above in quotation marks, we'd be saving the *word* "true" to our variable instead of the inherent value of "true" (i.e., "not false").

In a sense, everything in JavaScript has either an inherently "true" or "false" value. "null", "undefined", "0", and empty strings (**""**) are all inherently false, while every other value is inherently true. These values, although not identical to the Booleans "true" and "false", are commonly referred to as being "truthy" and "falsy." I promise I didn't make that up.

Arrays

An array is a group of multiple values (called members) that can be assigned to a single variable. The values in an array are said to be indexed, meaning you can refer to them by number according to the order in which they appear in the list. The first member is given the index number 0, the second is 1, and so on, which is why one almost invariably hears us nerds start counting things at zero—because that's how JavaScript counts things, and many other programming languages do the same. We can avoid a lot of future coding headaches by keeping this in mind.

So, let's say our script needs all of the variables we defined earlier. We could define them three times and name them something like **foo1**, **foo2**, and so on, or we can store them in an array, indicated by square brackets (**[ ]**).

```
var foo = [5, "five", "5"];
```

Now anytime you need to access any of those values, you can grab them from the single **foo** array by referencing their index number:

```
alert( foo[0] ); // Alerts "5"
alert( foo[1] ); // Alerts "five"
alert( foo[2] ); // Also alerts "5"
```

Comparison operators

Now that we know how to save values to variables and arrays, the next logical step is knowing how to compare those values. There is a set of special characters called comparison operators that evaluate and compare values in different ways:

`==`	Is equal to
`!=`	Is not equal to
`===`	Is identical to (equal to and of the same data type)
`!==`	Is not identical to
`>`	Is greater than
`>=`	Is greater than or equal to
`<`	Is less than
`<=`	Is less than or equal to

There's a reason all of these definitions read as parts of a statement. In comparing values, we're making an assertion, and the goal is to obtain a result that is either inherently true or inherently false. When we compare two values, JavaScript evaluates the statement and gives us back a Boolean value depending on whether the statement is true or false.

```
alert( 5 == 5 ); // This will alert "true"

alert( 5 != 6 ); // This will alert "true"

alert( 5 < 1 ); // This will alert "false"
```

Equal versus identical

The tricky part is understanding the difference between "equal to" (==) and "identical to" (===). We already learned that all of these values fall under a certain data type. For example, a string of "5" and a number 5 are similar, but they're not quite the same thing.

Well, that's exactly what === is meant to check.

```
alert( "5" == 5 ); // This will alert "true". They're both "5".

alert( "5" === 5 ); // This will alert "false". They're both "5", but
they're not the same data type.

alert( "5" !== 5 ); // This will alert "true", since they're not the
same data type.
```

WARNING

Be careful not to accidentally use a single equals sign, or you'll be reassigning the value of the first variable to the value of the second variable!

Even if you have to read it a couple of times, understanding the preceding sentence means you've already begun to adopt the special kind of crazy one needs to be a programmer. Welcome! You're in good company.

Mathematical Operators

The other type of operator is a mathematical operator, which performs mathematical functions on numeric values. We touched briefly on the straightforward mathematical operators for add (+), subtract (-), multiply (*), and divide (/). There are also some useful shortcuts you should be aware of:

+= Adds the value to itself

++ Increases the value of a number (or a variable containing a number value) by 1

-- Decreases the value of a number (or a variable containing a number value) by 1

If/else statements

If/else statements are how we get JavaScript to ask itself a true/false question. They are more or less the foundation for all the advanced logic that can be written in JavaScript, and they're about as simple as programming gets. In fact, they're almost written in plain English. The structure of a conditional statement is as follows.

```
if( true ) {
    // Do something.
}
```

It tells the browser "if this condition is met, then execute the commands listed between the curly braces (`{ }`)." JavaScript doesn't care about whitespace in our code, remember, so the spaces on either side of the `( true )` are purely for the sake of more readable code.

Here is a simple example using the array we declared earlier:

```
var foo = [5, "five", "5"];

if( foo[1] === "five" ) {
  alert("This is the word five, written in plain English.");
}
```

Since we're making a comparison, JavaScript is going to give us a value of either "true" or "false". The highlighted line of code breaks says "true or false: the value of the **`foo`** variable with an index of **`1`** is identical to the word 'five'?"

In this case, the alert would fire because the **`foo`** variable with an index of **`1`** (the second in the list, if you'll remember) is identical to "five". In this case, it is indeed true, and the alert fires.

We can also explicitly check if something is false, by using the **`!=`** comparison operator that reads as "not equal to."

```
if( 1 != 2 ) {
  alert("If you're not seeing this, we have bigger problems than
    JavaScript.");
  // 1 is never equal to 2, so we should always see this alert.
}
```

I'm not much good at math, but near as I can tell, 1 will never be equal to 2. JavaScript says, "That '1 is not equal to 2' line is a true statement, so I'll run this code."

If the statement doesn't evaluate to "true", the code inside of the curly braces will be skipped over completely:

```
if( 1 == 2 ) {
  alert("If you're seeing this, we have bigger problems than
    JavaScript.");
// 1 is not equal to 2, so this code will never run.
}
```

Idiomatic JavaScript

There is an effort in the JavaScript community to create a style guide for writing JavaScript code. The document "Principles of Writing Consistent, Idiomatic JavaScript" states the following: "All code in any code-base should look like a single person typed it, no matter how many people contributed." To achieve that goal, a group of developers has written an Idiomatic Style Manifesto that describes how whitespace, line breaks, quotation marks, functions, variables, and more should be written to achieve "beautiful code." Learn more about it at *github.com/rwldrn/idiomatic.js/*.

That covers "if," but what about "else"?

Lastly—and I promise we're almost done here—what if we want to do one thing if something is true and something *else* if that thing is false? We could write two if statements, but that's a little clunky. Instead, we can just say "else, do something...else."

```
var test = "testing";
if( test == "testing" ) {
        alert( "You haven't changed anything." );
} else {
   alert( "You've changed something!" );
}
```

Changing the value of the **testing** variable to something else—anything other than the word "testing"—will trigger the alert "You've changed something!"

Loops

There are cases in which we'll want to go through every item in an array and do something with it, but we won't want to write out the entire list of items and repeat ourselves a dozen or more times. You are about to learn a technique of *devastating power*, readers: loops.

I know. Maybe I made loops sound a little more exciting than they seem, but they *are* incredibly useful. With what we've covered already, we're getting good at dealing with single variables, but that can get us only so far. Loops allow us to easily deal with huge sets of data.

Say we have a form that requires none of the fields to be left blank. If we use the DOM to fetch every text input on the page, the DOM provides an array of every text input element. (I'll tell you more about how the DOM does this in the next chapter.) We could check every value stored in that array one item at a time, sure, but that's a lot of code and a maintenance nightmare. If we use a loop to check each value, we won't have to modify our script, regardless of how many fields are added to or removed from the page. Loops allow us to act on every item in an array, regardless of that array's size.

There are several ways to write a loop, but the **for** method is one of the most popular. The basic structure of a **for** loop is as follows:

```
for( initialize the variable;  test the condition;  alter the value; )
{
   // do something
}
```

Here is an example of a **for** loop in action.

```
for( var i = 0; i <= 2; i++ ) {
  alert( i ); // This loop will trigger three alerts, reading "0", "1",
    and "2" respectively.
}
```

exercise 19-1| English to JavaScript translation

In this quick exercise, you can get a feel for variables, arrays, and if/else statement by translating the statements written in English into lines of JavaScript code. You can find the answers in Appendix A.

1. Create a variable "friends" and assign it an array with four of your friends' names.
2. Show the user a dialog that displays the third name in your list of "friends".
3. Create the variable "name" and assign it a string value that is your first name.
4. If the value of "name" is identical to "Jennifer", show the user a dialog box that says "That's my name too!"
5. Create the variable "myVariable" and assign it a number value between 1 and 10. If "myVariable" is greater than five, show the user a dialog that says "upper." If not, show the user a dialog that says "lower."

That's a little dense, so let's break it down part-by-part:

`for ()`

First, we're calling the **`for`** statement, which is built into JavaScript. It says, "For every time this is true, do this." Next we need to supply that statement with some information.

`var i = 0;`

This creates a new variable, **`i`**, with its value set to zero. You can tell it's a variable by the single equals sign. More often than not you'll see coders using the letter "i" (short for "index") as the variable name, but keep in mind that you could use any variable name in its place. It's a common convention, not a rule.

We set that initial value to "0" because we want to stay in the habit of counting from zero up. That's where JavaScript starts counting, after all.

`1 <=2;`

With **`i <= 2;`**, we're saying "for as long as **`i`** is less-than or equal to 2, keep on looping." Since we're counting from zero, that means the loop will run three times.

`i++`

Finally, **`i++`** is shorthand for "every time this loop runs, add one to the value of **`i`** (**`++`** is one of the mathematical shortcut operators we saw earlier). Without this step, **`i`** would always equal zero, and the loop would run forever! Fortunately, modern browsers are smart enough not to let this happen. If one of these three pieces is missing, the loop simply won't run at all.

{ *script* **}**

Anything inside of those curly braces is executed once for each time the loop runs, which is three times in this case. That **`i`** variable is available for use in the code the loop executes as well, as we'll see next.

Let's go back to the "check each item in an array" example. How would we write a loop to do that for us?

```
var items = ["foo", "bar", "baz"]; // First we create an array.
for( var i = 0; i <= items.length; i++ ) {
  alert( items[i] ); // This will alert each item in the array.
}
```

This example differs from our first loop in two key ways:

`items.length`

Instead of using a number to limit the number of times the loop runs, we're using a property built right into JavaScript to determine the "length" of our array, which is the number of items it contains. **`.length`**

is just one of the standard properties and methods of the **`Array`** object in JavaScript.

`items[i]`

Remember how I mentioned that we can use that **`i`** variable inside of the loop? Well, we can use it to reference each index of the array. Good thing we started counting from zero; if we had set the initial value of **`i`** to 1, the first item in the array would have been skipped.

Now no matter how large or small that array should become, the loop will execute only as many times as there are items in the array, and will always hold a convenient reference to each item in the array.

There are literally dozens of ways to write a loop, but this is one of the more common patterns you're going to encounter out there in the wild. Developers use loops to perform a number of tasks, such as:

- Looping through a list of elements on the page and checking the value of each, applying a style to each, or adding/removing/changing an attribute on each. For example, we could loop through each element in a form and ensure that users have entered a valid value for each before they proceed.
- Creating a new array of items in an original array that have a certain value. We check the value of each item in the original array within the loop, and if the value matches the one we're looking for, we populate a new array with only those items. This turns the loop into a filter, of sorts.

Functions

I've introduced you to a few functions already in a sneaky way. Here's an example of a function that you might recognize:

```
alert("I've been a function all along!");
```

A function is a bit of code that doesn't run until it is referenced or called. **`alert()`** is a function built into our browser. It's a block of code that runs only when we explicitly tell it to. In a way, we can think of a function as a variable that contains *logic*, in that referencing that variable will run all the code stored inside it.

All functions share a common pattern (Figure 19-6). The function name is always immediately followed by a set of parentheses (no space), then a pair of curly braces that contain their associated code. The parentheses sometimes contain additional information used by the function called arguments. Arguments are data that can influence how the function behaves. For example, the **`alert`** function we know so well accepts a string of text as an argument, and uses that information to populate the resulting dialog.

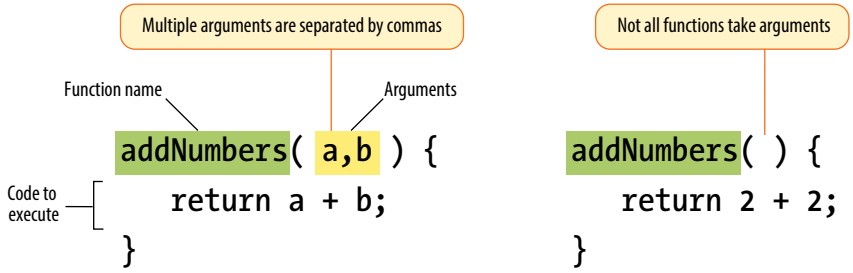


Figure 19-6. The structure of a function.

There are two types of functions: those that come "out-of-the-box" (native JavaScript functions) and those that you make up yourself (custom functions). Let's look at each.

Native functions

There are hundreds of predefined functions built into JavaScript, including:

`alert()`, `confirm()`, and `prompt()`

These functions trigger browser-level dialog boxes.

`Date()`

Returns the current date and time.

`parseInt("123")`

This function will, among other things, take a string data type containing numbers and turn it into a number data type. The string is passed to the function as an argument.

`setTimeout(`*`functionName`*`, 5000)`

Will execute a function after a delay. The function is specified in the first argument, and the delay is specified in milliseconds in the second (in the example, 5000 milliseconds equals 5 seconds).

There are scores more beyond this, as well.

Custom functions

To create a custom function, we type the **`function`** keyword followed by a name for the function, followed by opening and closing parentheses, followed by opening and closing curly brackets.

```
function name() {
   // Our function code goes here.
}
```

Just as with variables and arrays, the function's name can be anything you want, but all the same naming syntax rules apply.

