HTML MARKUP FOR STRUCTURE

PART II

IN THIS PART

CREATING A SIMPLE PAGE
(HTML Overview)

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

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.

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.

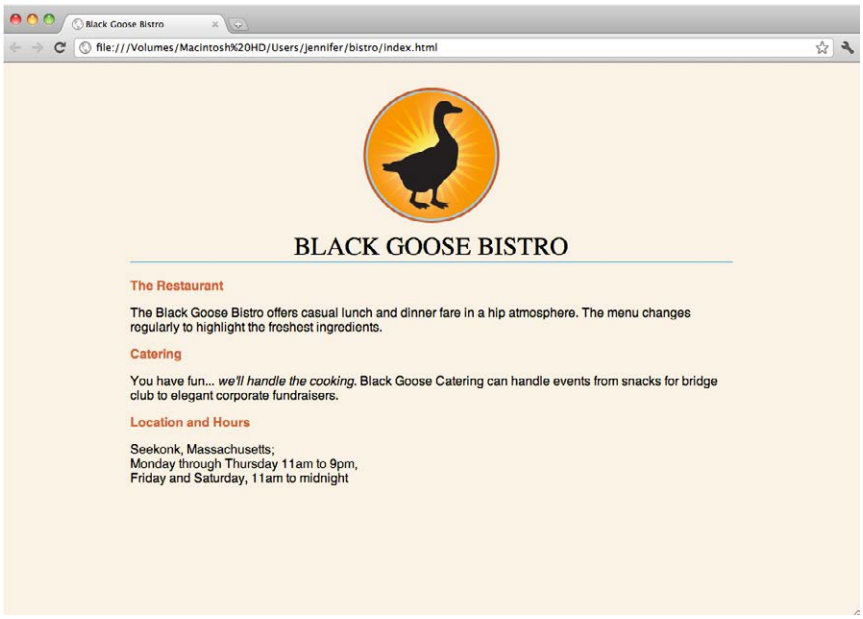


Figure 4-1. *In this chapter, we'll write the source document for this page step by step.*

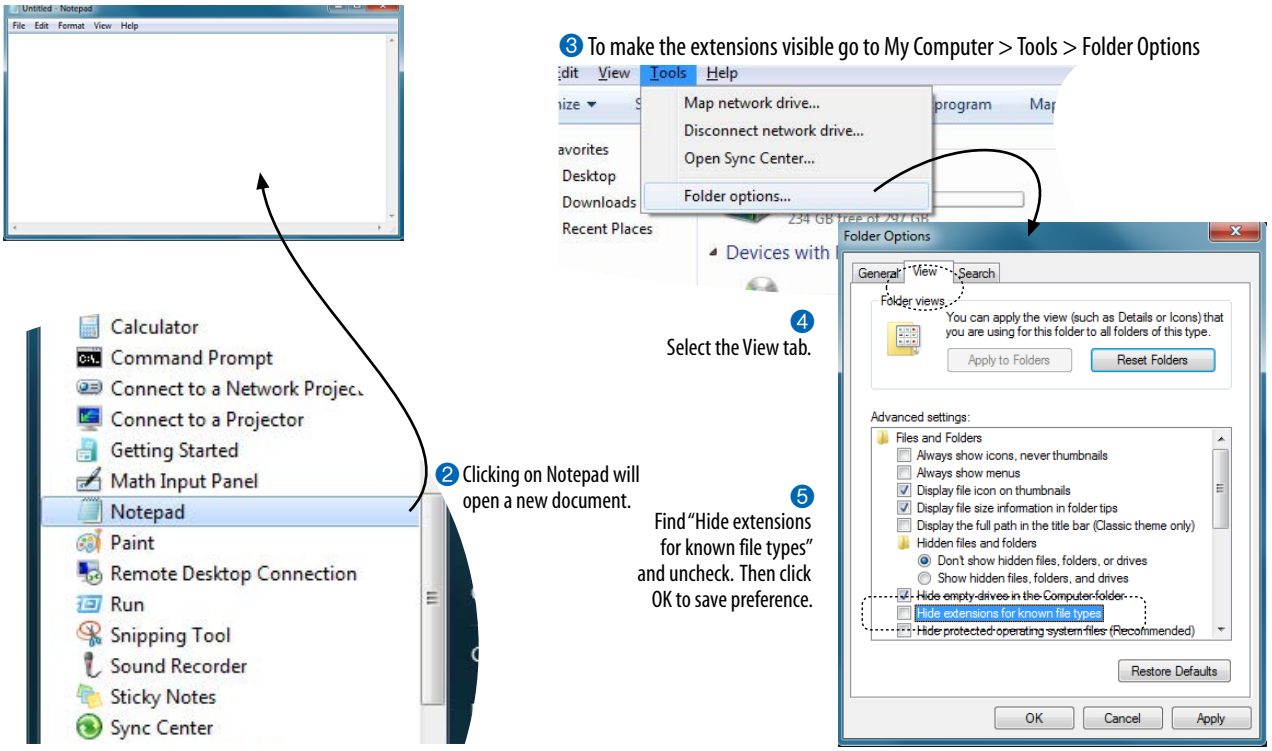
Creating a new document in Notepad (Windows)

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.

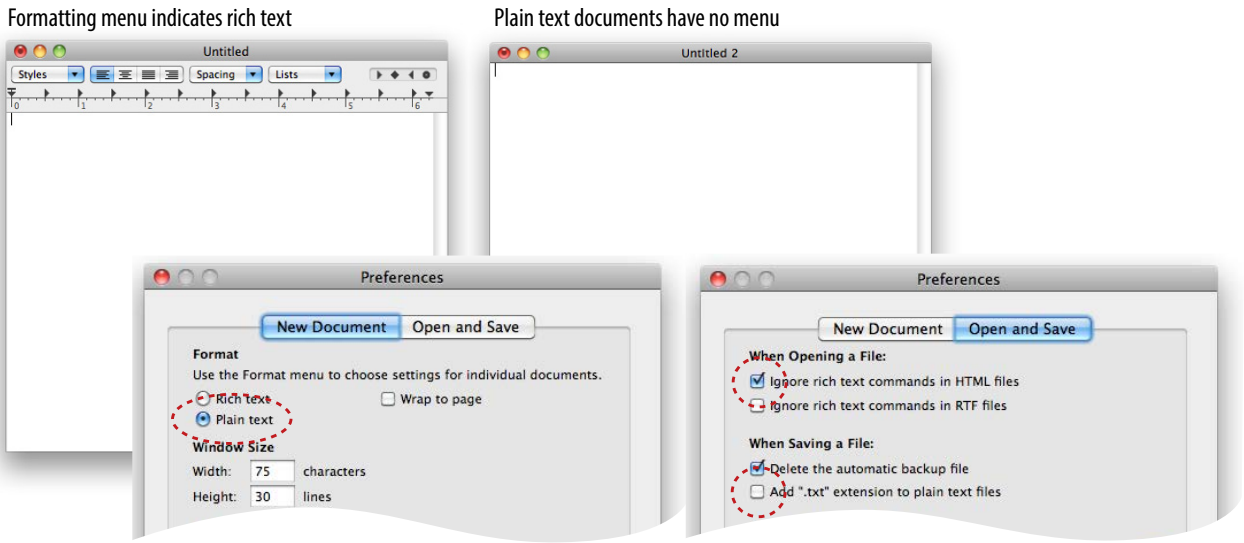


Figure 4-3. Launching TextEdit and choosing Plain Text settings in the Preferences.

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.

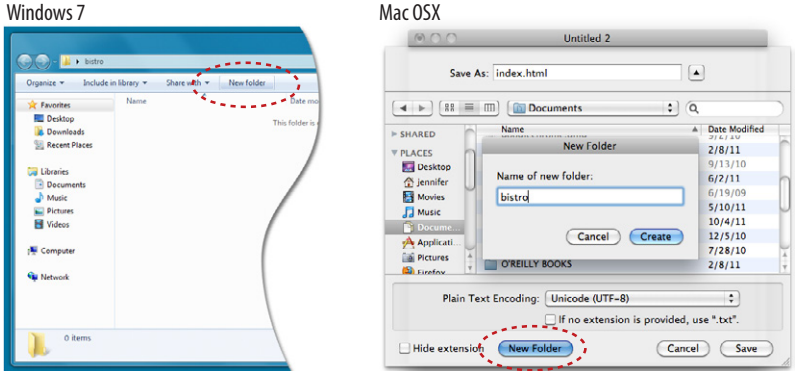


Figure 4-4. Saving index.html in a new folder called "bistro".

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

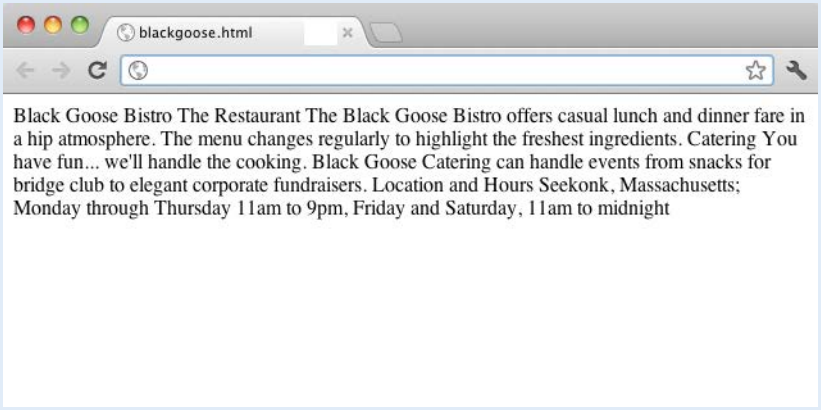


Figure 4-5. *A first look at the content in a browser.*

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.

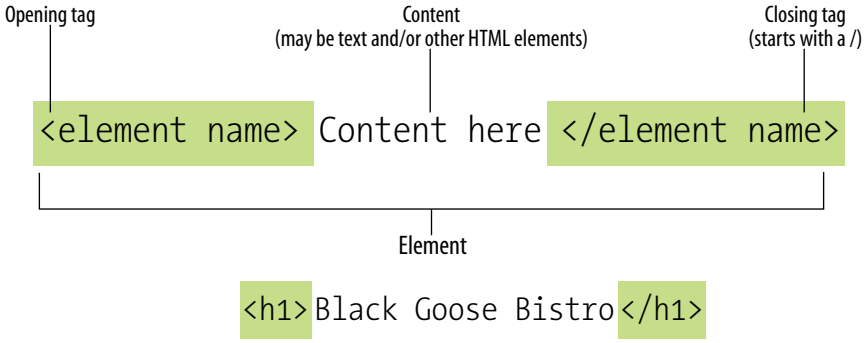


Figure 4-6. *The parts of an HTML container element.*

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.

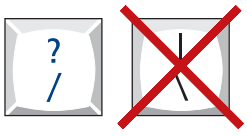
One last thing...capitalization. In HTML, the capitalization of element names is not important. So **<img>**, **<Img>**, and **<IMG>** are all the same as far as the browser is concerned. However, in XHTML (the stricter version of HTML) all element names must be all lowercase in order to be valid. Many 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.

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.

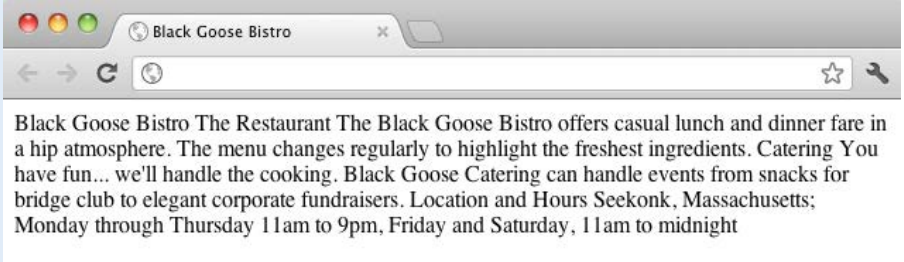


Figure 4-8. The page in a browser after the document structure elements have been defined.

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.

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.

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.

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.

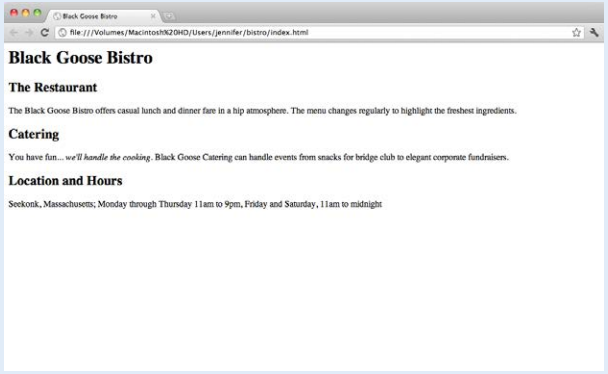


Figure 4-9. *The home page after the content has been marked up with HTML elements.*

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.

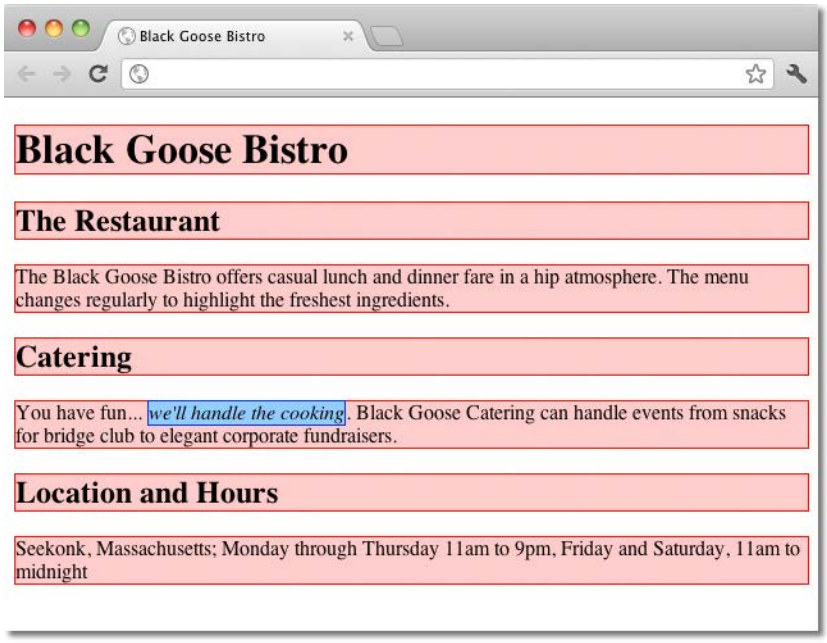


Figure 4-10. The outlines show the structure of the elements in the home page.

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.

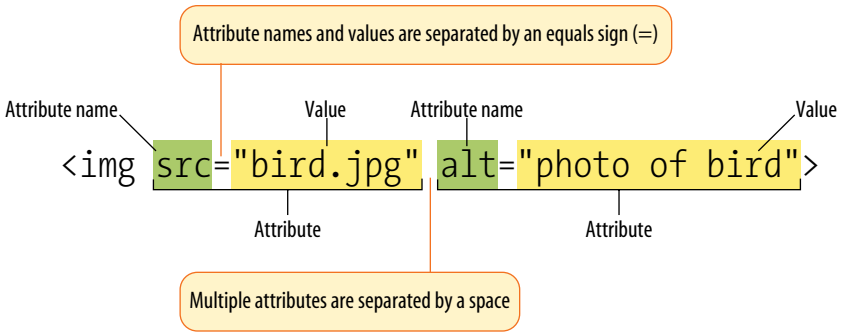


Figure 4-12. An `img` element with attributes.

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.

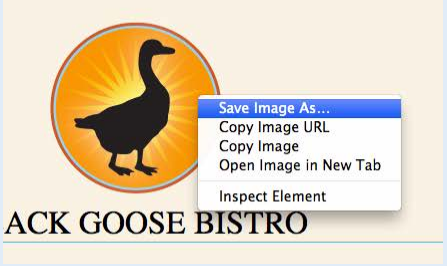
exercise 4-4 | Adding an image

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.

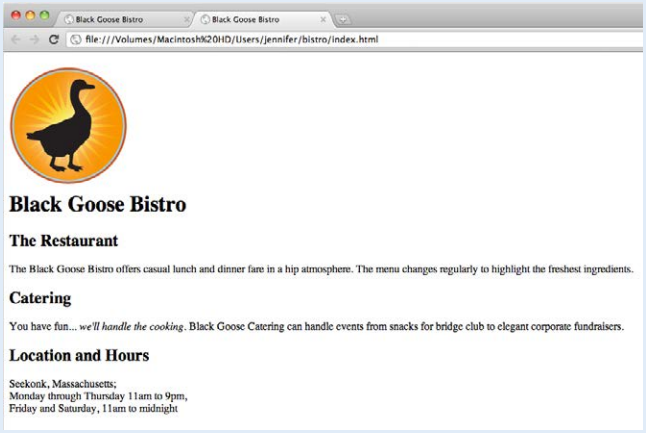


Figure 4-14. The Black Goose Bistro page with the logo image.

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

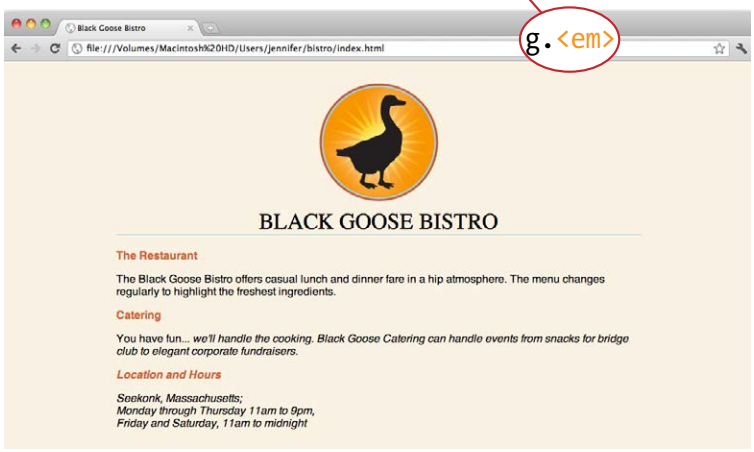
Omitting the slash in the closing tag (or even omitting the closing tag itself) for block elements, such as headings or paragraphs, may not be so dramatic. Browsers interpret the start of a new block element to mean that the previous block element is finished.