If we were to create a function that just alerts some text (which is a little redundant, I know), it would look like this:

```
function foo() {
   alert("Our function just ran!");
      // This code won't run until we call the function 'foo()'
}
```

We can then call that function and execute the code inside it anywhere in our script by writing the following:

```
foo(); // Alerts "Our function just ran!"
```

We can call this function any number of times throughout our code. It saves a lot of time and redundant coding.

Arguments

An argument is a value or data that a function uses when it runs.

Having a function that executes the exact same code throughout your script isn't likely to be all that useful. We can "pass arguments" (provide data) to native and custom functions in order to apply a function's logic to different sets of data at different times.

To hold a place for the arguments, add one or more comma-separated variables in the parentheses at the time the function is defined. Then, when we call that function, anything we include between the parentheses will be passed into that variable as the function executes. This might sound a little confusing, but it's not so bad once you see it in action.

For example, let's say we wanted to create a very simple function that alerts the number of items contained in an array. We've already learned that we can use **`.length`** to get the number of items in an array, so we just need a way to pass the array to be measured into our function. We do that by supplying the array to be measured as an argument. In order to do that, we specify a variable name in the parentheses when we define our custom function. That variable will then be available inside of the function and will contain whatever argument we pass when we call the function.

```
function alertArraySize(arr) {
        alert(arr.length);
}
```

Now any array we specify between the parentheses when we call the function will be passed to the function with the variable name **`arr`**. All we need to do is get its length.

```
var test = [1,2,3,4,5];
alertArraySize(test); // Alerts "5"
```

Returning a value

This part is particularly wild, and incredibly useful.

It's pretty common to use a function to calculate something and then give you a value that you can use elsewhere in your script. We could accomplish

this using what we know now, through clever application of variables, but there's a much easier way.

The **return** keyword inside a function effectively turns that function into a variable with a dynamic value! This one is a little easier to show than it is to tell, so bear with me while we consider this example.

```
function addNumbers(a,b) {
   return a + b;
}
```

We now have a function that accepts two arguments and adds them together. That wouldn't be much use if the result always lived inside that function, because we wouldn't be able to use the result anywhere else in our script. Here we use the **return** keyword to pass the result out of the function. Now any reference you make to that function gives you the result of the function—just like a variable would.

```
alert( addNumbers(2,5) ); // Alerts "7"
```

In a way, the **addNumbers** function is now a variable that contains a dynamic value: the value of our calculation. If we didn't return a value inside of our function, the preceding script would alert "undefined", just like a variable that we haven't given a value.

The **return** keyword has one catch. As soon as JavaScript sees that it's time to return a value, the function ends. Consider the following:

```
function bar() {
   return 3;
   alert("We'll never see this alert.");
}
```

When you call this function using **bar()**, the alert on the second line never runs. The function ends as soon as it sees it's time to return a value.

Variable scope and the var keyword

There are times when you'll want a variable that you've defined within a function to be available anywhere throughout your script. Other times, you may want to restrict it and make it available *only* to the function it lives in. This notion of the availability of the variable is known as its scope. A variable that can be used by any of the scripts on your page is globally scoped, and a variable that's available only within its parent function is locally scoped.

JavaScript variables use functions to manage their scope. If a variable is defined outside of a function, it will be globally scoped and available to all scripts. When you define a variable within a function and you want it to be used only by that function, you can flag it as locally scoped by preceding the variable name with the **var** keyword.

```
var foo = "value";
```

To expose a variable within a function to the global scope, we omit the **var** keyword and simply define the variable:

```
foo = "value";
```

You need to be careful about how you define variables within functions, or you could end up with unexpected results. Take the following JavaScript snippet, for example:

```
function double( num ){
  total = num + num;
  return total;
}
var total = 10;
var number = double( 20 );
alert( total ); // Alerts 40.
```

You may expect that because you specifically assigned a value of 10 to the variable **total,** the **alert(total)** function at the end of the script would return 10. But because we didn't scope the **total** variable in the function with the **var** keyword, it bleeds into the global scope. Therefore, although the variable **total** is set to 10, the following statement runs the function and grabs the value for **total** defined there. Without the **var**, the variable "leaked out."

As you can see, the trouble with global variables is that they'll be shared throughout all the scripting on a page. The more variables that bleed into the global scope, the better the chances you'll run into a "collision" in which a variable named elsewhere (in another script altogether, even) matches one of yours. This can lead to variables being inadvertently redefined with unexpected values, which can lead to errors in your script.

Remember that we can't always control all the code in play on our page. It's very common for pages to include code written by third parties, for example:

- Scripts to render advertisements
- User-tracking and analytics scripts
- Social media "share" buttons

It's best not to take any chances on variable collisions, so when you start writing scripts on your own, locally scope your variables whenever you can (see sidebar).

This concludes our little (OK, not so little) introductory tour of JavaScript syntax. There's a lot more to it, but this should give you a decent foundation for learning more on your own and being able to interpret scripts when you see them. We have just a few more JavaScript-related features to tackle before we look at a few examples.

Keeping variables out of the global scope

If you want to be sure that all of your variables stay out of the global scope, you can put all of the your JavaScript in the following wrapper:

```
<script>
(function() {
  //All your code here!
})();
<script>
```

This little quarantining solution is called an IIFE (Independently Invoked Functional Expression), and we owe this method and the associated catchy term to Ben Alman.

The Browser Object

In addition to being able to control elements on a web page, JavaScript also gives you access to and the ability to manipulate the parts of the browser window itself. For example, you might want to get or replace the URL that is in the browser's address bar, or open or close a browser window.

In JavaScript, the browser is known as the **window** object. The window object has a number of properties and methods that we can use to interact with it. In fact, our old friend **alert()** is actually one of the standard browser object methods. Table 19-1 lists just a few of the properties and methods that can be used with **window** to give you an idea of what's possible.

Table 19-1. *Browser properties and methods*

Property/method	Description
`event`	Represents the state of an event
`history`	Contains the URLs the user has visited within a browser window
`location`	Gives read/write access to the URI in the address bar
`status`	Sets or returns the text in the status bar of the window
`alert()`	Displays an alert box with a specified message and an OK button
`close()`	Closes the current window
`confirm()`	Displays a dialog box with a specified message and an OK and a Cancel button
`focus()`	Sets focus on the current window

Events

Event handlers "listen" for certain document, browser, or user actions and bind scripts to those actions.

JavaScript can access objects in the page and the browser window, but did you know it's also "listening" for certain events to happen? An event is an action that can be detected with JavaScript, such as when the document loads or when the user clicks on an element or just moves her mouse over it. HTML 4.0 made it possible for a script to be tied to events on the page whether initiated by the user, the browser itself, or other scripts. This is known as event binding.

In scripts, an event is identified by an event handler. For example, the **onload** event handler triggers a script when the document loads, and the **onclick** and **onmouseover** handlers trigger a script when the user clicks or mouses over an element, respectively. Table 19-2 lists some of the most common event handlers.

Table 19-2. Common events

Event handler	Event description
onblur	An element loses focus
onchange	The content of a form field changes
onclick	The mouse clicks an object
onerror	An error occurs when the document or an image loads
onfocus	An element gets focus
onkeydown	A key on the keyboard is pressed
onkeypress	A key on the keyboard is pressed or held down
onkeyup	A key on the keyboard is released
onload	A page or an image is finished loading
onmousedown	A mouse button is pressed
onmousemove	The mouse is moved
onmouseout	The mouse is moved off an element
onmouseover	The mouse is moved over an element
onmouseup	A mouse button is released
onsubmit	The submit button is clicked in a form

There are three common methods for applying event handlers to items within our pages:

- As an HTML attribute
- As a method attached to the element
- Using **addEventListener**

In the examples of the latter two approaches, we'll use the **window** object. Any events we attach to **window** apply to the entire document. We'll be using the **onclick** event in all of these as well.

As an HTML attribute

You can specify the function to be run in an attribute in the markup as shown in the following example.

```
<body onclick="myFunction();"> /* myFunction will now run when the user
clicks anything within 'body' */
```

Although still functional, this is an antiquated way of attaching events to elements within the page. It should be avoided for the same reason we avoid using **style** attributes in our markup to apply styles to individual elements. In this case, it blurs the line between the semantic layer and behavioral layers of our pages, and can quickly lead to a maintenance nightmare.

As a method

This is another somewhat dated approach to attaching events, though it does keep things strictly within our scripts. We can also attach functions using helpers already built into JavaScript.

```
window.onclick = myFunction; /* myFunction will run when the user
clicks anything within the browser window */
```

We can also use an anonymous function rather than a predefined one:

```
window.onclick = function() {
    /* Any code placed here will run when the user clicks anything
    within the browser window */
};
```

This approach has the benefit of both simplicity and ease of maintenance, but does have a fairly major drawback: we can bind only one event at a time with this method.

```
window.onclick = myFunction;

window.onclick = myOtherFunction;
```

In the example just shown, the second binding overwrites the first, so when the user clicks inside the browser window, only **myOtherFunction** will run. The reference to **myFunction** is thrown away.

addEventListener

NOTE

For more information on `addEventListener`*, see the "element.addEventListener" page on the Mozilla Developer Network (developer.mozilla.org/en/DOM/element.addEventListener)*

Although a little more complex at first glance, this approach allows us to keep our logic within our scripts and allows us to perform multiple bindings on a single object. The syntax is a bit more verbose. We start by calling the **addEventListener** method of the target object, and then specify the event in question and the function to be executed as two arguments.

```
window.addEventListener("click", myFunction);
```

Notice that we omit the preceding "on" from the event handler with this syntax.

Like the previous method, **addEventListener** can be used with an anonymous function as well:

```
window.addEventListener("click", function(e) {

});
```

Putting It All Together

Now you have been introduced to many of the important building blocks of JavaScript. You've seen variables, data types, and arrays. You've met if/else statements, loops, and functions. You know your browser objects from your event handlers. That's a lot of bits and pieces. Let's walk through a few simple script examples to see how they get put together.

Example 1: A tale of two arguments

Here's a simple function that accepts two arguments and returns the greater of the two values.

```
greatestOfTwo( first, second ) {
  if( first > second ) {
    return first;
  } else {
    return second;
  }
}
```

We start by naming our function: "greatestOfTwo". We set it up to accept two arguments, which we'll just call "first" and "second" for want of more descriptive words. The function contains an if/else statement that returns "first" if the first argument is greater than the second, and returns "second" if it isn't.

Example 2: The longest word

Here's a function that accepts an array of strings as a single argument and returns the longest string in the array.

```
longestWord( strings ) {
  var longest = strings[0];

  for( i = 1; i < strings.length; i++ ) {
    if ( strings[i].length > longest.length ) {
      longest = strings[i];
    }
  }
  return longest;
}
```

First, we name the function and allow it to accept a single argument. Then, we set the **`longest`** variable to an initial value of the first item in the array: **`strings[0]`**. We start our loop at 1 instead of 0 since we already have the first value in the array captured. Each time we iterate through the loop, we compare the length of the current item in the array to the length of the value saved in the **`longest`** variable. If the current item in the array contains more characters than the current value of the **`longest`** variable, we change the value of **`longest`** to that item. If not, we do nothing. After the loop is complete we return the value of **`longest`**, which will now contain the longest string in the array.

exercise 19-2 | **You try it**

In this exercise you will write script that updates the page's title in the browser window with a "new messages" count. You may have encountered this sort of script in the wild from time to time. We're going to assume for the sake of the exercise that this is going to become part of a larger web app some day, and we're tasked only with updating the page title with the current "unread messages" count.

I've created a document for you already (***title.html***), which is available in the ***materials*** folder for this chapter on *learningwebdesign.com*.

1. Start by opening ***title.html*** in a browser. You'll see a blank page, with the title tag already filled out. If you look up at the top of your browser window, you'll notice it reads "Million Dollar WebApp".
2. Now open the document in a text editor. You'll find a **`script`** element containing a comment just before the closing **`</body>`** tag. Feel free to delete the comment.
3. If we're going to be changing the page's title, we should save the original first. Create a variable named **`originalTitle`**. For its value, we'll have the browser get the title of the document using the DOM method **`document.title`**. Now we have a saved reference to the page title at the time the page is loaded. This variable should be global, so we'll declare it outside any functions.

```
var originalTitle = document.title;
```

4. Next, we'll define a function so we can reuse the script whenever it's needed. Let's call the function something easy to remember, so we know at a glance what it does when we encounter it in our code later. "showUnreadCount" works for me, but you can name it whatever you'd like.

```
var originalTitle = document.title;

function showUnreadCount() {
}
```

5. We need to think about what the function needs to make it useful. This function does something with the unread message count, so its argument is a single number referred to as "unread" in this example.

```
var originalTitle = document.title;

function showUnreadCount( unread ) {
}
```

6. Now let's add the code that runs for this function. We want the document title for the page to display the title of the document plus the count of unread messages. Sounds like a job for concatenation (**+**)! Here we set the **`document.title`** to be (**=**) whatever string was saved for **`originalTitle`** plus the number in **`showUnreadCount`**. As we learned earlier, JavaScript combines a string and a number as though they were both strings.

```
var originalTitle = document.title;

function showUnreadCount( unread ) {
  document.title = originalTitle + unread;
}
```

7. Let's try out our script before we go too much further. Below where you defined the function and the **`originalTitle`** variable, enter **`showUnreadCount( 3 );`**. Now save the page and reload it in your browser (Figure 19-7).

```
var originalTitle = document.title;

function showUnreadCount( unread ) {
  document.title = originalTitle + unread;
}
showUnreadCount(3);
```

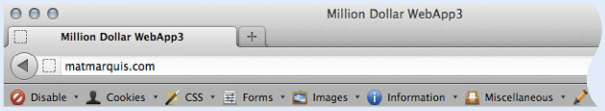


Figure 19-7. *Our title tag has changed! It's not quite right yet, though.*

8. Our script is working, but it's not very easy to read. Fortunately, there's no limit on the number of strings we can combine at once. Here we're adding additional strings that wrap the count value and the words "new messages" in parentheses (Figure 19-8).

```
var originalTitle = document.title;

function showUnreadCount( unread ) {
  document.title = originalTitle + " (" + unread
+ " new messages!) ";
}
showUnreadCount(3);
```

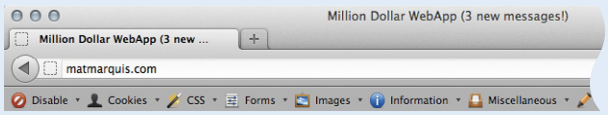


Figure 19-8. *Much better!*

Test Yourself

We covered a lot of new material in this chapter. Here's a chance to test what sunk in.