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>
```

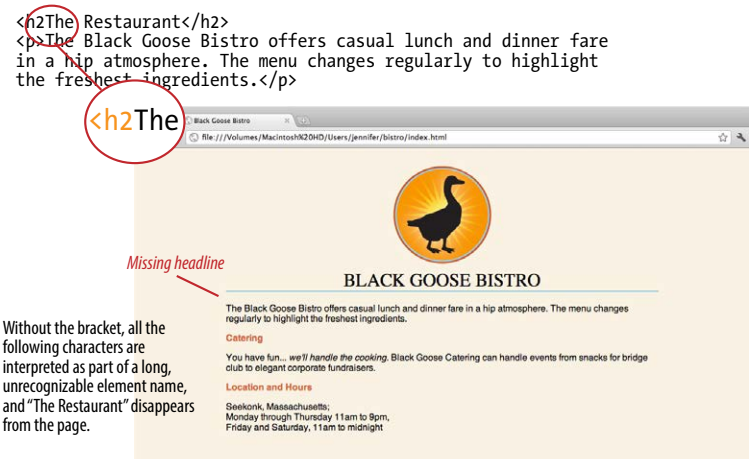


Browsers don't display any text within a tag, so my heading disappeared. The browser just ignored the foreign-looking element name and moved on to the next element.

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.

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.

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).

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

MARKING UP TEXT

Once your content is ready to go (you proofread it, right?) and you've added the markup to structure the document (**`html`**, **`head`**, **`title`**, and **`body`**), you are ready to identify the elements in the content. This chapter introduces the elements you have to choose from for marking up text content. There probably aren't as many of them as you might think, and really just a handful that you'll use with regularity. That said, this chapter is a big one and covers a lot of ground.

As we begin our tour of elements, I want to reiterate how important it is to choose elements semantically, that is, in a way that most accurately describes the content's meaning. If you don't like how it looks, change it with a style sheet. A semantically marked up document ensures your content is available and accessible in the widest range of browsing environments, from desktop computers and mobile devices to assistive screen readers. It also allows non-human readers, such as search engine indexing programs, to correctly parse your content and make decisions about the relative importance of elements on the page.

With these principles in mind, it is time to meet the HTML text elements, starting with the most basic element of them all, the humble paragraph.

IMPORTANT NOTE

I will be teaching markup according to the HTML5 standard maintained by the W3C (www.w3.org/TR/html5/) as it appeared as of this writing in mid-2012. There is another "living" (therefore unnumbered) version of HTML maintained by the WHATWG (whatwg.org) that is nearly the same, but usually has some differences. I will be sure to point out elements and attributes that belong to only one spec. Both specs are changing frequently, so I urge you to check online to see whether elements have been added or dropped.

You may have heard that not all browsers support HTML5. That is true. But the vast majority of the elements in HTML5 have been around for decades in earlier HTML versions, so they are supported universally. Elements that are new in HTML5 and may not be well supported will be indicated with this marker: NEW IN HTML5 *. So, unless I explicitly point out a support issue, you can assume that the markup descriptions and examples presented here will work in all browsers.*

IN THIS CHAPTER

Paragraphs

`<p>...</p>`

A paragraph element

Paragraphs are the most rudimentary elements of a text document. You indicate a paragraph with the **p** element by inserting an opening ** ** tag at the beginning of the paragraph and a closing ** ** tag after it, as shown in this example.

```
<p>Serif typefaces have small slabs at the ends of letter strokes. In
general, serif fonts can make large amounts of text easier to read.</p>

<p>Sans-serif fonts do not have serif slabs; their strokes are square
on the end. Helvetica and Arial are examples of sans-serif fonts.
In general, sans-serif fonts appear sleeker and more modern.</p>
```

Visual browsers nearly always display paragraphs on new lines with a bit of space between them by default (to use a term from CSS, they are displayed as a block). Paragraphs may contain text, images, and other inline elements (called phrasing content in the spec), but they may *not* contain headings, lists, sectioning elements, or any element that typically displays as a block by default.

NOTE

You must assign an element to all the text in a document. In other words, all text must be enclosed in some sort of element. Text that is not contained within tags is called "naked" or "anonymous" text, and it will cause a document to be invalid. For more information about checking documents for validity, see Chapter 4, Creating a Simple Page (HTML Overview).

In HTML, it is OK to omit the closing ** ** tag. A browser just assumes it is closed when it encounters the next block element. However, in the stricter XHTML syntax, the closing tag is required (no surprise there). Many web developers, including myself, prefer to close paragraphs and all elements, even in HTML, for the sake of consistency and clarity. I recommend folks who are just learning markup, like yourself, do the same.

Headings

`<h1>...</h1>`
`<h2>...</h2>`
`<h3>...</h3>`
`<h4>...</h4>`
`<h5>...</h5>`
`<h6>...</h6>`

Heading elements

In the last chapter, we used the **h1** and **h2** elements to indicate headings for the Black Goose Bistro page. There are actually six levels of headings, from **h1** to **h6**. When you add headings to content, the browser uses them to create a document outline for the page. Assistive reading devices such as screen readers use the document outline to help users quickly scan and navigate through a page. In addition, search engines look at heading levels as part of their algorithms (information in higher heading levels may be given more weight). For these reasons, it is a best practice to start with the Level 1 heading (**h1**) and work down in numerical order (see note), creating a logical document structure and outline.

NOTE

HTML5 has a new outlining system that looks beyond headings to generate the outline. See the sidebar Sectioning Content later in this chapter for details.

This example shows the markup for four heading levels. Additional heading levels would be marked up in a similar manner.

```
<h1>Type Design</h1>

<h2>Serif Typefaces</h2>
<p>Serif typefaces have small slabs at the ends of letter strokes.
In general, serif fonts can make large amounts of text easier to
read.</p>
```

```
<h3>Baskerville</h3>

<h4>Description</h4>
<p>Description of the Baskerville typeface.</p>

<h4>History</h4>
<p>The history of the Baskerville typeface.</p>

<h3>Georgia</h3>
<p>Description and history of the Georgia typeface.</p>

<h2>Sans-serif Typefaces</h2>
<p>Sans-serif typefaces do not have slabs at the ends of strokes.</p>
```

The markup in this example would create the following document outline:

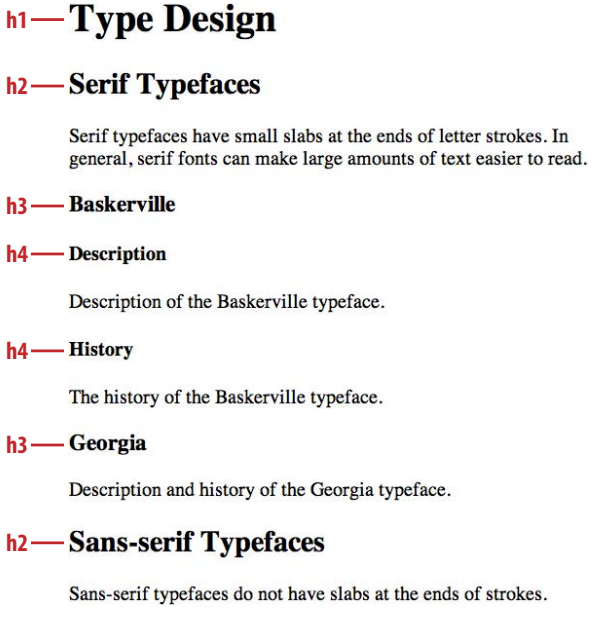
1. Type Design
    1. Serif Typefaces
       + text paragraph
        1. Baskerville
            1. Description
               + text paragraph
            2. History
               + text paragraph
        2. Georgia
           + text paragraph
    2. Sans-Serif Typefaces
       + text paragraph

By default, the headings in our example will be displayed in bold text, starting in very large type for `h1`s, with each consecutive level in smaller text, as shown in Figure 5-1. You can use a style sheet to change their appearance.

NOTE

All screenshots in this book were taken using the Chrome browser on a Mac unless otherwise noted.

Figure 5-1. The default rendering of four heading levels.



Indicating a Shift in Themes

`<hr>`

A horizontal rule

If you want to indicate that one topic or thought has completed and another one is beginning, you can insert what is called in HTML5 a "paragraph-level thematic break" using the **hr** element. It is used as a logical divider between sections of a page or paragraphs of text. The **hr** element adds a logical divider between sections or paragraphs of text without introducing a new heading level.

In HTML versions prior to HTML5, **hr** was defined as a "horizontal rule" because it inserted a horizontal line on the page. Browsers still render **hr** as a 3D shaded rule and put it on a line by itself with some space above and below by default, but it now has a new semantic purpose. If a decorative line is all you're after, it is better to create a rule by specifying a colored border before or after an element with CSS.

hr is an empty element—you just drop it into place where you want the thematic break to occur, as shown in this example and Figure 5-2. Note that in XHTML, the **hr** element must be closed with a slash: **<hr />**.

```
<h3>Times</h3>
<p>Description and history of the Times typeface.</p>
<hr>
<h3>Georgia</h3>
<p>Description and history of the Georgia typeface.</p>
```

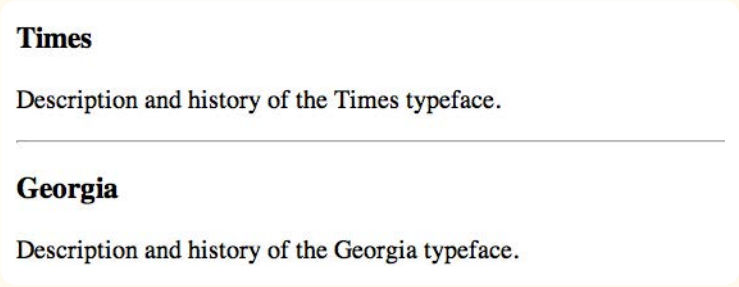


Figure 5-2. The default rendering of a horizontal rule.

Heading groups

`<hgroup>...</hgroup>`

A group of stacked headings

NEW IN HTML5

It is common for headlines to have clarifying subheads or taglines. Take, for example, the title of Chapter 4 in this book:

Creating a Simple Page
(HTML Overview)

In the past, marking stacks of headings and subheadings was somewhat problematic. The first line, "Creating a Simple Page," is clearly an **h1**, but if you make the second line an **h2**, you may introduce an unintended new level to the document outline. The best you could do was mark it as a paragraph, but that didn't exactly make semantic sense.

For this reason, HTML5 includes the **hgroup** element for identifying a stack of headings as a group.* Browsers that support **hgroup** know to count only the highest-ranked heading in the outline and ignore the rest. Here is how the **hgroup** element could be used to mark up the title of Chapter 4. With this markup, only the **h1**, "Creating a Simple Page," would be represented in the document outline.