1. Name one good thing and one bad thing about linking to an external *.js* file.

2. Given the following array:

   ```
   var myArray = [1, "two", 3, "4"]
   ```

   write what the alert message will say for each of these examples:

   a. `alert( myArray[0] );`

   b. `alert( myArray[0] + myArray[1] );`

   c. `alert( myArray[2] + myArray[3] );`

   d. `alert( myArray[2] - myArray[0] );`

3. What will each of these alert messages say?

   a.
   ```
   var foo = 5;
   foo += 5;
   alert( foo );
   ```

   b.
   ```
   var foo = 5;
   alert(foo++);
   ```

   c.
   ```
   var foo = 2;
   alert( foo + " " + "remaining");
   ```

   d.
   ```
   var foo = "Mat";
   var bar = "Jennifer";
   if( foo.length > bar.length ) {
     alert( foo  + " is longer." );
   } else {
     alert( bar + " is longer." );
   }
   ```

   e.
   ```
   alert( 10 === "10" );
   ```

4. Describe what this does:

   ```
   for( var i = 0; i <= items.length;  i++ ) {  }
   ```

For Further Reading

Does this whet your appetite for more? Do you see yourself making things *happen* with JavaScript? Well, there is certainly no shortage of JavaScript tutorials online to get you started. I also recommend the following books (all of them just happen to be published by O'Reilly Media):

- *Learning JavaScript*, by Shelley Powers
- *JavaScript & jQuery: The Missing Manual*, by David Sawyer McFarland
- *JavaScript: The Good Parts*, by Douglas Crockford

5. What is the problem with globally scoped variables?

6. Match what's happening with its event handler.

a. `onload`	1. The user finishes a form and hits the submit button
b. `onchange`	2. The page finishes loading
c. `onfocus`	3. The pointer hovers over a link
d. `onmouseover`	4. A text entry field is selected and ready for typing
e. `onsubmit`	5. A user changes her name in a form field

USING JAVASCRIPT

by Mat Marquis

IN THIS CHAPTER

Using the DOM to access and change elements, attributes, and contents

Using polyfills to make browser versions work consistently

Using JavaScript libraries

A brief introduction to Ajax

Now that you have a sense for the language of JavaScript, let's look at some of the ways we can put it to use in modern web design. First, we'll explore DOM scripting, which allows us to manipulate the elements, attributes, and text on a page. I'll introduce you to some ready-made JavaScript and DOM scripting resources, so you don't have to go it alone. You'll learn about "polyfills," which provide older browsers with modern features and normalize functionality. I'll also introduce you to JavaScript libraries that make developers' lives easier with collections of polyfills and shortcuts for common tasks.

Meet the DOM

You've seen references to the Document Object Model (DOM for short) several times throughout this book, but now is the time to give it the attention it deserves. The DOM gives us a way to access and manipulate the contents of a document. We commonly use it for HTML, but the DOM can be used with any XML language as well. And although we're focusing on its relationship with JavaScript, it is worth noting that the DOM can be accessed by other languages too, such as PHP, Ruby, Python, C++, Java, Perl, and more. Although DOM Level 1 was released by the W3C in 1998, it was nearly five years later that DOM scripting began to gain steam.

The DOM gives us a way to access and manipulate the contents of a document.

The DOM is a programming interface (an API) for HTML and XML pages. It provides a structured map of the document, as well as a set of methods to interface with the elements contained therein. Effectively, it translates our markup into a format that JavaScript (and other languages) can understand. It sounds pretty dry, I know, but the basic gist is that the DOM serves as a map to all the elements on a page. We can use it to find elements by their names or attributes, then add, modify, or delete elements and their content.

Without the DOM, JavaScript wouldn't have any sense of a document's contents—and by that, I mean the *entirety* of the document's contents. Everything from the page's `doctype` to each individual letter in the text can be accessed via the DOM and manipulated with JavaScript.

The node tree

A simple way to think of the DOM is in terms of the document tree (Figure 20-1). You saw documents diagrammed in this way when you were learning about CSS selectors.

```
<html>
<head>
  <title>Document title</title>
  <meta charset="utf-8">
</head>
<body>
  <div>
    <h2>Subhead</h2>
    <p>Paragraph text with a <a href="foo.html">link</a> here.</p>
  </div>
  <div>
    <p>More text here.</p>
  </div>
</body>
</html>
```

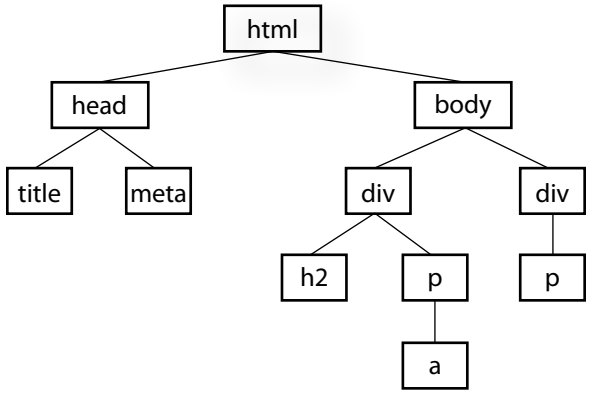


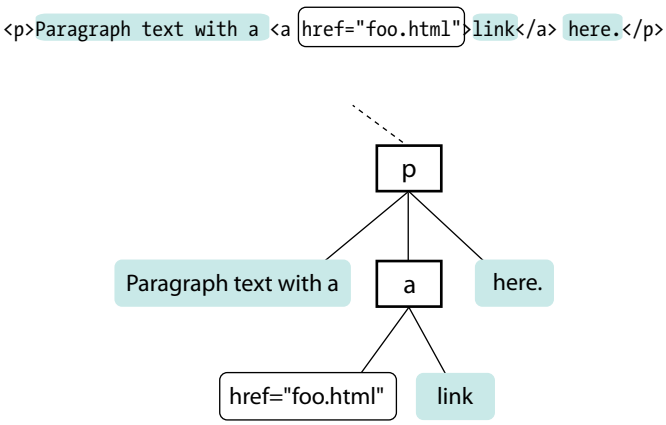
Figure 20-1. A simple document tree

AT A GLANCE

The DOM is a collection of nodes:

- Element nodes
- Attribute nodes
- Text nodes

Figure 20-2. The nodes within the first p element in our sample document.



Each element within the page is referred to as a node. If you think of the DOM as a tree, each node is an individual branch that can contain further branches. But the DOM allows deeper access to the content than CSS because it treats the actual content as a node as well. Figure 20-2 shows the structure of the first **p** element. The element, its attributes, and its contents are all nodes in the DOM's node tree.

It also provides a standardized set of methods and functions through which JavaScript can interact with the elements on our page. Most DOM scripting involves reading from and writing to the document.

There are several ways to use the DOM to find what you want in a document. Let's go over some of the

specific methods we can use for accessing objects defined by the DOM (we JS folks call this "crawling the DOM" or "traversing the DOM"), as well as some of the methods for manipulating those elements.

Accessing DOM nodes

The **document** object in the DOM identifies the page itself, and more often than not will serve as the starting point for our DOM crawling. The **document** object comes with a number of standard properties and methods for accessing collections of elements. This is reminiscent of the **length** property we learned about in Chapter 19, Introduction to JavaScript. Just as **length** is a standard property of all arrays, the **document** object comes with a number of built-in properties containing information about the document. We then wind our way to the element we're after by chaining those properties and methods together, separated by periods, to form a sort of route through the document.

To give you a general idea of what I mean, the statement in this example says to look on the page (**document**), find the element that has the **id** value "beginner", find the HTML content within that element (**innerHTML**), and save those contents to a variable (**foo**).

```
var foo = document.getElementById( "beginner" ).innerHTML;
```

Because the chains tend to get long, it is also common to see each property or method broken onto its own line to make it easier to read at a glance. Remember, whitespace in JavaScript is ignored, so this has no effect on how the statement is parsed.

```
var foo = document
  .getElementById( "beginner" )
    .innerHTML;
```

There are several methods for accessing nodes in the document.

By element name

getElementsByTagName()

We can access individual elements by the tags themselves using **document.getElementsByTagName()**. This method retrieves any element or elements you specify as an argument.

For example, **document.getElementsByTagName("p")** returns every paragraph on the page, wrapped in something called a collection or nodeList, in the order they appear in the document from top to bottom. nodeLists behave much like arrays. To access specific paragraphs in the nodeList, we reference them by their index, just like an array.

```
var paragraphs = document.getElementsByTagName("p");
```

Based on this variable statement, **paragraphs[0]** is a reference to the first paragraph in the document, **paragraph[1]** refers to the second, and so on.

NOTE

nodeLists are living collections. If you manipulate the document in a nodeList loop—for example, looping through all paragraphs and appending new ones along the way—you can end up in an infinite loop. Good times!

If we had to access each element in the nodeList separately, one at a time... well, it's a good thing we learned about looping through arrays earlier. Loops work the exact same way with a nodeList.

```
var paragraphs = document.getElementsByTagName("p");
for( var i = 0; i < paragraphs.length; i++ ) {
 // do something
}
```

Now we can access each paragraph on the page individually by referencing **paragraphs[i]** inside the loop, just as with an array, but with elements on the page instead of values.

By id attribute value

getElementById()

This method returns a single element based on that element's ID (the value of its **id** attribute), which we provide to the method as an argument. For example, to access this particular image:

```
<img src="photo.jpg" alt="" id="lead-photo">'
```

we include the **id** value as an argument for the **getElementById()** method:

```
var photo = document.getElementById("lead-photo");
```

By class attribute value

getElementsByClassName()

WARNING

This is a relatively new method for accessing DOM nodes. Although **getElementsByClassName()** *is available in the current versions of modern browsers, it will not work in IE8 or below.*

Just as it says on the tin, this allows you to access nodes in the document based on the value of a **class** attribute. This statement assigns any element with a **class** value of "column-a" to the variable **firstColumn** so it can be accessed easily from within a script.

```
var firstColumn = document.getElementsByClassName("column-a");
```

Like **getElementsByTagName**, this returns a nodeList that we can reference by index or loop through one at a time.

By selector

querySelectorAll()

WARNING

Because it is a newer method, **querySelectorAll()** *is available in the current versions of modern browsers, but isn't available in IE7 or below.*

querySelectorAll allows you to access nodes of the DOM based on a CSS-style selector. The syntax of the arguments in the following examples should look familiar to you. It can be as simple as accessing the child elements of a specific element:

```
var sidebarPara = document.querySelectorAll(".sidebar p");
```

or as complex as selecting an element based on an attribute:

```
var textInput = document.querySelectorAll("input[type='text']");
```

Like **getElementsByTagName** and **getElementsByClassName**, **querySelectorAll** returns a nodeList (even if the selector matches only a single element).

Accessing an attribute value

getAttribute()

As I mentioned earlier, elements aren't the only thing you can access with the DOM. To get the value of an attribute attached to an element node, we call **getAttribute()** with a single argument: the attribute name. Let's assume we have an image, *source.jpg*, marked up like this:

```
<img src="stratocaster.jpg" alt="electric guitar" id="lead-image">
```

In the following example, we access that specific image (**getElementbyId**) and save a reference to it in a variable (**bigImage**). At that point, we could access any of the element's attributes (**alt**, **src**, or **id**) by specifying it as an argument in the **getAttribute** method. In the example, we get the value of the **src** attribute and use it as the content in an alert message. (I'm not sure *why* we would ever do that, but it does demonstrate the method.)

```
var bigImage = document.getElementById("lead-image");
alert( bigImage.getAttribute("src") ); // Alerts "stratocaster.jpg".
```

Manipulating nodes

Once we've accessed a node using one of the methods discussed previously, the DOM gives us several built-in methods for manipulating those elements, their attributes, and their contents.

setAttribute()

To continue with the previous example, we saw how we *get* the attribute value, but what if we wanted to *set* the value of that **src** attribute to a new pathname altogether? Use **setAttribute()**! This method requires two arguments: the attribute to be changed and the new value for that attribute.