```
<hgroup>
  <h1>Creating a Simple Page</h1>
  <h2>(HTML Overview)</h2>
</hgroup>
```

SUPPORT ALERT

The `hgroup` *element is not supported in Internet Explorer versions 8 and earlier (see the sidebar HTML5 Support in Internet Explorer later in this chapter for a workaround). Older versions of Firefox and Safari (prior to 3.6 and 4, respectively) do not support it according to the spec, but they don't ignore it completely, so you can apply styles to it.*

Lists

Humans are natural list makers, and HTML provides elements for marking up three types of lists:

- **Unordered lists.** Collections of items that appear in no particular order.
- **Ordered lists.** Lists in which the sequence of the items is important.
- **Description lists.** Lists that consist of name and value pairs, including but not limited to terms and definitions.

All list elements—the lists themselves and the items that go in them—are displayed as block elements by default, which means that they start on a new line and have some space above and below, but that may be altered with CSS. In this section, we'll look at each list type in detail.

Unordered lists

`<ul>...</ul>`
Unordered list

`<li>...</li>`
List item within an unordered list

Just about any list of examples, names, components, thoughts, or options qualify as unordered lists. In fact, most lists fall into this category. By default, unordered lists display with a bullet before each list item, but you can change that with a style sheet, as you'll see in a moment.

To identify an unordered list, mark it up as a **ul** element. The opening **
** tag goes before the first list item, and the closing tag **
** goes after the last item. Then, each item in the list gets marked up as a list item (**li**) by enclosing it in opening and closing **li** tags, as shown in this example. Notice that there are no bullets in the source document. They are added automatically by the browser (Figure 5-3).

```
<ul>
  <li><a href="">Serif</a></li>
  <li><a href="">Sans-serif</a></li>
  <li><a href="">Script</a></li>
  <li><a href="">Display</a></li>
  <li><a href="">Dingbats</a></li>
</ul>
```

NOTE

The only thing that is permitted within an unordered list (that is, between the start and end `ul` *tags) is one or more list items. You can't put other elements in there, and there may not be any untagged text. However, you can put any type of flow element within a list item (*`li`*).*

* Although potentially useful, the future of the **hgroup** element is uncertain. If you are interested in using it for a published site, you should check the HTML5 specification first.

- Serif
- Sans-serif
- Script
- Display
- Dingbats

Figure 5-3. The default rendering of the sample unordered list. The bullets are added automatically by the browser.

Nesting Lists

Any list can be nested within another list; it just has to be placed within a list item. This example shows the structure of an unordered list nested in the second ordered list item.

```
<ol>
  <li></li>
  <li>
    <ul>
      <li></li>
      <li></li>
      <li></li>
    </ul>
  </li>
</ol>
```

When you nest an unordered list within another unordered list, the browser automatically changes the bullet style for the second-level list. Unfortunately, the numbering style is not changed by default when you nest ordered lists. You need to set the numbering styles yourself using style sheets.

But here's the cool part. We can take that same unordered list markup and radically change its appearance by applying different style sheets, as shown in Figure 5-4. In the figure, I've turned off the bullets, added bullets of my own, made the items line up horizontally, even made them look like graphical buttons. The markup stays exactly the same.

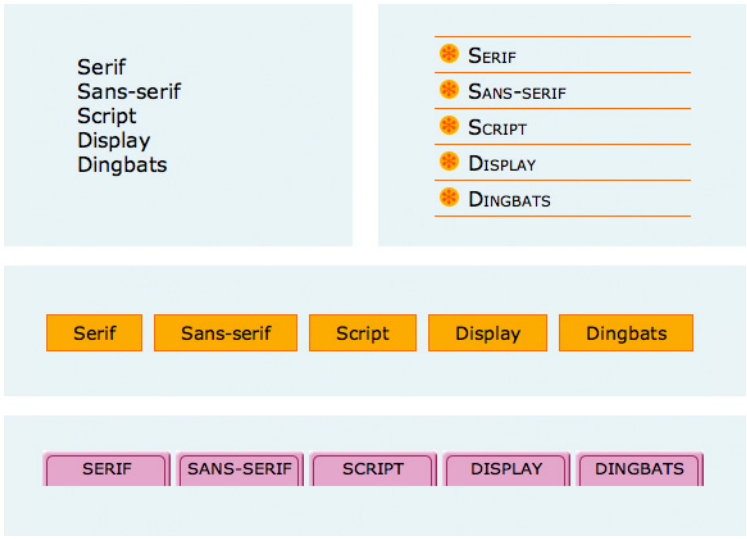


Figure 5-4. With style sheets, you can give the same unordered list many different looks.

Ordered lists

`<ol>...</ol>`
Ordered list

`<li>...</li>`
List item within an ordered list

Ordered lists are for items that occur in a particular order, such as step-by-step instructions or driving directions. They work just like the unordered lists described earlier, except they are defined with the `ol` element (for ordered list, of course). Instead of bullets, the browser automatically inserts numbers before ordered list items, so you don't need to number them in the source document. This makes it easy to rearrange list items without renumbering them.

Ordered list elements must contain one or more list item elements, as shown in this example and in Figure 5-5:

```
<ol>
  <li>Gutenburg develops moveable type (1450s)</li>
  <li>Linotype is introduced (1890s)</li>
  <li>Photocomposition catches on (1950s)</li>
  <li>Type goes digital (1980s)</li>
</ol>
```

1. Gutenburg develops moveable type (1450s)
2. Linotype is introduced (1890s)
3. Photocomposition catches on (1950s)
4. Type goes digital (1980s)

Figure 5-5. The default rendering of an ordered list. The numbers are added automatically by the browser.

If you want a numbered list to start at a number other than "1," you can use the **start** attribute in the **ol** element to specify another starting number, as shown here:

```
<ol start="17">
  <li>Highlight the text with the text tool.</li>
  <li>Select the Character tab.</li>
  <li>Choose a typeface from the pop-up menu.</li>
</ol>
```

The resulting list items would be numbered 17, 18, and 19, consecutively.

Changing Bullets and Numbering

You can use the **list-style-type** style sheet property to change the bullets and numbers for lists. For example, for unordered lists, you can change the shape from the default dot to a square or an open circle, substitute your own image, or remove the bullet altogether. For ordered lists, you can change the numbers to roman numerals (I., II., III. or i., ii., iii.), letters (A., B., C., or a., b., c.), and several other numbering schemes. In fact, as long as the list is marked up semantically, it doesn't need to display with bullets or numbering at all. Changing the style of lists with CSS is covered in Chapter 18, CSS Techniques.

Description lists

Description lists are used for any type of name/value pairs, such as terms and their definitions, questions and answers, or other types of terms and their associated information. Their structure is a bit different from the other two lists that we just discussed. The whole description list is marked up as a **dl** element. The content of a **dl** is some number of **dt** elements indicating the names and **dd** elements for their respective values. I find it helpful to think of them as "terms" (to remember the "t" in **dt**) and "definitions" (for the "d" in **dd**), even though that is only one use of description lists in HTML5.

`<dl>...</dl>`
A description list

`<dt>...</dt>`
A name, such as a term or label

`<dd>...</dd>`
A value, such as a description or definition

Here is an example of a list that associates forms of typesetting with their descriptions (Figure 5-6).

```
<dl>
  <dt>Linotype</dt>
  <dd>Line-casting allowed type to be selected, used, then recirculated
into the machine automatically. This advance increased the speed of
typesetting and printing dramatically.</dd>

  <dt>Photocomposition</dt>
  <dd>Typefaces are stored on film then projected onto photo-sensitive
paper. Lenses adjust the size of the type.</dd>

  <dt>Digital type</dt>
  <dd><p>Digital typefaces store the outline of the font shape in a
format such as Postscript. The outline may be scaled to any size for
output.</p>
      <p>Postscript emerged as a standard due to its support of
```

```
graphics and its early support on the Macintosh computer and Apple
laser printer.</p>
  </dd>
</dl>
```

Linotype
: Line-casting allowed type to be selected, used, then recirculated into the machine automatically. This advance increased the speed of typesetting and printing dramatically.

Photocomposition
: Typefaces are stored on film then projected onto photo-sensitive paper. Lenses adjust the size of the type.

Digital type
: Digital typefaces store the outline of the font shape in a format such as Postscript. The outline may may be scaled to any size for output.

: Postscript emerged as a standard due to its support of graphics and its early support on the Macintosh computer and Apple laser printer.

Figure 5-6. The default rendering of a definition list. Definitions are set off from the terms by an indent.

The **dl** element is only allowed to contain **dt** and **dd** elements. It is OK to have multiple definitions with one term and vice versa. You cannot put headings or content-grouping elements (like paragraphs) in names (**dt**), but the value (**dd**) can contain any type of flow content.

Sectioning Roots

The **blockquote** is in a category of elements called sectioning roots. Headings in a sectioning root element will not be included in the main document outline. That means you can have a complex heading hierarchy within a **blockquote** without worrying how it will affect the overall structure of the document. Other sectioning root elements include **figure**, **details**, **fieldset** (for organizing form fields), **td** (a table cell), and **body** (because it has its own outline, which also happens be the outline of the document).

More Content Elements

We've covered paragraphs, headings, and lists, but there are a few more special text elements to add to your HTML toolbox that don't fit into a neat category: long quotations (**blockquote**), preformatted text (**pre**), and figures (**figure** and **figcaption**). One thing these elements do have in common is that they are considered "grouping content" in the HTML5 spec (along with **p**, **hr**, the list elements, and the generic **div**, covered later in this chapter). The other thing they share is that browsers typically display them as block elements by default.

Long quotations

`<blockquote>...</blockquote>`

A lengthy, block-level quotation

If you have a long quotation, a testimonial, or a section of copy from another source, you should mark it up as a **blockquote** element. It is recommended that content within **blockquote** elements be contained in other elements, such as paragraphs, headings, or lists, as shown in this example (see the sidebar Sectioning Roots).