In this example, we use a bit of JavaScript to swap out the image by changing the value of the **src** attribute.

```
var bigImage = document.getElementById("lead-image");
bigImage.setAttribute("src", "lespaul.jpg");
```

Just think of all the things you could do with a document by changing the values of attributes. Here we swapped out an image, but this same method could be used to make a number of changes throughout our document:

- Update the **checked** attributes of checkboxes and radio buttons based on user interaction elsewhere on the page
- Find the **link** element for our *.css* file and point the **href** value to a different style sheet, changing all the page's styles
- Update a **title** attribute with information on an element's state ("this element is currently selected," for example)

innerHTML

innerHTML gives us a simple method for accessing and changing the text and markup inside an element. It behaves differently from the methods we've covered so far. Let's say we need a quick way of adding a paragraph of text to the first element on our page with a class of **intro**:

```
var introDiv = document.getElementsByClassName("intro");
introDiv.innerHTML = "<p>This is our intro text</p>";
```

The second statement here adds the content of the string to **introDiv** (an element with the **class** value "intro") as a *real live element* because **innerHTML** tells JavaScript to parse the strings " " and " " as markup.

style

The DOM also allows you to add, modify, or remove a CSS style from an element using the **style** property. It works similarly to applying a style with the inline **style** attribute. The individual CSS properties are available as properties of the **style** property. I bet you can figure out what these statements are doing using your new CSS and DOM know-how:

```
document.getElementById("intro").style.color = "#fff";
document.getElementById("intro").style.backgroundColor = "#f58220";
    //orange
```

In JavaScript and the DOM, property names that are hyphenated in CSS (such as **background-color** and **border-top-width**) become camel case (**backgroundColor** and **borderTopWidth**, respectively) so the **-** character isn't mistaken for an operator.

In the examples you've just seen, the **style** property is used to set the styles for the node. It can also be used to get a style value for use elsewhere in the script. This statement gets the background color of the **#intro** element and assigns it to the **brandColor** variable:

```
var brandColor = document.getElementById("intro").style.backgroundColor;
```

Adding and removing elements

So far, we've seen examples of getting and setting nodes in the existing document. The DOM also allows developers to change the document structure itself by adding and removing nodes on the fly. We'll start out by creating new nodes, which is fairly straightforward, and then we'll see how we add the nodes we've created to the page. The methods shown here are more surgical and precise than adding content with **innerHTML**. While we're at it, we'll remove nodes, too.

createElement()

To create a new element, use the aptly named **createElement()** method. This function accepts a single argument: the element to be created. Using this

method is a little counterintuitive at first because the new element doesn't appear on the page right away. Once we create an element in this way, that new element remains floating in the JavaScript ether until we add it to the document. Think of it as creating a *reference* to a new element that lives purely in memory—something that we can manipulate in JavaScript as we see fit, then add to the page once we're ready.

```
var newDiv = document.createElement("div");
```

createTextNode()

If we want to enter text into either an element we've created or an existing element on the page, we can call the **createTextNode()** method. To use it, provide a string of text as an argument, and the method creates a DOM-friendly version of that text, ready for inclusion on the page. Much like **createElement**, this creates a reference to the new text node that we can store in a variable and add to the page when the time comes.

```
var ourText = document.createTextNode("This is our text.");
```

appendChild()

So we've created a new element and a new string of text, but how do we make them part of the document? Enter the **appendChild** method. This method takes a single argument: the node you want to add to the DOM. You call it on the existing element that will be its *parent* in the document structure. Time for an example.

Here we have a simple **div** on the page with the **id** "our-div":

```
<div id="our-div"></div>
```

Let's say we want to add a paragraph to **#our-div** that contains the text "Hello, world". We start by creating the **p** element (**document.createElement()**) as well as a text node for the content that will go inside it (**createTextNode()**).

```
var ourDiv = document.getElementById("our-div");
var newParagraph = document.createElement("p");
var copy = document.createTextNode("Hello, world!");
```

Now we have our element and some text, and we can use **appendChild()** to put the pieces together.

```
newParagraph.appendChild( copy );
ourDiv.appendChild( newParagraph );
```

The first statement appends **copy** (that's our "Hello, world" text node) to the new paragraph we created (**newParagraph**), so now that element has some content. The second line appends the **newParagraph** to the original **div** (**ourDiv**). Now **ourDiv** isn't sitting there all empty in the DOM, and it will display on the page with the content "Hello, world."

You should be getting the idea of how it works. How about a couple more?

insertBefore()

The **insertBefore()** method, as you might guess, inserts an element before another element. It takes two arguments: the first is the node that gets inserted, and the second is the element it gets inserted in front of. You also need to know the parent to which the element will be added.

So, for example, to insert a new heading before the paragraph in this markup:

```
<div id="our-div">
  <p id="our-paragraph">Our paragraph text</p>
</div>
```

we start by assigning variable names to the **div** and the **p** it contains, then create the **h1** element and its text node and put them together, just as we saw in the last example.

```
var ourDiv = document.getElementById("our-div");
var para = document.getElementById("our-paragraph");

var newHeading = document.createElement("h1");
var headingText = document.createTextNode("A new heading");
newHeading.appendChild( headingText );
// Add our new text node to the new heading
```

Finally, in the last statement shown here, the **insertBefore()** method places the **newHeading h1** element before the **para** element inside **ourDiv**.

```
ourDiv.insertBefore( newHeading, para );
```

replaceChild()

The **replaceChild()** method replaces one node with another and takes two arguments. The first argument is the new child (i.e., the node you want to end up with). The second is the node that gets replaced by the first. Like **insertBefore()**, you also need to identify the parent element in which the swap happens. For the sake of simplicity, let's say we start with the following markup:

```
<div id="our-div">
  <div id="swap-me"></div>
</div>
```

and we want to replace the **div** with the **id** "swap-me" with an image. We start by creating a new **img** element and setting the **src** attribute to the pathname to the image file. In the final statement, we use **replaceChild()** to put **newImg** in place of **swapMe**.

```
var ourDiv = document.getElementById("our-div");
var swapMe = document.getElementById("swap-me");
var newImg = document.createElement("img");
// Create a new image element

newImg.setAttribute( "src", "path/to/image.jpg" );
// Give the new image a "src" attribute
ourDiv.replaceChild( newImg, swapMe );
```

removeChild()

To paraphrase my mother, "We brought these elements into this world, and we can take them out again." You remove a node or an entire branch from the document tree with the **removeChild()** method. The method takes one argument, which is the node you want to remove. Remember that the DOM thinks in terms of *nodes*, not just elements, so the child of an element may be the text (node) it contains, not just other elements.

Like **appendChild**, the **removeChild** method is always called on the parent element of the element to be removed (hence, "remove *child*"). That means we'll need a reference to both the parent node and the node we're looking to remove. Let's assume the following markup pattern:

```
<div id="parent">
  <div id="remove-me">
    <p>Pssh, I never liked it here anyway.</p>
  </div>
</div>
```

Our script would look something like this:

```
var parentDiv = document.getElementById("parent");
var removeMe = document.getElementById("remove-me");

parentDiv.removeChild( removeMe );
    // Removes the div with the id "remove-me" from the page.
```

For further reading

That should give you a good idea of what DOM Scripting is all about. Of course, I've just barely scratched the surface of what can be done with the DOM, but if you'd like to learn more, definitely check out the book *DOM Scripting: Web Design with JavaScript and the Document Object Model, Second Edition* by Jeremy Keith and Jeffrey Sambells (Friends of Ed).

Polyfills

You've gotten familiar with a lot of new technologies in this book so far: new HTML5 elements, new ways of doing things with CSS3, using JavaScript to manipulate the DOM, and more. In a perfect world, all browsers would be in lockstep, keeping up with the cutting-edge technologies and getting the established ones right along the way (see Browser Wars sidebar). In that perfect world, browsers that couldn't keep up (I'm looking at you, IE6) would just vanish completely. Sadly, that is not the world we live in, and browser inadequacies remain the thorn in every developer's side.

I'll be the first to admit that I enjoy a good wheel-reinvention. It's a great way to learn, for one thing. For another, it's the reason our cars aren't rolling around on roundish rocks and sections of tree trunk. But when it comes to dealing with every strange browser quirk out there, we don't have to start from scratch. Tons of people smarter than I have run into these issues before

The Browser Wars

JavaScript came about during a dark and lawless time, before the web standards movement, when all the major players in the browser world were—for want of a better term—winging it. It likely won't come as a major surprise to anyone that Netscape and Microsoft implemented radically different versions of the DOM, with the prevailing sentiment being "may the best browser win."

I'll spare you the gory details of the Battle for JavaScript Hill, but the two competing implementations were so different that they were both largely useless, unless you wanted to either maintain two separate code bases or add a "best viewed in Internet Explorer/Netscape" warning label to your sites.

Enter the web standards movement! During this cutthroat time, the W3C was putting together the foundations for the modern-day standardized DOM that we've all come to know and love. Fortunately for us, Netscape and Microsoft got on board with the standards movement. The standardized DOM is supported all the way back to Internet Explorer 5 and Netscape Navigator 6. Unfortunately, Internet Explorer's advancements in this area stagnated for quite some time following IE6. As a result, older versions of IE have a few significant differences from the modern-day DOM. Fortunately with Internet Explorer 9, and soon with 10, they're catching right back up.

Trouble is, your project likely still needs to support those users with older versions of IE. It's a pain, but we're up for it. We have an amazing set of tools at our disposal, such as polyfills and JavaScript libraries full of helper functions, that normalize the strange little quirks we're apt to encounter from browser to browser.

and have already found clever ways to work around them and fix the parts of JavaScript and the DOM where some browsers may fall short. We can use JavaScript to fix JavaScript.

Polyfill is a term coined by Remy Sharp to describe a JavaScript "shim" that normalizes differing behavior from browser to browser.

> *"A shim that mimics a future API providing fallback functionality to older browsers." —Paul Irish*

There's a lot of time travel going on in that quote, but basically what he's saying is that we're making something new work in browsers that don't natively support it—whether that's brand-new technology like detecting a user's physical location or fixing something that one of the browsers just plain got wrong.

There are tons of polyfills out there targeted to specific tasks, such as making old browsers recognize new HTML5 elements or CSS3 Selectors, and new ones are popping up all the time as new problems arise. I'm going to fill you in on the most commonly used polyfills in the modern developer's toolbox as of the release of this book. You may find that new ones are necessary by the time you hit the web design trenches.

HTML5 shiv (or shim)

You may remember seeing this one back in Chapter 5, Marking Up Text, but let's give it a little more attention now that you have some JavaScript under your belt.

An HTML5 shiv/shim is used to enable Internet Explorer 8 and earlier to recognize and style newer HTML5 elements such as `article`, `section`, and `nav`.

How it works

There are several variations on the HTML5 shim/shiv, but they all work in much the same way: crawl the DOM looking for elements that IE doesn't recognize, and then immediately replace them with the same element so they are visible to IE in the DOM. Now any styles we write against those elements work as expected.

Who made it

Sjoerd Visscher originally discovered this technique, and many, many variations of these scripts exist now. Remy Sharp's version is the one likely in widest use today.

How to use it

Every variation on this script has the same requirement: it must be referenced in the `head` of the document, in order to "tell" Internet Explorer about these new elements before it finishes rendering the page.

```
<!--[if lt IE 9]>
    <script src="html5shim.js"></script>
<![endif]-->
```

Potential drawbacks

The major caveat here is that older versions of Internet Explorer that have JavaScript disabled or unavailable will receive unstyled elements.

Where to get it and learn more

- The Wikipedia entry for HTML Shiv (*en.wikipedia.org/wiki/HTML5_Shiv*)
- Remy Sharp's original post (*remysharp.com/2009/01/07/html5-enabling-script*)

Modernizr

Modernizr isn't a polyfill in and of itself, but rather a test suite that can be used to detect the presence of browser features and load polyfills as needed. The Modernizr team also curates a massive repository of polyfills for a huge number of features (see previous note).

NOTE

The polyfill archive maintained by the Modernizr team is available at github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills.

How it works

Modernizr looks for the presence of methods and functions used by the JavaScript APIs of newer HTML5 and CSS3 features, and uses their presence to determine whether the browser natively supports the feature or should receive a polyfill. For example, if the browser contains built-in methods for interacting with the HTML5 **canvas** element, we can assume that that browser supports **canvas**. This is known as "feature detection," and it stands in stark contrast to the more outdated practice of UA (User Agent, or browser) detection. Modernizr also includes, right out of the box, an HTML5 shim similar to the one detailed previously.

Who made it

Modernizr was created by Faruk Ateş, and is actively developed by Paul Irish, Alex Sexton, Ryan Seddon, and Alexander Farkas.

How to use it

Modernizr.com has a builder tool that will allow you to include only the tests that are relevant to your project, as well as a "development" build that contains the entire library of tests. Once you've downloaded a custom build, simply include it as you would any other external script.

Where to get it and learn more

- The Modernizr site (*modernizr.com*)

Selectivizr

Selectivizr allows older versions of Internet Explorer to understand complex CSS3 selectors such as **`:nth-child`** and **`::first-letter`**.

How it works

Selectivizr uses JavaScript to fetch and parse the contents of your style sheet and patch holes where the browser's native CSS parser falls short.

Who made it

Selectivizr was created and is maintained by Keith Clark.

How to use it

Selectivizr must be used with a JavaScript library (I talk about them in the next section). The link to the script goes in an IE conditional comment after the link to the library *.js* file, like so:

```
<script type="text/javascript" src="[JS library]"></script>
<!--[if (gte IE 6)&(lte IE 8)]>
  <script type="text/javascript" src="selectivizr.js"></script>
  <noscript><link rel="stylesheet" href="[fallback css]" /></noscript>
<![endif]-->
```

Potential drawbacks

Because we're forgoing the native CSS parser here, we may see a slight performance hit in applicable browsers.

Where to get it and learn more

- The Selectivizr site (*selectivizr.com*)

Respond.js

Respond.js is a fast and lightweight polyfill that allows older browsers (again, most commonly Internet Explorer 8 and below) to understand **`min-width`** and **`max-width`** media queries, which are commonly used in responsive designs.

How it works

Like Selectivizr, Respond.js looks through style sheets independent of the browser's built-in parser, and upon finding a **`min-width`** or **`max-width`** media query, manually applies those styles to elements on the page through JavaScript, depending on the browser window's width.

Who made it

Respond.js was created by my fellow Filament Group and jQuery Mobile team member Scott Jehl. It was originally developed for use on the responsive BostonGlobe.com site, and was later released as an open source project.

How to use it

Unsurprisingly, one need only download Respond.js and reference it in a `script` tag within the `head` of the document (after the style sheets).

Potential drawbacks:

Again, like Selectivizr, we may see a slight performance hit when using this script, but only in browsers where it ends up being used.

Where to get it and learn more:

- Scott Jehl's Respond page on github (*github.com/scottjehl/Respond*)

JavaScript Libraries

Continuing on the "you don't have to write everything from scratch yourself" theme, it's time to take on JavaScript libraries. A JavaScript library is a collection of prewritten functions and methods that you can use in your scripts to accomplish common tasks or simplify complex ones.

There are many, many JS Libraries out there. Some are large frameworks that include all of the most common polyfills, shortcuts, and widgets you'd ever need to build full-blown Ajax web applications (see the sidebar What Is Ajax?). Some are targeted at specific tasks, such as handling forms, animation, charts, or math functions. For seasoned JavaScript-writing pros, starting with a library is an awesome time-saver. And for folks like you who are just getting started, a library can handle tasks that might be beyond the reach of your own skills.

What Is Ajax?

Ajax (sometimes written AJAX) stands for Asynchronous JavaScript And XML. The "XML" part isn't that important—you don't have to use XML to use Ajax (more on that in a moment). The "asynchronous" part is what matters.

Traditionally, when a user interacted with a web page in a way that required data to be delivered from the server, everything had to stop and wait for the data, and the whole page needed to reload when it was available. This made for a not especially smooth user experience.

But with Ajax, because the page can get data from the server in the background, you can make updates to the page based on user interaction smoothly and in real time. This makes web applications feel more like "real" applications.

You see this on a number of modern websites, although sometimes it's subtle. On Twitter, for example, scrolling to the bottom of a page loads in a set of new tweets. Those aren't hardcoded in the page's markup; they're loaded dynamically as needed. Google's image search uses a similar approach. When you reach the bottom of the current page, you're presented with a button that allows you to load more, but you never navigate away from the current page.

The term "Ajax" was first coined by Jesse James Garrett in an article entitled "Ajax: A New Approach to Web Applications." Ajax is not a single technology, but rather a combination of HTML, CSS, the DOM, and JavaScript, including the `XMLHttpRequest` object that allows data to be transferred asynchronously. Ajax may use XML for data, but it has become more common to use JSON (JavaScript Object Notation), a JavaScript-based and human-readable format, for data exchange.

Writing web applications with Ajax isn't the type of thing you would do right out of the gate, but many of the JavaScript libraries discussed in this chapter have built-in Ajax helpers and methods that let you get started with significantly less effort.

The disadvantage of libraries is that because they generally contain all of their functionality in one big *.js* file, you may end up forcing your users to download a lot of code that never gets used. But the library authors are aware of this and have made many of their libraries modular, and they continue to make efforts to optimize their code. In some cases, it's also possible customize the script and use just the parts you need.

A few libraries you ought to know

Some of the most popular JS libraries as of this writing include:

- **jQuery** (*jquery.com*). jQuery, written in 2005 by John Resig, is by far the most popular JavaScript library today, finding its way onto more than half of the 10,000 most-visited websites. It is free, open source, and employs a syntax that makes it easy to use if you are already handy with CSS, JavaScript, and the DOM. You can supplement jQuery with the jQuery UI library, which adds cool interface elements such as calendar widgets, drag-and-drop functionality, expanding accordion lists, and simple animation effects. I mentioned earlier that I work on jQuery Mobile. That's another jQuery-based library that provides UI elements and polyfills designed to account for the variety of mobile browsers and their notorious quirks.
- **Dojo** (*dojotoolkit.org*). Dojo is an open source, modular toolkit that is particularly helpful for developing web applications with Ajax.
- **Prototype** (*prototypejs.org*). The Prototype JavaScript Framework, written by Sam Stephenson, was developed to add Ajax support to Ruby on Rails.
- **MooTools** (*mootools.net*). MooTools (which stands for My Object-Oriented Tools) is another open source, modular library written by Valerio Proietti.
- **YUI** (*yuilibrary.com*). The Yahoo! User Interface Library is another free, open source JS library for building rich web applications. It is part of The YUI Library project at Yahoo!, founded by Thomas Sha.

As for smaller JS libraries that handle specialized functions, because they are being created and made obsolete all the time, I recommend doing a web search for "JavaScript libraries for _____" and see what is available. Some library categories include:

- Forms
- Animation
- Games
- Information graphics
- Image and 3-D effects in **canvas**

NOTE

For a comparison of over 20 JavaScript libraries and their sizes and features, see the "Comparison of JavaScript frameworks" entry on Wikipedia: en.wikipedia.org/wiki/Comparison_of_JavaScript_frameworks.

The Google Developers site also maintains a list of the more popular open source JavaScript libraries, available here: developers.google.com/speed/libraries/.

- String and math functions
- Database handling

How to use a JS library (jQuery)

It's easy to implement any of the libraries I just listed. All you do is download the JavaScript (*.js*) file, put it on your server, point to it in your **`script`** tag, and you're good to go. It's the *.js* file that does all the heavy lifting, providing prewritten functions and syntax shortcuts. Once you've included it, you can write your own scripts that leverage the features built into the framework. Of course, what you actually do with it is the interesting part (and largely beyond the scope of this chapter, unfortunately).

As a member of the jQuery Mobile team, I have a pretty obvious bias here, so we're going to stick with jQuery in the upcoming examples. Not only is it the most popular library anyway, but they said they'd give me a dollar every time I say "jQuery."

Download the jQuery .js file

To get started with jQuery (*cha-ching*), go to *jQuery.com* and hit the big Download button to get your own copy of *jquery.js*. You have a choice between a "production" version that has all the extra whitespace removed for a smaller file size, or a "development" version that is easier to read but nearly eight times larger in file size. The production version should be just fine if you are not going to edit it yourself.

Copy the code, paste it into a new plain-text document, and save it with the same filename that you see in the address bar in the browser window. As of this writing, the latest version of jQuery is 1.7.2, and the filename of the production version is *jquery-1.7.2.min.js* (the *min* stands for "minimized"). Put the file in the directory with the other files for your site. Some developers keep their scripts in a *js* directory for the sake of organization, or they may simply keep them in the root directory for the site. Wherever you decide put it, be sure to note the pathname to the file because you'll need it in the markup.

Add it to your document

Include the jQuery script the same way you'd include any other script in the document: with a **`script`** element.

```
<script src="pathtoyourjs/jquery-1.7.2.min.js"></script>
```

And that's pretty much it. There is an alternative worth mentioning, however. If you don't want to host the file yourself, you can point to one of the publically hosted versions and use it that way. The jQuery Download page lists a few options, including the following link to the code on Google's server. Simply copy this code exactly as you see it here, paste it into the **`head`**

of the document or before the **</body>** tag, and you've got yourself some jQuery!

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.
min.js"></script>
```

Get "ready"

You don't want to go firing scripts before the document and the DOM are ready for them, do you? Well, jQuery has a statement known as the ready event that checks the document and waits until it's ready to be manipulated. Not all scripts require this (for example, if you were only firing a browser alert), but if you are doing anything with the DOM, it is a good idea to start by setting the stage for your scripts by including this function in your custom **script** or *.js* file:

```
<script src="pathtoyourjs/jquery-1.7.2.min.js"></script>

<script>
$(document).ready(function(){

   // Your code here

});
</script>
```

Scripting with jQuery

Once you're set up, you can begin writing your own scripts using jQuery. The shortcuts jQuery offers break down into two general categories:

- A giant set of built-in feature detection scripts and polyfills
- A shorter, more intuitive syntax for targeting elements (jQuery's selector engine)

You should have a decent sense of what the polyfills do after making your way through that last section, so let's take a look at what the selector engine does for you.

One of the things that jQuery simplifies is moving around through the DOM because you can use the selector syntax that you learned for CSS. Here is an example of getting an element by its **id** value *without* a library:

```
var paragraph = document.getElementById( "status" );
```

The statement finds the element with the ID "status" and saves a reference to the element in a variable (**paragraph**). That's a lot of characters for a simple task. You can probably imagine how things get a little verbose when you're accessing lots of elements on the page. Now that we have jQuery in play, however, we can use this shorthand.

```
var paragraph = $("#status");
```

That's right—that's the **id** selector you know and love from writing CSS. And it doesn't just stop there. *Any* selector you'd use in CSS will work within that special helper function.

You want to find everything with a class of "header"? Use **$(".header");**.

By the element's name? Sure: **$("div");**.

Every subhead in your sidebar? Easy-peasy: **$("#sidebar .sub");**.

You can even target elements based on the value of attributes: **$("[href='http://google.com"]");**.

But it doesn't stop with selectors. We can use a huge number of helper functions built into jQuery and libraries like it to crawl the DOM like so many, uh, Spider-men. Spider-persons. Web-slingers.

jQuery also allows us to chain objects together in a way that can target things even CSS can't (an element's parent element, for example). Let's say we have a paragraph and we want to add a **class** to that paragraph's parent element. We don't necessarily know what that parent element will be, so we're unable to target the parent element directly. In jQuery we can use the **parent()** object to get to it.

```
$("p.error").parent().addClass("error-dialog");
```

Another major benefit is that this is highly readable at a glance: "find any paragraph(s) with the class "error" and add the class "error-dialog" to their parent(s)."

But what if I don't know how to write scripts…?

It takes time to learn JavaScript, and it may be a while before you can write scripts on your own. But not to worry. If you do a web search for what you need (for example, "jQuery image carousel" or "jQuery accordion list"), there is a very good chance you will find lots of scripts that people have created and shared, complete with documentation on how to use them. Because jQuery uses a selector syntax very similar to CSS, it makes it easier to customize jQuery scripts for use with your own markup.

Big Finish

In all of two chapters, we've gone from learning the very basics of variables to manipulating the DOM to leveraging a JavaScript library. Even with all we've covered here, we've just barely begun to cover all the things JavaScript can do.

The next time you're looking at a website and it does something cool, view the source in your browser and have a look around for the JavaScript. You can learn a lot from reading and even taking apart someone else's code.

And remember, there's nothing you can break with JavaScript that can't be undone with a few strokes of the Delete key.

Better still, JavaScript comes with an entire community of passionate developers who are eager to learn and just as eager to teach. Seek out like-minded developers and share the things you've learned along the way. If you're stuck on a tricky problem, don't hesitate to seek out help and ask questions. It's rare that you'll encounter a problem that nobody else has, and the open source developer community is always excited to share the things they've learned. That's why you've had to put up with me for two chapters, as a matter of fact.

Test Yourself

Just a couple of questions for those of you playing along at home.

1. Ajax is a combination of what technologies?

2. What does this do?

```
document.getElementById("main")
```

3. What does this do?

```
document.getElementById("main").getElementsByTagName("section");
```

4. What does this do?

```
document.body.style.backgroundColor = "papayawhip"
```

5. What does this do? This one is a little tricky because it nests functions, but you should be able to piece it together.