```
<p>Renowned type designer, Matthew Carter, has this to say about his
profession:</p>

<blockquote>
  <p>Our alphabet hasn't changed in eons; there isn't much latitude in
what a designer can do with the individual letters.</p>

  <p>Much like a piece of classical music, the score is written
down - it's not something that is tampered with - and yet, each
conductor interprets that score differently. There is tension in
the interpretation.</p>
</blockquote>
```

NOTE

There is also the inline element, `q`*, for short quotations in the flow of text. We'll talk about it later in this chapter.*

Figure 5-7 shows the default rendering of the **`blockquote`** example. This can be altered with CSS.

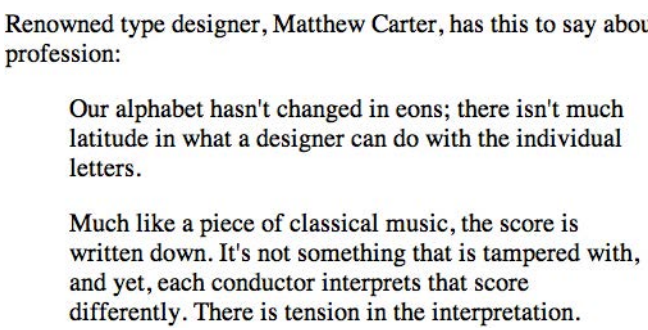


Figure 5-7. *The default rendering of a* `blockquote` *element.*

Preformatted text

`<pre>...</pre>`

Preformatted text

In the previous chapter, you learned that browsers ignore whitespace such as line returns and character spaces in the source document. But in some types of information, such as code examples or poetry, the whitespace is important for conveying meaning. For these purposes, there is the preformatted text (**`pre`**) element. It is a unique element in that it is displayed exactly as it is typed—including all the carriage returns and multiple character spaces. By default, preformatted text is also displayed in a constant-width font (one in which all the characters are the same width, also called monospace), such as Courier.

NOTE

The `white-space:pre` *CSS property can also be used to preserve spaces and returns in the source. Unlike the* `pre` *element, text formatted with the* `white-space` *property is not displayed in a constant-width font.*

The **`pre`** element in this example displays as shown in Figure 5-8. The second part of the figure shows the same content marked up as a paragraph (**`p`**) element for comparison.

```
<pre>
This is                an             example of
      text with a          lot of
                           curious
                           whitespace.
</pre>
```

```
<p>
This is                an             example of
      text with a          lot of
                           curious
                           whitespace.
</p>
```

```
This is                an             example of
      text with a          lot of
                           curious
                           white space.
```

This is an example of text with a lot of curious white space.

Figure 5-8. Preformatted text is unique in that the browser displays the whitespace exactly as it is typed into the source document. Compare it to the paragraph element, in which line returns and character spaces are reduced to a single space.

Figures

`<figure>...</figure>`

Contact information

NEW IN HTML5

`<figcaption>...</figcaption>`

Contact information

NEW IN HTML5

The **figure** element is used for content that illustrates or supports some point in the text. A figure may contain an image, a video, a code snippet, text, or even a table—pretty much anything that can go in the flow of web content—and should be treated and referenced as a self-contained unit. That means if a figure is removed from its original placement in the main flow (to a sidebar or appendix, for example), both the figure and the main flow should continue to make sense.

Although it is possible to simply drop an image into text, wrapping it in **figure** tags makes its purpose explicitly clear. It also allows you to apply special styles to figures but not to other images on the page.

```
<figure>
   <img src="piechart.png" alt="chart showing fonts on mobile devices">
</figure>
```

A caption can be attached to the figure using the optional **figcaption** element above or below the figure content.

```
<figure>
   <pre><code>
    body {
      background-color: #000;
      color: red;
    }
    </code></pre>
   <figcaption>
         Sample CSS rule.
   </figcaption>
</figure>
```

In Exercise 5-1, you'll get a chance to mark up a document yourself and try out the basic text elements we've covered so far.

SUPPORT ALERT

The `figure` *and* `figcaption` *elements are not supported in Internet Explorer versions 8 and earlier (see the sidebar HTML5 Support in Internet Explorer later in this chapter for a workaround). Older versions of Firefox and Safari (prior to 3.6 and 4, respectively) do not support it according to the spec, but allow you to apply styles.*

exercise 5-1 | Marking up a recipe

The owners of the Black Goose Bistro have decided to start a blog to share recipes and announcements. In the exercises in this chapter, we'll assist them with content markup.

Below you will find the raw text of a recipe. It's up to you to decide which element is the best semantic match for each chunk of content. You'll use paragraphs, headings, lists, and at least one special content element.

You can write the tags right on this page. Or, if you want to use a text editor and see the results in a browser, this text file is available online at *www.learningwebdesign.com/4e/materials*. The resulting code appears in Appendix A.

```
Tapenade (Olive Spread)

This is a really simple dish to prepare and it's always a big hit at parties. My father recommends:

"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."

Ingredients

1 8oz. jar sundried tomatoes
2 large garlic cloves
2/3 c. kalamata olives
1 t. capers

Instructions

Combine tomatoes and garlic in a food processor. Blend until as smooth as possible.

Add capers and olives. Pulse the motor a few times until they are incorporated, but still retain some
texture.

Serve on thin toast rounds with goat cheese and fresh basil garnish (optional).
```

Organizing Page Content

So far, the elements we've covered handle very specific tidbits of content: a paragraph, a heading, a figure, and so on. Prior to HTML5, there was no way to group these bits into larger parts other than wrapping them in a generic division (**`div`**) element (I'll cover **`div`** in more detail later). HTML5 introduced new elements that give semantic meaning to sections of a typical web page or application, including sections (**`section`**), articles (**`article`**), navigation (**`nav`**), tangentially related content (**`aside`**), headers (**`header`**), and footers (**`footer`**). The new element names are based on a Google study that looked at the top 20 names given to generic division elements (*code.google.com/webstats/2005-12/classes.html*). Curiously, the spec lists the old **`address`** element as a section as well, so we'll look at that one here too.

The elements discussed in this section are well supported by current desktop and mobile browsers, but there is a snag with Internet Explorer versions 8 and earlier. See the sidebar HTML5 Support in Internet Explorer for details on a workaround.

HTML5 Support in Internet Explorer

Most browsers today support the new HTML5 semantic elements, and for those that don't, creating a style sheet rule that tells browsers to format each one as a block-level element is all that is needed to make them behave correctly.

```
section, article, nav, aside, header, footer,
hgroup { display: block; }
```

Unfortunately, that fix won't work with Internet Explorer versions 8 and earlier (versions 9 and later are fine). Not only do early IE browsers not recognize the elements, they also ignore any styles applied to them. The solution is to use JavaScript to create each element so IE knows it exists and will allow nesting and styling. Here's what a JavaScript command creating the **section** element looks like:

```
document.createElement("section");
```

Fortunately, Remy Sharp created a script that creates all of the HTML5 elements for IE in one fell swoop. It is called "HTML5 Shiv" (or Shim) and it lives on a Google-run server, so you can just point to it in your documents. To make sure the new HTML5 elements work in IE8 and earlier, copy this code in the **head** of your document and use a style sheet to style the new elements as blocks:

```
<!--[if lt IE 9]>
<script src="http://html5shiv.googlecode.com/svn/
trunk/html5-els.js"></script>
<![endif]-->
```

Find out more about the HTML5 Shiv here: *html5doctor.com/how-to-get-html5-working-in-ie-and-firefox-2/*.

The HTML5 Shiv is also part of the Modernizr polyfill script that adds HTML5 and CSS3 functionality to older non-supporting browsers. Read more about it online at *modernizr.com*. It is also discussed in Chapter 20, Using JavaScript.

Sections and articles

`<section>…</section>`

Thematic group of content

NEW IN HTML5

`<article>…</article>`

Self-contained, reusable composition

NEW IN HTML5

NOTE

The HTML5 spec recommends that if the purpose for grouping the elements is simply to provide a hook for styling, use the generic `div` *element instead.*

Long documents are easier to use when they are divided into smaller parts. For example, books are divided into chapters, and newspapers have sections for local news, sports, comics, and so on. To divide long web documents into thematic sections, use the aptly named **section** element. Sections typically have a heading (inside the **section** element) and any other content that has a meaningful reason to be grouped together.

The **section** element has a broad range of uses, from dividing a whole page into major sections or identifying thematic sections within a single article. In the following example, a document with information about typography resources has been divided into two sections based on resource type.

```
<section>
  <h2>Typography Books</h2>
  <ul>
     <li>…</li>
  </ul>
</section>

<section>
  <h2>Online Tutorials</h2>
  <p>These are the best tutorials on the web.</p>
  <ul>
    <li>…</li>
  </ul>
</section>
```

Use the **article** element for self-contained works that could stand alone or be reused in a different context (such as syndication). It is useful for magazine or newspaper articles, blog posts, comments, or other items that could be extracted for external use. You can think of it as a specialized section element that answers the question "Could this appear on another site and make sense?" with "yes."

To make things interesting, a long **article** could be broken into a number of sections, as shown here:

```
<article>
  <h1>Get to Know Helvetica</h1>
  <section>
    <h2>History of Helvetica</h2>
    <p>…</p>
  </section>

  <section>
    <h2>Helvetica Today</h2>
    <p>…</p>
  </section>
</article>
```

Conversely, a **section** in a web document might be comprised of a number of articles.