```
document
  .getElementById("main")
    .appendChild(
      document.createElement("p")
        .appendChild(
          documentCreateTextNode("Hey, I'm walking here!")
        )
     );
```

6. Match the polyfill with the tasks on the right.

a. HTML5 Shim	1. Add support for `::first-letter`
b. Respond.js	2. Add support for `min-width` and `max-width` media queries
c. Modernizr	3. Add support for `nav` and `aside`
d. Selectivizr	4. Check browser for `canvas` support

7. What is the benefit of using a JavaScript library such as jQuery?
   a. Access to a packaged collection of polyfills
   b. Possibly shorter syntax
   c. Simplified Ajax support
   d. All of the above

ANSWERS

Chapter 1: Where Do I Start?

1. B, D, A, C
2. The W3C guides the development of web-related technologies.
3. C, D, A, E, B
4. Frontend design is concerned with aspects of a site that appear in or are related to the browser. Backend development involves the programming required on the server for site functionality.
5. A web authoring tool provides a visual interface for creating entire web pages, including the necessary HTML, CSS, and scripts. HTML editors provide only shortcuts to writing HTML documents manually.

Chapter 2: How the Web Works

1. c; 2. j; 3. h; 4. g; 5. f; 6. i; 7. b; 8. a; 9. d; 10. e

Chapter 3: Some Big Concepts You Need to Know

1. There are a number of unknown factors when developing a site:
   - What the size of the screen or browser window is
   - What the user's Internet connection speed is
   - Whether the user is at a desk or on the go (context and attention span)
2. 1. c; 2. d; 3. e; 4. a; 5. b
3. Sight impairment: make sure the content is semantic and in logical order for when it is read by a screen reader
   - Hearing impairment: provide transcripts for audio and video content
   - Mobility impairment: use measures that help users without a mouse or keyboard
   - Cognitive impairment: content should be simple and clearly organized
4. You would use a waterfall chart to evaluate your site's performance in the optimization process.

5. Responsive design takes care of the layout, but does not in itself provide alternate content that may be appropriate for the mobile context. Servers are able to detect more features than CSS media queries and can make better decisions about what content to serve.

Chapter 4: Creating a Simple Page (HTML Overview)

1. A tag is part of the markup (brackets and element name) used to delimit an element. An element consists of the content and its tags.
2. The minimal markup of an HTML document is as follows:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf8">
    <title>Title</title>
</head>
<body>
</body>
</html>
```

3. a. *Sunflower.html*—Yes
   b. *index.doc*—No, it must end in .html or *.htm*
   c. *cooking home page.html*—No, there may be no character spaces
   d. *Song_Lyrics.html*—Yes
   e. *games/rubix.html*—No, there may be no slashes in the name
   f. *%whatever.html*—No, there may be no percent symbols
4. All of the following markup examples are incorrect. Describe what is wrong with each one, and then write it correctly.
   a. It is missing the `src` attribute: `<img src="birthday.jpg">`
   b. The slash in the end tag is missing: `<i>Congratulations!</i>`
   c. There should be no attribute in the end tag: `<a href="file.html">linked text</a>`
   d. The slash should be a forward slash: `<p>This is a new paragraph</p>`
5. Make it a comment: `<!-- product list begins here -->`

Chapter 5: Marking Up Text

1.
```
<p>People who know me know that I love to cook.</p>
<hr>
<p>I've created this site to share some of my favorite
       recipes.</p>
```

2. A `blockquote` is a block-level element used for long quotations or quoted material that may consist of other block elements. The `q` (quote) element is for short quotations that go in the flow of text and do not cause line breaks.
3. `pre`

4. The `ul` element is an unordered list for lists that don't need to appear in a particular order. They display with bullets by default. The `ol` element is an ordered list in which sequence matters. The browser automatically inserts numbers for ordered lists.
5. Use a style sheet to remove bullets from an unordered list.
6. `<abbr title="World Wide Web Consortium">W3C</abbr>`
7. A `dl` is the element used to identify an entire description list. The `dt` element is used to identify just one term within that list.
8. The `id` attribute is used to identify a unique element in a document, and the name in its value may appear only once in a document. `class` is used to classify multiple elements into conceptual groups.
9. An `article` element is intended for a self-contained body of content that would be appropriate for syndication or might appear in a different context. A `section` divides content into thematically related chunks.
10.

`&mdash;`	em dash (—)
`&`	ampersand (&)
` `	non-breaking space
`&copy;`	copyright (©)
`&bull;`	bullet (•)
`&trade;`	trademark symbol (™)

Exercise 5-1

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Tapenade Recipe</title>
</head>
<body>

<h1>Tapenade (Olive Spread)</h1>

<p>This is a really simple dish to prepare and it's always a big hit
at parties. My father recommends:</p>

<blockquote><p>"Make this the night before so that the flavors have
time to blend. Just bring it up to room temperature before you serve it.
 In the winter, try serving it warm."</p></blockquote>

<h2>Ingredients</h2>

<ul>
  <li>1 8oz. jar sundried tomatoes</li>
  <li>2 large garlic cloves</li>
  <li>2/3 c. kalamata olives</li>
  <li>1 t. capers</li>
</ul>

<h2>Instructions</h2>
```

```
<ol>
  <li>Combine tomatoes and garlic in a food processor. Blend until
as smooth as possible.</li>

  <li>Add capers and olives. Pulse the motor a few times until they are incorporated, but still retain some
    texture.</li>

  <li>Serve on thin toast rounds with goat cheese and fresh basil garnish (optional).</li>
</ol>

</body>
</html>
```

Exercise 5-2

```
<article>
  <header>
    <p>posted by BGB, <time datetime="2012-11-15" pubdate>November 15,
    2012</time></p>
  </header>
  <h1>Low and Slow</h1>
  <p>This week I am <em>extremely</em> excited about a new cooking technique
    called <dfn><i>sous vide</i></dfn>. In <i>sous vide</i> cooking, you submerge the food (usually vacuum-
    sealed in plastic) into a water bath that is precisely set to the target temperature you want the food
    to be cooked to. In his book, <cite>Cooking for Geeks</cite>, Jeff Potter describes it as <q>ultra-low-
    temperature poaching</q>.</p>
  <p>Next month, we will be serving <b>Sous Vide Salmon with Dill Hollandaise</b>. To reserve a seat at the
    chef table, contact us before November 30.</p>
  <p>blackgoose@example.com<br> 555-336-1800</p>
  <p><small>Warning: Sous vide cooked salmon is not pasteurized. Avoid it if you are pregnant or have
    immunity issues.</small></p>
</article>
```

Exercise 5-3

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Black Goose Bistro: Blog</title>
</head>
<body>
<header>
  <h1>The Black Goose Blog</h1>
  <nav>
    <ul>
      <li>Home</li>
      <li>Menu</li>
      <li>Blog</li>
      <li>Contact</li>
    </ul>
  </nav>
</header>

<article>
  <header>
    <h2>Summer Menu Items</h2>
    <p>posted by BGB, <time datetime="2013-06-15" pubdate>June 15, 2013</time></p>
  </header>
  <p>Our chef has been busy putting together the perfect menu for the
```

```
summer months. Stop by to try these appetizers and main courses while
the days are still long.</p>

  <section id="appetizers">
    <h3>Appetizers</h3>
    <dl>
      <dt>Black bean purses</dt>
      <dd>Spicy black bean and a blend of mexican cheeses wrapped in sheets
of phyllo and baked until golden. <span class="price">$3.95</span></dd>
      <dt class="newitem">Southwestern napoleons with lump crab &mdash;
<strong>new item!</strong><dt>
      <dd>Layers of light lump crab meat, bean and corn salsa, and our
handmade flour tortillas. <span class="price">$7.95</span></dd>
    </dl>
  </section>
  <section id="maincourses">
    <h3>Main courses</h3>
    <dl>
      <dt>Shrimp sate kebabs with peanut sauce</dt>
      <dd>Skewers of shrimp marinated in lemongrass, garlic, and fish sauce then grilled to perfection.
    Served with spicy peanut sauce and jasmine rice. <span class="price">$12.95</span></dd>

      <dt class="newitem">Jerk rotisserie chicken with fried plantains &mdash; <strong>new item!</strong></dt>
      <dd>Tender chicken slow-roasted on the rotisserie, flavored with spicy and fragrant jerk sauce and
    served with fried plantains and fresh mango. <span class="price">$12.95</span></dd>
    </dl>
  </section>
</article>

<article>
  <header>
    <h2>Low and Slow</h2>
    <p>posted by BGB, <time datetime="2012-11-15" pubdate>November 15, 2012</time></p>
  </header>
  <p>This week I am <em>extremely</em> excited about a new cooking technique called <dfn><i>sous vide</i></
    dfn>. In <i>sous vide</i> cooking, you submerge the food (usually vacuum-sealed in plastic) into a water
    bath that is precisely set to the target temperature of the food. In his book, <cite>Cooking for Geeks</
    cite>, Jeff Potter describes it as <q>ultra-low-temperature poaching</q>.</p>
  <p>Next month, we will be serving <b>Sous Vide Salmon with Dill Hollandaise</b>. To reserve a seat at the
    chef table, contact us before November 30.</p>
</article>

<footer>
  <div id="about">
    <p>Location:<br>Baker's Corner, Seekonk, MA</p>
    <p>Hours:<br>Tuesday to Saturday, <time datetime="11:00">11am</time> to <time datetime="00:00">midnight</
    time></p>
  </div>
  <p><small>All content copyright &copy; 2012, Black Goose Bistro and Jennifer Robbins</small><p>
</footer>

</body>
</html>
```

Chapter 6: Adding Links

1. `<a href="tutorial.html">...</a>`
2. `<a href="examples/instructions.html">...</a>`

3. ...
4. ...
5. ...
6. ...
7. ...
8. ...
9. <img src="images/arrow.gif" alt="" >
10. <img src="../images/arrow.gif" alt="" >
11. <img src="../../images/bullet.gif" alt="" >

Exercise 6-1

```
<li><a href="http://www.epicurious.com">Epicurious</a></li>
```

Exercise 6-2

```
<p><a href="index.html">Back to the home page</a></p>
```

Exercise 6-3

```
<li><a href="recipes/tapenade.html">Tapenade (Olive Spread)</a></li>
```

Exercise 6-4

```
<li><a href="recipes/pasta/linguine.html">Linguine with Clam Sauce</a></li>
```

Exercise 6-5

```
<p><a href="../index.html">[Back to the home page]</a></p>
```

Exercise 6-6

```
<p><a href="../../index.html">[Back to the home page]</a></p>
```

Exercise 6-7

1. Go to the Tapenade recipe
2. Try this with Garlic Salmon
3. Try the Linguine with Clam Sauce
4. About Jen's Kitchen
5. Go to AllRecipes.com

Chapter 7: Adding Images

1. The `src` and `alt` attributes are required for the document to be valid. If the `src` attribute is omitted, the browser won't know which image to use. You may leave the value of the `alt` attribute empty if alternative text would be meaningless or clumsy when read in context.
2. `<img src="furry.jpg" alt="">`
3. a) It improves accessibility by providing a description of the image if it is not available or not viewable, and b) because HTML documents are not valid if the `alt` attribute is omitted.
4. It allows the browser to render the rest of the content while the image is being retrieved from the server, which can speed up the display of the page. Leave `width` and `height` attributes out if you are doing a responsive site design where image sizes need to stay flexible.
5. The three likely causes for a missing image are: a) the URL is incorrect, so the browser is looking in the wrong place or for the wrong file name (names are case-sensitive); b) the image file is not in an acceptable format; and c) the image file is not named with the proper suffix (*.gif*, *.jpg*, or *.png*, as appropriate).

Exercise 7-1

In *index.html*:

```
<h2>The Tuscan Countryside</h2>

<p><a href="countryside.html"><img src="thumbnails/countryside_thumb.jpg" alt="view of the rolling tuscan hills"
width="100" height="75"></a> This is …</p>

<h2>Sienna</h2>

<p><a href="sienna.html"><img src="thumbnails/sienna_thumb.jpg" alt="view from the bedroom window" width="75"
height="100"></a> The closest city …</p>
```

In *countryside.html*:

```
<p><img src="photos/countryside.jpg" alt="rolling hills in greens and golds" width="500" height="375"></p>
```

In *sienna.html*:

```
<p><img src="photos/sienna.jpg" alt="a narrow, steep street in Sienna" width="375" height="500"></p>
```

Chapter 8: Basic Table Markup

1. The table itself (`table`), rows (`tr`), header cells (`th`), data cells (`td`), and an optional caption (`caption`).
2. If you want to add additional information about the structure of a table, to specify widths to speed up display, or to add certain style properties to a column of cells.
3. a) The `caption` should be the first element inside the `table` element; b) There can't be text directly in the `table` element; it must go in a `th` or `td`; c) The `th` elements must go inside the `tr` element; d) There is no `colspan` element; this should be a `td` with a `colspan` attribute; e) The second `tr` element is missing a closing tag.

Exercise 8-1

```
<table>
<tr>
  <th>Album</th>
  <th>Year</th>
</tr>
<tr>
  <td>Rubber Soul</td>
  <td>1968</td>
</tr>
<tr>
  <td>Revolver</td>
  <td>1966</td>
</tr>
<tr>
  <td>Sgt. Pepper's</td>
  <td>1967</td>
</tr>
<tr>
  <td>The White Album</td>
  <td>1968</td>
</tr>
<tr>
  <td>Abbey Road</td>
  <td>1969</td>
<tr>
</table>
```

Exercise 8-2

```
<table>
  <tr>
    <th>7:00pm</th><th>7:30pm</th><th>8:00pm</th>
  </tr>
  <tr>
    <td colspan="3">The Sunday Night Movie</td>
  </tr>
  <tr>
    <td>Perry Mason</td>
    <td>Candid Camera</td>
    <td>What's My Line</td>
  </tr>
  <tr>
    <td>Bonanza</td>
    <td colspan="2">The Wackiest Ship in the Army</td>
  </tr>
</table>
```

Exercise 8-3

```
<table>
  <tr>
    <td>apples</td>
    <td rowspan="3">oranges</td>
    <td>pears</td>
  </tr>
  <tr>
    <td>bananas</td>
    <td rowspan="2">pineapple</td>
  </tr>
  <tr>
    <td>lychees</td>
  </tr>
</table>
```

Exercise 8-4

```
<table>
   <caption>Your Content Here</caption>
  <tr>
    <th rowspan="2"> </th>
    <th colspan="2">A common header for two subheads</th>
    <th rowspan="2">Header 3</th>
  </tr>
  <tr>
    <th>Header 1</th>
    <th>Header 2</th>
  </tr>
  <tr>
    <th scope="row">Thing A</th>
    <td>data A1</td>
    <td>data A2</td>
    <td>data A3</td>
  </tr>
  <tr>
    <th scope="row">Thing B </th>
    <td>data B1</td>
    <td>data B2</td>
    <td>data B3</td>
  </tr>
  <tr>
    <th scope="row">Thing C</th>
    <td>data C1</td>
    <td>data C2</td>
    <td>data C3</td>
  </tr>
</table>
```

Chapter 9: Forms

1. a. POST (because of security issues)
   b. POST (because it uses the file selection input type)
   c. GET (because you may want to bookmark search results)
   d. POST (because it is likely to have a length text entry)

2. a. Pull-down menu: `<select>`
   b. Radio buttons: `<input type="radio">`
   c. `<textarea>`
   d. Eight checkboxes: `<input type="checkbox">`
   e. Scrolling menu: `<select multiple="multiple">`
3. Each of these markup examples contains an error. Can you spot what it is?
   a. The `type` attribute is missing.
   b. Checkbox is not an element name; it is a value of the `type` attribute in the `input` element.
   c. The `option` element is not empty. It should contain the value for each option (for example, `option>Orange </option>`).
   d. The required `name` attribute is missing.
   e. The width and height of a text area are specified with the `cols` and `rows` attributes, respectively.

Exercises 9-1 through 9-3: Final source document

```
<!DOCTYPE html >
<html>
<head>
  <meta charset="utf-8" >
   <title>Contest Entry Form</title>
   <style type="text/css">
      ol, ul {
     list-style-type: none;
      }
   </style>
</head>

<body>

<h1>&ldquo;Pimp My Shoes&rdquo; Contest Entry Form</h1>

<p>Want to trade in your old sneakers for a custom pair of Forcefields? Make a case for why your shoes have
    <em>got</em> to go and you may be one of ten lucky winners.</p>


<form action="http://www.learningwebdesign.com/contest.php" method="post">

<fieldset>
<legend>Contest Entry Information</legend>

<ul>
<li><label for="form-name">Name:</label> <input type="text" name="username" id="form-name"></li>
<li><label for="form-email">Email Address:</label> <input type="email" name="emailaddress" id="form-email">
    </li>
<li><label for="form-tel">Telephone Number:</label> <input type="tel" name="telephone" id="form-tel"></li>
<li><label for="form-story">My shoes are SO old...</label><br>
<textarea name="story" rows="4" cols="60" maxlength="300" id="form-story" placeholder="No more than 300
    characters long"></textarea></li>
</ul>
</fieldset>
```

```
<h2>Design your custom Forcefields:</h2>

<fieldset>
<legend>Custom Shoe Design</legend>

<fieldset>
<legend>Color <em>(choose one)</em>:</legend>
  <ul>
  <li><label><input type="radio" name="color" value="red"> Red</label></li>
  <li><label><input type="radio" name="color" value="blue"> Blue</label></li>
  <li><label><input type="radio" name="color" value="black"> Black</label></li>
  <li><label><input type="radio" name="color" value="silver"> Silver</label></li>
  </ul>
</fieldset>

<fieldset>
<legend>Features <em>(Choose as many as you want)</em></legend>
  <ul>
  <li><label><input type="checkbox" name="feature" value="laces"> Sparkley laces</label></li>
  <li><label><input type="checkbox" name="feature" value="logo" checked> Metallic logo</label></li>
  <li><label><input type="checkbox" name="feature" value="heels"> Light-up heels</label></li>
  <li><label><input type="checkbox" name="feature" value="mp3"> MP3-enabled</label></li>
  </ul>
</fieldset>

<fieldset>
<legend>Size</legend>
<label for="form-size"><p>Sizes reflect standard men's sizes:</label>
<select id="form-size" name="size" size="1">
  <option>5</option>
  <option>6</option>
  <option>7</option>
  <option>8</option>
  <option>9</option>
  <option>10</option>
  <option>11</option>
  <option>12</option>
  <option>13</option>
</select>
</p>
</fieldset>

</fieldset>

<p><input type="submit" value="Pimp My Shoes!">
<input type="reset"></p>
</form>
</body>
</html>
```

Chapter 10: What's Up, HTML5?

1. XHTML is defined by and requires the stricter syntax rules of XML. HTML is more forgiving.
2. a. `<h1> … </h1>`

   b. `<img src="image.png" />`

c. `<input type="radio" checked="checked">`

d. `<hr />`

e. `<title>Sifl &amp; Olly</title>`

f.
```
<ul>
   <li>popcorn</li>
   <li>butter</li>
   <li>salt</li>
 </ul>
```

3. A DTD stands for Document Type Definition and is a document that defines all the elements, attributes, and values in a language and their rules for use.
4. HTML5 is unique among HTML specs in that:
   - It includes APIs, not just element and attribute definitions.
   - It includes instructions for how browsers should render elements and handle errors.
   - It does not use a DTD.
   - It can be written in either HTML or XHTML syntax.
5. A global attribute can be used with any HTML element.
6. Web Workers, d; Editing API, e; Geolocation API, a; Web Socket, b; Offline Applications, c
7. Ogg, container; H.264, video; VP8, video; Vorbis, audio; WebM, container; Theora, video; AAC, audio; MPEG-4, container
8. `strokeRect()` and `fill()`

Chapter 11: CSS Orientation

1. selector: `blockquote`; property: `line-height`; value: `1.5`; declaration: `line-height: 1.5`
2. The paragraph text will be gray because when there are conflicting rules of identical weight, the last one listed in the style sheet will be used.
3. a. Use one rule with multiple declarations applied to the `p` element.
```
p {font-family: sans-serif;
   font-size: 1em;
   line-height: 1.2em;}
```

   b. The semicolons are missing.
```
blockquote {
   font-size: 1em;
   line-height: 150%;
   color: gray;
}
```

   c. There should not be curly braces around every declaration, only around the entire declaration block.
```
body {background-color: black;
  color: #666;
  margin-left: 12em;
  margin-right: 12em;}
```

   d. This could be handled with a single rule with a grouped element type selector.

```
p, blockquote, li {color: white;}
```

e. This inline style is missing the property name.

```
<strong style="color: red">Act now!</strong>
```

4. div#intro { color: red; }

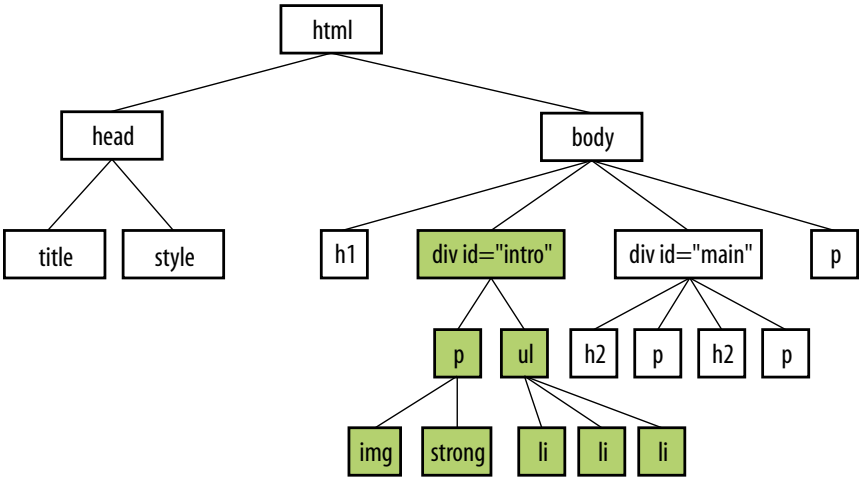


Figure A-1. The highlighted elements would be red as a result of the style rule: `div#intro {color: red;}`.

Exercise 11-1

```
h1 {
  color: red;
  border-bottom: 1px solid red;
}
p {
  font-size: small;
  font-family: sans-serif;
  margin-left: 100px;
}
h2 {
  color: red;
  margin-left: 100px;
}
img {
  float: right;
  margin: 0 12px;
}
```

Chapter 12: Formatting Text

1. a. All text elements in the document: `body {color: red;}`

   b. `h2` elements: `h2 {color: red;}`

   c. `h1` elements and all paragraphs: `h1, p {color: red;}`

   d. Elements belonging to the `class` "special":`.special {color: red;}`

e. All elements in the "intro" section: `#intro {color: red;}`

f. `strong` elements in the "main" section: `#main strong {color: red;}`

g. Extra credit: Just the paragraph that appears after the "main" section (hint: this selector will not work in Internet Explorer 6): `h2 + p {color: red;}`

2. a. ❹, b. ❶, c. ❼, d. ❸, e. ❷, f. ❾, g. ❽, h. ❺, i. ❻

Exercises 12-1 through 12-3

```
<head>
<meta charset="utf-8">
<title>Black Goose Bistro Summer Menu</title>
<link href='http://fonts.googleapis.com/css?family=Marko+One' rel='stylesheet'>
<style>

body {
  font-family: Georgia, serif;
  font-size: 100%;
  line-height: 1.75em;
}
p, dl {
  font-size: .875em;
}
h1 {
  font: bold 1.5em "Marko One", Georgia, serif;
  color: purple;
  text-shadow: .1em .1em .2em lightslategray;
}
h2 {
  font-size: 1em;
  text-transform: uppercase;
  letter-spacing: .5em;
  color: purple;
}
dt {
  font-weight: bold;
  color: sienna;
}
strong {
  font-style: italic;
}
dt strong {
  color: maroon;
}
#info p {
  font-style: italic;
  color: gray;
}
.price {
  font-family: Georgia, serif;
  font-style: italic;
  color: gray;
}
p.warning, sup {
  font-size: small;
  color: red;
}
.label {
```

```
    font-weight: bold;
    font-variant: small-caps;
    font-style: normal;
}
h1, h2, #info {
    text-align: center;
}
h2 + p {
    text-align: center;
    font-style: italic;
}

</style>
</head>
```

Chapter 13: Colors and Backgrounds

1. g. a, b, and c
2. d. rgb(FF, FF, FF)
3. a. –5; b. –1; c. –4; d. –6; e. –2; f. –3
4. a. –1; b. –3; c. –2; d. –6; e. –5; f. –4

Exercise 13-1

```
body {
    …
    background-color: #d2dc9d;
}
#header {
    …
    background-color: rgba(255,255,255,.5);
}
a:link {
    color: #939;
}
a:visited {
    color: #937393;
}
a:focus {
    background-color: #fff;
    color: #c700f2;
}
a:hover {
    background-color: #fff;
    color: #c700f2;
}
a:active {
    background-color: #fff;
    color: #f0f;
}
h1 {
    …
    color: #939;
}
h2 {
```

```
  …
  color: #c60;
}
```

Exercise 13-2

```
body {
  …
  background-color: #d2dc9d;
  background-image: url(images/bullseye.png);
}
```

Exercise 13-3

```
#header {
  …
  background-color: rgba(255,255,255,.5);
  background-image: url(images/purpledot.png);
  background-repeat: repeat-x;
}
```

Exercise 13-4

```
body {
  …
  background-color: #d2dc9d;
  /* background-image: url(images/bullseye.png);
  background-position: center 200px; */
  background-image: url(images/blackgoose.png);
  background-repeat: no-repeat;
  background-position: center 100px;
}
#header {
  …
  background-color: rgba(255,255,255,.5);
  background-image: url(images/purpledot.png);
  background-repeat: repeat-x;
  background-position: center top;
}
```

Exercise 13-5

```
body {
  …
  background-color: #d2dc9d;
  background-image: url(images/blackgoose.png);
  background-repeate: no-repeat;
  background-position: center 100px;
  background-attachment: fixed;
}
```

Exercise 13-6

```
body {
  …
  background: #d2dc9d url(images/blackgoose.png) no-repeat center 100px fixed;
}
```

```
#header {
  …
  background: rgba(255,255,255,.5) url(images/purpledot.png) repeat-x center top;
}
```

Exercise 13-7

```
#header {
  …
  background-image: url(images/purpledot.png) center top repeat-x;}
  background:
   url(images/purpledot.png) left top repeat-y,
   url(images/purpledot.png) right top repeat-y,
   url(images/gooseshadow.png) 90% bottom no-repeat;
  background-color: rgba(255,255,255,.5);
}
```

Exercise 13-8

```
<head>
  …
  <link rel="stylesheet" href="menustyles.css">
</head>
```