```
<section id="essays">
  <article>
    <h1>A Fresh Look at Futura</h1>
    <p>…</p>
  </article>

  <article>
    <h1>Getting Personal with Humanist</h1>
    <p>…</p>
  </article>
</section>
```

The **section** and **article** elements are easily confused, particularly because it is possible to nest one in the other and vice versa. Keep in mind that if the content is self-contained and could appear outside the current context, it is best marked up as an **article**.

Sectioning Elements

Another thing that **section** and **article** have in common "under the hood" is that both are what HTML5 calls sectioning elements. When a browser runs across a sectioning element in the document, it creates a new item in the document's outline automatically. In prior HTML versions, only headings (**h1**, **h2**, etc.) triggered new outline items. The new **nav** (primary navigation) and **aside** (for sidebar-like information) are also sectioning elements.

In the new HTML5 outlining system, a sectioning element may have its own internal heading hierarchy, starting with **h1**, regardless of its position in the document that contains it. That makes it possible to take an **article** element with its internal outline, place it in another document flow, and know that it won't break the host document's outline. The goal of the new outlining algorithm is to make the markup meet the needs of content use and reuse on the modern Web.

As of this writing, no browsers support the HTML5 outlining system, so to make your documents accessible and logically structured for all users, it is safest to use headings in descending numerical order, even within sectioning elements.

For more information, I recommend the HTML5 Doctor article "Document Outlines," by Mike Robinson, that tackles HTML5 outlines in more detail than I am able to squeeze in here (*html5doctor.com/outlines/*).

In addition, Roger Johansson's article "HTML5 Sectioning Elements, Headings, and Document Outlines" describes some potential gotchas when working with sectioning elements (*www.456bereastreet.com/archive/201103/html5_sectioning_elements_headings_and_document_outlines/*).

Aside (sidebars)

`<aside>…<aside>`

Tangentially related material

NEW IN HTML5

The **aside** element identifies content that is related but tangential to the surrounding content. In print, its equivalent is a sidebar, but they couldn't call the element **sidebar**, because putting something on the "side" is a presentational description, not semantic. Nonetheless, a sidebar is a good mental model for using the **aside** element. **aside** can be used for pull quotes, background information, lists of links, callouts, or anything else that might be associated with (but not critical to) a document.

In this example, an **aside** element is used for a list of links related to the main article.

```
<h1>Web Typography</h1>
<p>Back in 1997, there were competing font formats and tools for making
them…</p>
<p>We now have a number of methods for using beautiful fonts on web
pages…</p>
<aside>
  <h2>Web Font Resources</h2>
  <ul>
  <li><a href="http://typekit.com/">Typekit</a></li>
  <li><a href="http://www.google.com/webfonts">Google Fonts</a></li>
  </ul>
</aside>
```

The **aside** element has no default rendering, so you will need to make it a block element and adjust its appearance and layout with style sheet rules.

Navigation

`<nav>…</nav>`

Primary navigation links

NEW IN HTML5

The new **nav** element gives developers a semantic way to identify navigation for a site. Earlier in this chapter, we saw an unordered list that might be used as the top-level navigation for a font catalog site. Wrapping that list in a **nav** element makes its purpose explicitly clear.

```
<nav>
<ul>
  <li><a href="">Serif</a>/li>
  <li><a href="">Sans-serif</a></li>
  <li><a href="">Script</a></li>
  <li><a href="">Display</a></li>
  <li><a href="">Dingbats</a>/li>
</ul>
</nav>
```

Not all lists of links should be wrapped in **nav** tags, however. The spec makes it clear that it should be used for links that provide primary navigation around a site or a lengthy section or article.

The **nav** element may be especially helpful from an accessibility perspective. Once screen readers and other devices become HTML5-compatible, users can easily get to or skip navigation sections without a lot of hunting around.

Headers and footers

Because web authors have been labeling header and footer sections in their documents for years, it was kind of a no-brainer that full-fledged **header** and **footer** elements would come in handy. Let's start with headers.

`<header>…</header>`

Introductory material for page, section, or article

NEW IN HTML5

`<footer>…</footer>`

Footer for page, section, or article

NEW IN HTML5

Headers

The **header** element is used for introductory material that typically appears at the beginning of a web page or at the top of a section or article. There is no specified list of what a **header** must or should contain; anything that makes sense as the introduction to a page or section is acceptable. In the following example, the document header includes a logo image, the site title, and navigation.

```
<header>
  <img src="/images/logo.png">
  <hgroup>
  <h1>Nuts about Web Fonts</h1>
  <h2>News from the Web Typography Front</h2>
  </hgroup>
  <nav>
    <ul>
      <li><a href="">Home</a></li>
      <li><a href="">Blog</a></li>
      <li><a href="">Shop</a></li>
    </ul>
  </nav>
</header>

… page content …
```

When used in an individual article, the **header** might include the article title, author, and the publication date, as shown here:

```
<article>
  <header>
    <h1>More about WOFF</h1>
    <p>by Jennifer Robbins, <time datetime="11-11-2011"
    pubdate>November 11, 2011</time></p>
  </header>
  <p>...article content starts here…</p>
</article>
```

WARNING

Neither `header` *nor* `footer` *elements are permitted to contain nested* `header` *or* `footer` *elements.*

Footers

The **footer** element is used to indicate the type of information that typically comes at the end of a page or an article, such as its author, copyright information, related documents, or navigation. The **footer** element may apply to the entire document, or it could be associated with a particular section or article. If the footer is contained directly within the **body** element, either before or after all the other **body** content, then it applies to the entire page or application. If it is contained in a sectioning element (**section**, **article**, **nav**, or **aside**), it is parsed as the footer for just that section. Note that although it is called "footer," there is no requirement that it come last in the docu-

ment or sectioning element. It could also appear at or near the beginning if it makes semantic sense.

NOTE

You can also add headers and footers to sectioning root elements: `body`, `blockquote`, `details`, `figure`, `td`, *and* `fieldset`.

In this simple example we see the typical information listed at the bottom of an article or blog post marked up as a **`footer`**.

```
<article>
  <header>
    <h1>More about WOFF</h1>
    <p>by Jennifer Robbins, <time datetime="11-11-2011"
    pubdate>November 11, 2011</time></p>
  </header>
  <p>...article content starts here…</p>
  <footer>
    <p><small>Copyright &copy;2012 Jennifer Robbins.</small></p>
    <nav>
    <ul>
      <li><a href="">Previous</a></li>
      <li><a href="">Next</a></li>
    </ul>
    </nav>
  </footer>
</article>
```

Addresses

`<address>...</address>`

Contact information

Last, and well, least, is the **`address`** element that is used to create an area for contact information for the author or maintainer of the document. It is generally placed at the end of the document or in a section or article within a document. An **`address`** would be right at home in a **`footer`** element.

It is important to note that the **`address`** element should *not* be used for any old address on a page, such as mailing addresses. It is intended specifically for author contact information (although that could potentially be a mailing address). Following is an example of its intended use. The "a href" parts are the markup for links...we'll get to those in Chapter 6, Adding Links.

NOTE

You'll get a chance to try out the section elements in Exercise 5-3 at the end of this chapter.

```
<address>
Contributed by <a href="../authors/robbins/">Jennifer Robbins</a>,
<a href="http://www.oreilly.com/">O'Reilly Media</a>
</address>
```

The Inline Element Roundup

Now that we've identified the larger chunks of content, we can provide semantic meaning to phrases within the chunks using what HTML5 calls text-level semantic elements. On the street, you are likely to hear them called inline elements because they display in the flow of text by default and do not cause any line breaks. That's also how they were referred to in HTML versions prior to HTML5.

Text-level (inline) elements

Despite all the types of information you could add to a document, there are only a couple dozen text-level semantic elements in HTML5. Table 5-1 lists all of them.

Table 5-1. Text-level semantic elements

Element	Description
a	An anchor or hypertext link (see Chapter 6 for details)
abbr	Abbreviation
b	Added visual attention, such as keywords (bold)
bdi	NEW IN HTML5 Indicates text that may have directional requirements
bdo	Bidirectional override; explicitly indicates text direction (left to right, ltr, or right to left, rtl)
br	Line break
cite	Citation; a reference to the title of a work, such as a book title
code	Computer code sample
data	WHATWG ONLY Machine-readable equivalent dates, time, weights, and other measurable values
del	Deleted text; indicates an edit made to a document
dfn	The defining instance or first occurrence of a term
em	Emphasized text
i	Alternative voice (italic)
ins	Inserted text; indicates an insertion in a document
kbd	Keyboard; text entered by a user (for technical documents)
mark	NEW IN HTML5 Contextually relevant text
q	Short, inline quotation
ruby, rt, rp	NEW IN HTML5 Provides annotations or pronunciation guides under East Asian typography and ideographs
s	Incorrect text (strike-through)
samp	Sample output from programs
small	Small print, such as a copyright or legal notice (displayed in a smaller type size)
span	Generic phrase content
strong	Content of strong importance
sub	Subscript
sup	Superscript
time	NEW IN HTML5 Machine-readable time data
u	Underlined
var	A variable or program argument (for technical documents)
wbr	Word break

The Inline Elements Backstory

Many of the inline elements that have been around since the dawn of the Web were introduced to change the visual formatting of text selections due to the lack of a style sheet system. If you wanted bolded text, you marked it as **b**. Italics? Use the **i** element. In fact, there was once a **font** element used solely to change the font, color, and size of text (the horror!). Not surprisingly, HTML5 kicked the purely presentational **font** element to the curb. However, many of the old-school presentational inline elements (for example, **u** for underline and **s** for strike-through) have been kept in HTML5 and given new semantic definitions (**b** is now for "keywords," **s** for "inaccurate text").

Some inline elements are purely semantic (such as **abbr** or **time**) and don't have default renderings. For these, you'll need to use a CSS rules if you want to change the way they display.

In the element descriptions in this section, I'll provide both the definition of the inline elements and the expected browser default rendering if there is one.

Obsolete HTML 4.01 Text Elements

HTML5 finally retired many elements that were marked as deprecated (phased out and discouraged from use) in HTML 4.01. For the sake of thoroughness, I include them here in case you run across them in legacy markup. But there's no reason to use them—most have analogous style sheet properties or are simply poorly supported.

Element	Description
`acronym`	Indicates an acronym (e.g., NASA); authors should use **`abbr`** instead
`applet`	Inserts a Java applet
`basefont`	Establishes default font settings for a document
`big`	Makes text slightly larger than default text size
`center`	Centers content horizontally
`dir`	Directory list (replaced by unordered lists)
`font`	Font face, color, and size
`isindex`	Inserts a search box
`menu`	Menu list (replaced by unordered lists; however, **`menu`** is now used to provide contextual menu commands)
`strike`	Strike-through text
`tt`	Teletype; displays in constant-width font

Emphasized text

`<em>...</em>`
Stressed emphasis

Use the **`em`** element to indicate which part of a sentence should be stressed or emphasized. The placement of **`em`** elements affects how a sentence's meaning is interpreted. Consider the following sentences that are identical, except for which words are stressed.