Chapter 14: Thinking Inside the Box

a. `border: double black medium;`

b. `overflow: scroll;`

c. `padding: 2em;`

d. `padding: 2em; border: 4px solid red;`

e. `margin: 2em; border: 4px solid red;`

f. `padding: 1em 1em 1em 6em; border: 4px dashed; margin: 1em 6em;`

   or

   `padding: 1em; padding-left: 6em; border: 4px dashed; margin: 1em 6em;`

g. `padding: 1em 50px; border: 2px solid teal; margin: 0 auto;`

Exercise 14-1

```
#products {
  ..
  padding: 1em;
}
#testimonials {
  …
  padding: 1em;
  padding-left: 55px;
}
```

Exercise 14-2

```
#products {
```

```
  …
  padding: 1em;
  border: double #FFBC53;
}
#products h3 {
  …
  border-top: 1px solid;
  border-left: 3px solid;
  padding-left: 1em;
}
#testimonials {
  …
  padding: 1em;
  padding-left: 55px;
  border-radius: 20px;
}
a {
  text-decoration: none;
  border-bottom: 1px dotted;
  padding-bottom: .1em;
}
```

Exercise 14-3

```
body {
  margin: 0;
}
a {
  text-decoration: none;
  border-bottom: 1px dotted;
  padding-bottom: .1em;
}
/* link styles omitted to save space */

/* styles for the intro section */
#intro {
  text-align: center;
  margin: 2em 0 1em;
}
#intro h1 {
  margin-bottom: 0;
}
#intro h2 {
  …;
  margin-top: -10px;
}
#intro p {
  …
  margin: 1em;
}
/* styles for navigation omitted to save space */

/* styles for the products section */
#products {
  …
  padding: 1em;
  border: double #FFBC53;
  margin: 1em;
  }
…
```

```
#products h3 {
  …
  border-top: 1px solid;
  border-left: 3px solid;
  padding-left: 1em;
  margin-top: 2.5em;
}

/* styles for the testimonials box */
#testimonials {
  ….
  padding: 1em;
  padding-left: 55px;
  border-radius: 20px;
  margin: 1em 10%;
  }
/* remaining styles omitted to save space */
```

Chapter 15: Floating and Positioning

1. b is not true. Floats are positioned against the content edge, not the padding edge.
2. c is incorrect. Floats do not use offset properties, so there is no reason to include `right`.
3. Clear the footer `div` to make it start below a floated sidebar: `div#footer { clear: both; }`.
4. a) absolute; b) absolute, fixed; c) fixed; d) relative, absolute, fixed; e) static; f) relative; g) absolute, fixed; h) relative, absolute, fixed; i) relative

Exercise 15-1

```
#products img {
  float: left;
  margin: 0 6px 6px 0;
}
#products .more {
  clear: left;
}
```

Exercise 15-2

```
#nav ul {
  …
  margin: 0 auto;
  width: 19.5em;
}
#nav ul li {
  …
  float: left;
}
#nav ul li a {
  display: block;
  padding: .5em;
  border: 1px solid #ba89a8;
  border-radius: .5em;
  margin: .25em;
}
…
```

```
#nav ul a:focus {
  color:#FC6
  border-color: #fff;
}
#nav ul a:hover {
  color: #fc6;
  border-color: #fff;
}
…
#products {
  …
  clear: both;
}
```

Exercise 15-3

```
#products {
  …
  width: 55%;
  float: left;
}
#products h2 {
  …
  text-align: left;
  }
#testimonials {
  …
  margin: 1em 2% 1em 64%;
}
p#copyright {
  …
  clear: left;
  }
```

Exercise 15-4

```
…
#content {
  position: relative;
}
#testimonials {
  …
  margin: 0 1em;
  position: absolute;
  top: 0;
  right: 0;
  width: 14em;
  }
#products {
  …
  margin: 1em 20.5em 1em 1em;
  clear: both;
}
#award {
  position: absolute;
  top: 35px;
  left: 25px;
}
```

Exercise 15-5

```
…
#award {
  position: fixed;
  top: 35px;
  left: 25px;
}
```

Chapter 16: Page Layout with CSS

1. Fixed, c.; Fluid, a.; Elastic, b.
2. Fixed, b.; Fluid, c.; Elastic, a.
3. Fixed, c.; Fluid, b.; Elastic, a.
4. Fixed, c.; Fluid, a.; Elastic, b.

Exercise 16-1

```
<style>
#wrapper {
  width: 960px;
  margin: 0 auto;
}
#header {
  background-color: #CCC;
  padding: 15px;
}
#links {
  float: right;
  width: 22.5%;
  margin: 0 2.5% 0 0 ;
  outline: 2px dashed #dd0009;
}
#main {
  float: right;
  width: 45%;
  margin: 0 2.5%;
  outline: 2px dashed #0053ae;
}
#news {
  float: right;
  width: 22.5%;
  margin: 0 0 0 2.5% ;
  outline: 2px dashed #009554;
}
#footer {
  clear: right;
  padding: 15px;
  background: #CCC;
}
/* remaining unchanged styles omitted to save space */
</style>

<body>
<div id="wrapper">
  … contents of page here...
</div>
</body>
```

Exercise 16-2

```
#main {
  float: left;
  width: 400px;
  margin-top: 0;
  margin-left: 320px;
  margin-right: 20px;
}

#news {
  float: left;
  width: 300px;
  margin-top: 0;
  margin-left: -740px;
}

#links {
  float: left;
  width: 220px;
  margin: 0;
}
```

Chapter 17: Transitions, Transforms, and Animation

1. Tweening is the process in animation in which frames are generated between two end point states.
2. A transition would have two keyframes, one for the beginning state and one for the end.
3. a) `transition-delay: 0.5s;` b) `transition-timing-function: linear;` c) `transition-duration: .5s;` d) `transition-property: line-height;`
4. c) `text-transform` is not an animatable property.
5. Ease is the default timing function. It starts out slowly, speeds up quickly, and then slows down again at the very end.
6. .2s is the `transition-duration` value.
7. Trick question! They will arrive at the same time, 300ms after the transition begins. The timing function has no effect on the total amount of time it takes.
8. a) `transform: rotate(7deg);` b) `translate(-25px, -50px);` c) `transform-origin: bottom right;` d) `transform: scale(1.2);`
9. The 3 value indicates that the element should be resized three times larger than its original *height*.
10. a) `perspective: 250;` because lower number values are more dramatic.
11. The border is 3 pixels wide at 50% through the animation;
12. a) `animation-direction: reverse;` b) `animation-duration: 5s;` c) `animation-duration: 2s;` d) `animation-iteration-count: 3;`

Exercise 17-1

```
a {
  /* non-transition styles omitted to save space */
  position: relative;
```

```
  -webkit-transition: background-color 0.2s ease-in, border-color 0.2s, top 0.2s, box-shadow 0.2s;
  -moz-transition:  background-color 0.2s, border-color 0.2s, top 0.2s, box-shadow 0.2s;
  -o-transition:  background-color 0.2s, border-color 0.2, top 0.2s, box-shadow 0.2s;
  -ms-transition:  background-color 0.2s, border-color 0.2s, top 0.2s, box-shadow 0.2s;
  transition:  background-color 0.2s, border-color 0.2s, top 0.2s, box-shadow 0.2s;
}
a:hover, a:focus {
  background-color: #fdca00;
  border-color: #fda700;
 }
a:active {
  top: 3px;
  box-shadow: 0 1px 2px rgba(0,0,0,.5);
}
```

Exercise 17-2

Vendor-prefixed properties have been omitted to save space.

```
img {
  width: 200px;
  height: 150px;
  box-shadow: 2px 2px 2px rgba(0,0,0,.4);
  transition: transform .3s ease-in-out;
}
a:hover img {
  box-shadow: 6px 6px 6px rgba(0,0,0,.3);
}
a:hover #img1, a:focus #img1  {
  transform: scale(1.5) rotate(-3deg);
}
a:hover #img2, a:focus #img2  {
  transform: scale(1.5) rotate(5deg);
}
a:hover #img3, a:focus #img3  {
  transform: scale(1.5) rotate(-7deg);
}
a:hover #img4, a:focus #img4  {
  transform: scale(1.5) rotate(2deg);
}
```

Chapter 18: CSS Techniques

1. d) All of the above
2. d) a and c
3. The differences between LESS and Sass include:
   - LESS lacks some of the functionality of Sass.
   - They use a slightly different syntax (`$variable` versus `@variable`).
   - Sass is compiled into standard CSS by a Ruby program on the server; LESS uses JavaScript.
4. e) b and d
5. Give the label elements the same width and float them to the left, then align the text right so it appears next to the control it describes.

6. If you do not set the viewport size, the mobile browser will scale down the page, even if it is designed to be 320 pixels wide.
7. c, e, d, a, b
8. b, e, a, d, c

Exercises 18-1 through 18-3

```
img {
  max-width: 100%;
}

@media screen and (min-width: 481px) {
  #products img {
    float: left;
    margin: 0 6px 6px 0;
  }
  #products .more {
    clear: left;
  }
  #products {
    margin: 1em;
  }
  #testimonials {
    margin: 1em 5%;
    border-radius: 16px;
  }
}

@media screen and (min-width: 780px) {
  #products {
    float: left;
    margin: 0 2% 1em;
    clear: both;
    width: 55%;
    overflow: auto;
  }
  #testimonials {
    margin: 1em 2% 1em 64%;
  }
  p#copyright {
    clear: both;
   }
  #content {
    max-width: 1024px;
    margin: 0 auto;
  }
}
```

Chapter 19: Introduction to JavaScript

1. When you link to an external *.js* file, you can reuse the same scripts for multiple documents. The downside is that it requires an additional HTTP request.
2. a) 1; b) 1two; c) 34; d) 2
3. a) 10; b) 6; c) “2 remaining”; d) “Jennifer is longer.”; e) false

4. It loops through a number of items by starting at the first one in the array and ending when there are no more left.
5. Globally scoped variables may "collide" with variables with the same names in other scripts. It is best to use the `var` keyword in functions to keep your variables scoped locally.
6. a. 2; b. 5; c. 4; d. 3; e. 1

Exercise 19-1

1. `var friends = ["name", "othername", "thirdname", "lastname"];`
2. `alert(friends[2]);`
3. `var name = "yourName";`
4. `if( name === Jennifer) { alert("That's my name too!"); }`
5.
```
var myVariable = #;
 if( myVariable > 5) {
   alert("upper");
 } else {
   alert ("lower");
 }
```

Exercise 19-2

```
<script>
var originalTitle = document.title;
function showUnreadCount( unread ) {
  document.title = originalTitle + " (" + unread + "new message!");
}
showUnreadCount(3);
</script>
```

Chapter 20: Using JavaScript

1. Ajax is a combination of HTML, CSS, and JavaScript (with the `XMLHttpRequest` JavaScript method used to get data in the background).
2. It accesses the element that has the **`id`** value "main".
3. It creates a **`nodeList`** of all the section elements in the element with the **`id`** of "main".
4. It sets the background color of the page (**`body`** element) to "papayawhip".
5. It creates a new text node that says, "Hey, I'm walking here!", inserts it in a newly created `p` element, and puts the new `p` element in the element with the `id` "main".
6. a. 3; b. 2; c. 4; d. 1
7. All of the above.

Chapter 21: Web Graphics Basics

1. You can get a license to have exclusive rights to an image so that your competitor doesn't use the same photo on their site.
2. ppi stands for "pixels per inch" and is a measure of resolution.
3. Indexed color is a mode for storing color information in an image that stores each pixel color in a color table. GIF and 8-bit PNG formats are indexed color images.
4. There are 256 colors in an 8-bit graphic and 32 colors in a 5-bit graphic.
5. GIF can contain animation and transparency. JPEG cannot.
6. GIF can contain animation. PNGs cannot.
7. PNGs can have multiple levels of transparency. GIF has only binary (on/off) transparency.
8. Lossy compression is cumulative, which means you lose image data every time you save an image as a JPEG. If you open a JPEG and save it as a JPEG again, even more image information is thrown out than the first time you saved it. Be sure to keep your full-quality original and save JPEG copies as needed.
9. In binary transparency, a pixel is either entirely transparent or entirely opaque. Alpha transparency allows up to 256 levels of transparency.
10. Ⓐ GIF or PNG-8 because it is text, flat colors, and hard edges. Ⓑ JPEG because it is a photograph. Ⓒ GIF or PNG-8 because although it has some photographic areas, most of the image is flat colors with hard edges. Ⓓ GIF or PNG-8 because it is a flat graphical image. Ⓔ JPEG because it is a photograph.

Chapter 22: Lean and Mean Web Graphics

1. Smaller graphic files means shorter download and display times. Every second counts toward creating a favorable user experience of your site.
2. Dithering introduces a speckle pattern that interrupts strings of identical pixels, and therefore the GIF compression scheme can't compress areas with dithering as efficiently as flat colors.
3. The fewer pixel colors in the image, the smaller the resulting GIF, both because the image can be stored at a lower bit depth and because there are more areas of similar color for the GIF to compress.
4. The Quality (compression) setting is the most effective tool for controlling the size of a JPEG.
5. JPEG compression works effectively on smooth or blurred areas, so introducing a slight blur allows the JPEG compression to work more efficiently, resulting in smaller files.
6. Just as you would do for an indexed GIF, optimize a PNG-8 by designing with flat colors, reducing the number of colors, and avoiding dithering. There are no strategies for optimizing a PNG-24 because they are designed to store images with lossless compression.