```
<p><em>Matt</em> is very smart.</p>
<p>Matt is <em>very</em> smart.</p>
```

The first sentence indicates *who* is very smart. The second example is about *how* smart he is.

Emphasized text (**`em`**) elements nearly always display in italics by default (Figure 5-9), but of course you can make them display any way you like with a style sheet. Screen readers may use a different tone of voice to convey stressed content, which is why you should use an **`em`** element only when it makes sense semantically, not just to achieve italic text.

Important text

`<strong>...</strong>`
Strong importance

The **`strong`** element indicates that a word or phrase as important. In the following example, the **`strong`** element identifies the portion of instructions that requires extra attention.

```
<p>When checking out of the hotel, <strong>drop the keys in the red box
by the front desk</strong>.</p>
```

Visual browsers typically display **strong** text elements in bold text by default. Screen readers may use a distinct tone of voice for important content, so mark text as **strong** only when it makes sense semantically, not just to make text bold.

The following is a brief example of our **em** and **strong** text examples. Figure 5-9 should hold no surprises.

Matt is very smart.

Matt is *very* smart.

When returning the car, **drop the keys in the red box by the front desk**.

Figure 5-9. The default rendering of emphasized and strong text.

The previously presentational elements that are sticking around in HTML5 with fancy new semantic definitions

As long as we're talking about bold and italic text, let's see what the old **b** and **i** elements are up to now. The elements **b**, **i**, **u**, **s**, and **small** were introduced in the old days of the Web as a way to provide typesetting instructions (bold, italic, underline, strikethrough, and smaller text, respectively). Despite their original presentational purposes, these elements have been included in HTML5 and given updated, semantic definitions based on patterns of how they've been used. Browsers still render them by default as you'd expect (Figure 5-10). However, if a type style change is all you're after, using a style sheet rule is the appropriate solution. Save these for when they are semantically appropriate.

Let's look at these elements and their correct usage, as well as the style sheet alternatives.

`<b>...</b>`
Keywords or visually emphasized text (bold)

`<i>...</i>`
Alternative voice (italic)

`<s>...</s>`
Incorrect text (strike-through)

`<u>...</u>`
Annotated text (underline)

`<small>...</small>`
Legal text; small print (smaller type size)

b

HTML 4.01 definition: Bold

HTML5 definition: Keywords, product names, and other phrases that need to stand out from the surrounding text without conveying added importance or emphasis.

CSS alternative: For bold text, use **font-weight**. Example: **font-weight: bold**

Example: `<p>The slabs at the ends of letter strokes are called <b>serifs</b>.</p>`

NOTE

It helps me to think about how a screen reader would read the text. If I don't want the word read in a loud, emphatic tone of voice, but it really should be bold, then `b` *may be more appropriate than* `strong`.

i

HTML 4.01 definition: Italic

HTML5 definition: Indicates text that is in a different voice or mood than the surrounding text, such as a phrase from another language, a technical term, or thought.

CSS alternative: For italic text, use **font-style**. Example: **font-style: italic**

Example:
```
<p>Simply change the font and <i>Voila!</i>, a new personality.</p>
```

s

HTML 4.01 definition: Strike-through text

HTML5 definition: Indicates text that is incorrect.

CSS Property: To put a line through a text selection, use **text-decoration**. Example: **text-decoration: line-through;**

Example:
```
<p>Scala Sans was designed by <s>Eric Gill</s> Martin Majoor.</p>
```

u

HTML 4.01 definition: Underline

HTML5 definition: There are a few instances when underlining has semantic significance, such as underlining a formal name in Chinese or indicating a misspelled word after a spell check. Note that underlined text is easily confused as a link and should generally be avoided except for a few niche cases.

CSS Property: For underlined text, use **text-decoration**. Example: **text-decoration: underline**

Example:
```
<p>New York subway signage is set in <u>Halvetica</u>.<p>
```

small

HTML 4.01 definition: Renders in font smaller than the surrounding text

HTML5 definition: Indicates an addendum or side note to the main text, such as the legal "small print" at the bottom of a document.

CSS Property: To make text smaller, use **font-size**. Example: **font-size: 80%**

Example:
```
<p><a href="">Download <b>Jenville Handwriting Font</b></a></p>

<p><small>This font is free for commercial use.</small></p>
```

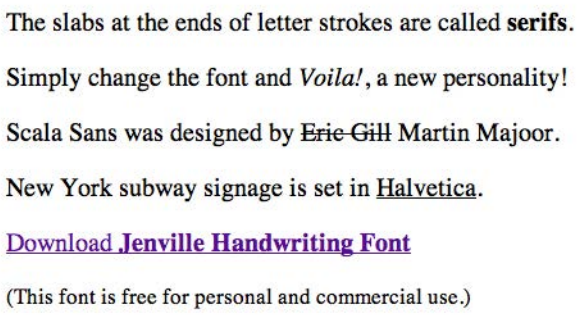


Figure 5-10. The default rendering of `b`, `i`, `u`, `s`, *and* `small` *elements.*

Short quotations

Use the quotation (`q`) element to mark up short quotations, such as "To be or not to be," in the flow of text, as shown in this example (Figure 5-11).

```
Matthew Carter says, <q>Our alphabet hasn't changed in eons.</q>
```

`<q>...</q>`

Short inline quotation

According to the HTML spec, browsers should add quotation marks around `q` elements automatically, so you don't need to include them in the source document. And for the most part they do, with the exception of Internet Explorer versions 7 and earlier. Fortunately, as of this writing, those browsers make up only 5–8% of browser usage, and it's sure to be significantly less by the time you read this. If you are concerned about a small percentage of users seeing quotations without their marks, stick with using quotation marks in your source, a fine alternative.

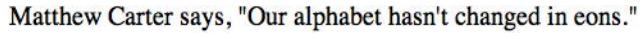


Figure 5-11. Nearly all browsers add quotation marks automatically around `q` *elements.*

REMINDER

Nesting Elements

You can apply two elements to a string of text (for example, a phrase that is both a quote and in another language), but be sure they are nested properly. That means the inner element, including its closing tag, must be completely contained within the outer element, and not overlap.

```
<q><i>Je ne sais pas.</i></q>
```

Abbreviations and acronyms

Marking up acronyms and abbreviations with the `abbr` element provides useful information for search engines, screen readers, and other devices. Abbreviations are shortened versions of a word ending in a period (Conn. for Connecticut, for example). Acronyms are abbreviations formed by the first letters of the words in a phrase (such as WWW or USA). The `title` attribute provides the long version of the shortened term, as shown in this example:

```
<abbr title="Points">pts.</abbr>
<abbr title="American Type Founders">ATF</abbr>
```

`<abbr>...</abbr>`

Abbreviation or acronym

NOTE

In HTML 4.01, there was an `acronym` *element especially for acronyms, but it has been made obsolete in HTML5 in favor of using the* `abbr` *for both.*

Citations

`<cite>...</cite>`
Citation

The **cite** element is used to identify a reference to another document, such as a book, magazine, article title, and so on. Citations are typically rendered in italic text by default. Here's an example:

```
<p>Passages of this article were inspired by <cite>The Complete Manual
of Typography</cite> by James Felici.</p>
```

Defining terms

`<dfn>...</dfn>`
Defining term

It is common to point out the first and defining instance of a word in a document in some fashion. In this book, defining terms are set in blue text. In HTML, you can identify them with the **dfn** element and format them visually using style sheets.

```
<p><dfn>Script typefaces</dfn> are based on handwriting.</p>
```

Program code elements

`<code>...</code>`
Code

`<var>...</var>`
Variable

`<samp>...</samp>`
Program sample

`<kbd>...</kbd>`
User-entered keyboard strokes

A number of inline elements are used for describing the parts of technical documents, such as code (**code**), variables (**var**), program samples (**samp**), and user-entered keyboard strokes (**kbd**). For me, it's a quaint reminder of HTML's origins in the scientific world (Tim Berners-Lee developed HTML to share documents at the CERN particle physics lab in 1989).

Code, sample, and keyboard elements typically render in a constant-width (also called monospace) font such as Courier by default. Variables usually render in italics.

Subscript and superscript

`<sub>...</sub>`
Subscript

`<sup>...</sup>`
Superscript

The subscript (**sub**) and superscript (**sup**) elements cause the selected text to display in a smaller size, positioned slightly below (**sub**) or above (**sup**) the baseline. These elements may be helpful for indicating chemical formulas or mathematical equations.

Figure 5-12 shows how these examples of subscript and superscript typically render in a browser.

```
<p>H<sub>2</sub>O</p>

<p>E=MC<sup>2</sup></p>
```

Figure 5-12. Subscript and superscript

Highlighted text

The new **mark** element indicates a word that may be considered especially relevant to the reader. One might use it to call out a search term in a page of results, to manually call attention to a passage of text, indicate the current page in a series. Some designers (and browsers) give marked text a light colored background as though it was marked with a highlighter marker, as shown in Figure 5-13.

```
<p> ... PART I. ADMINISTRATION OF THE GOVERNMENT. TITLE IX.
TAXATION. CHAPTER 65C. MASS. <mark>ESTATE TAX</mark>. Chapter 65C:
Sect. 2. Computation of <mark>estate tax</mark>.</p>
```

... PART I. ADMINISTRATION OF THE GOVERNMENT. TITLE IX. TAXATION. CHAPTER 65C. MASS. ESTATE TAX. Chapter 65C: Sect. 2. Computation of estate tax.

Figure 5-13. Search terms are marked as mark elements and given a yellow background with a style sheet so they are easier for the reader to find.

`<mark>...</mark>`

Contextually relevant text

NEW IN HTML5

SUPPORT ALERT

The **mark** *element is not supported in Internet Explorer versions 8 and earlier (see the sidebar HTML5 Support in Internet Explorer earlier in this chapter for a workaround). Older versions of Firefox and Safari (prior to 3.6 and 4, respectively) do not support it according to the spec, but do allow you to apply styles to it.*

Times and machine-readable information

When we look at the phrase "noon on November 4," we know that it is a date and a time. But the context might not be so obvious to a computer program. The **time** element allows us to mark up dates and times in a way that is comfortable for a human to read, but also encoded in a standardized way that computers can use. The content of the element presents the information to people, and the **datetime** attribute presents the same information in a machine-readable way.

The **time** element indicates dates, times, or date-time combos. It might be used to pass the date and time information to an application, such as saving an event to a personal calendar. It might be used by search engines to find the most recently published articles. Or it could be used to restyle time information into an alternate format (e.g., changing 18:00 to 6 p.m.).

The **datetime** attribute specifies the date and/or time information in a standardized time format illustrated in Figure 5-14. It begins with the date (year, month, day), followed by the letter T to indicate time, listed in hours, minutes, seconds (optional), and milliseconds (also optional). Finally, the time zone is indicated by the number of hours behind (–) or ahead (+) of Greenwich Mean Time (GMT). For example, "–05:00" indicates the Eastern Standard time zone, which is five hours behind GMT.

`<time>...</time>`

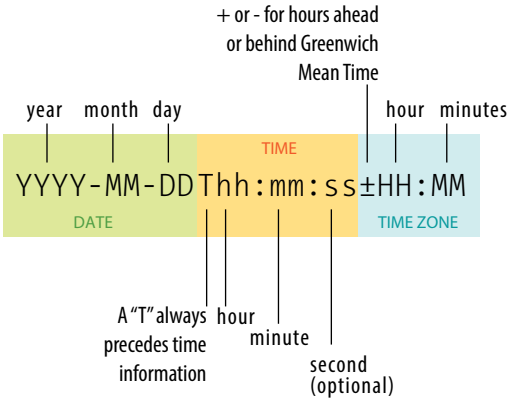
Time data

NEW IN HTML5

NOTE

The **time** *element is not intended for marking up times for which a precise time or date cannot be established, such as "the end of last year" or "the turn of the century."*

Figure 5-14. Standardized date and time syntax.



Example:

3pm PST on December 25, 2012

```
2012-12-25T15:00-8:00
```

The WHATWG HTML specification includes a **pubdate** attribute for indicating that the time is the publication date of a document, as shown in this example. The **pubdate** attribute is not included in the W3C HTML5 spec as of this writing, but it may be included at a later date if it becomes widely used.

```
Written by Jennifer Robbins (<time datetime="2012-09-01T 20:00-05:00"
pubdate>September 1, 2012, 8pm EST</time>)
```

`<data>...</data>`

Machine-readable data

WHATWG ONLY

The WHATWG also includes the **data** element for helping computers make sense of content, which can be used for all sorts of data, including dates, times, measurements, weights, and so on. It uses the **value** attribute for the machine-readable information. Here are a couple of examples:

```
<data value="12">Twelve</data>
<data value="2011-11-12">Last Saturday</data>
```

SUPPORT ALERT

Both `time` *and* `data` *are new elements and are not universally supported as of this writing. However, you can apply styles to them and they will be recognized by browsers other than IE8 and earlier.*

I'm not going to go into more detail on the **data** element, because as of this writing, the powers that be are still discussing exactly how it should work, and the W3C has not adopted it for the HTML5 spec. Also, as a beginner, you are unlikely to be dealing with machine-readable data yet anyway. But still, it is interesting to see how markup can be used to provide usable information to computer programs and scripts as well as to your fellow humans.

Inserted and deleted text

`<ins>...</ins>`

Inserted text

`<del>...</del>`

Deleted text

The **ins** and **del** elements are used to mark up edits indicating parts of a document that have been inserted or deleted (respectively). These elements rely on style rules for presentation (i.e., there is no dependable browser default). Both the **ins** and **del** elements can contain either inline or block elements, depending on what type of content they contain.

```
Chief Executive Officer: <del title="retired">Peter Pan</del><ins>Pippi
Longstockings</ins>
```

Adding Breaks

Line breaks

`<br>`

Line break

Occasionally, you may need to add a line break within the flow of text. We've seen how browsers ignore line breaks in the source document, so we need a specific directive to tell the browser to "add a line break here."

The inline line break element (**br**) does exactly that. The **br** element could be used to break up lines of addresses or poetry. It is an empty element, which means it does not have content. Just add the **br** element (**
** in XHTML) in the flow of text where you want a break to occur, as shown in here and in Figure 5-15.

```
<p>So much depends <br>upon <br><br>a red wheel <br>barrow</p>
```

So much depends
upon

a red wheel
barrow

Figure 5-15. Line breaks are inserted at each `br` *element.*

Accommodating Non-Western Languages

Because the Web is "world-wide," there are a few elements designed to address the needs of non-western languages.

Changing direction

The **bdo** (bidirectional override) element allows a phrase in a right-to-left (**rtl**) reading language (such as Hebrew or Arabic) to be included in a left-to-right (**ltr**) reading flow, or vice versa.

```
This is how you write Shalom: <bdo dir="rtl">&#x0
5E9;&#x05DC;&#x05D5;&#x05DD;</bdo>
```

The **bdi** (bidirectional isolation) element is similar, but it is used to isolate a selection that *might* read in a different direction, such as a name or comment added by a user.

Hints for East Asian languages

HTML5 also includes the **ruby**, **rt**, and **rp** elements used to add ruby annotation to East Asian languages. Ruby annotations are little notes that typically appear above ideographs and provide pronunciation clues or translations. Within the **ruby** element, the **rt** element indicates the helpful ruby text. Browsers that support ruby text typically display it in a smaller font above the main text. As a backup for browsers that don't support ruby, you can put the ruby text in parentheses, each marked with the **rp** element. Non-supporting browsers display all the text on the same line, with the ruby in parentheses. Supporting browsers ignore the content of the **rp** elements and display only the **rt** text above the glyphs. The Ruby system has spotty browser support as of this writing.

```
<ruby>
 字<rp>(</rp><rt>han</rt><rp>)<rp>
 汉<rp>(</rp><rt>zi</rt><rp>)<rp>
</ruby>
```

This example was taken from the HTML5 Working Draft at whatwg.com, used with permission under an MIT License.

Unfortunately, the **br** element is easily abused (see the following warning). Consider whether using the CSS **white-space** property (introduced in Chapter 12, Formatting Text) might be a better alternative for maintaining line breaks from your source without extra markup.

Word breaks

<wbr>

Word break

The word break (**wbr**) element lets you mark the place where a word should break if it needs to (a "line break opportunity" according to the spec). It takes some of the guesswork away from the browser and allows authors to specify the best spot for the word to be split over two lines. Keep in mind that the word breaks at the **wbr** element only if it needs to (Figure 5-16). If there is enough room, the word stays in one piece. Browsers have supported this element for a long time, but it has recently been incorporated into the HTML standard.

```
<p>The biggest word you've ever heard and this is how it goes:
<em>supercali<wbr>fragilistic<wbr>expialidocious</em>!</p>
```

The biggest word you've ever heard and this is how it goes: *supercalifragilistic expialidocious*!

Figure 5-16. When there is not enough room for a word to fit on a line, it will break at the location of the **wbr** *element.*

WARNING

Be careful that you aren't using **br** *elements to force breaks into text that really ought to be a list. For example, don't do this:*

```
<p>Times<br>
Georgia<br>
Garamond
</p>
```

If it's a list, use the semantically correct unordered list element instead, and turn off the bullets with style sheets.

```
<ul>
  <li>Times</li>
  <li>Georgia</li>
  <li>Garamond</li>
</ul>
```

exercise 5-2 | Identifying inline elements

This little post for the Black Goose Bistro blog will give you an opportunity to identify and mark up a variety of inline elements. See if you can find phrases to mark up accurately with the following elements:

`b` `br` `cite` `dfn` `em` `i` `q` `small` `time`

Because markup is always somewhat subjective, your resulting markup may not look exactly like the example in Appendix A, but there is an opportunity to use all of the elements listed above in the article. For extra credit, there is a phrase that should have two elements applied to it (remember to nest them properly by closing the inner element before you close the outer one).

You can write the tags right on this page. Or, if you want to use a text editor and see the results in a browser, this text file is available online at *www.learningwebdesign.com/4e/materials*. The resulting code appears in Appendix A.

```
<article>
   <header>
   <p>posted by BGB, November 15, 2012</p>
   </header>
<h1>Low and Slow</h1>
<p>This week I am extremely excited about a new cooking technique
called sous vide. In sous vide 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, Cooking for Geeks, Jeff Potter describes
it as ultra-low-temperature poaching.</p>
<p>Next month, we will be serving Sous Vide Salmon with Dill
Hollandaise. To reserve a seat at the chef table, contact us
before November 30.</p>

<p>blackgoose@example.com
555-336-1800</p>

<p> Warning: Sous vide cooked salmon is not pasteurized. Avoid it
if you are pregnant or have immunity issues.</p>
</article>
```

Generic Elements (div and span)

`<div>...</div>`
Generic block-level element

`<span>...</span>`
Generic inline element

What if none of the elements we've talked about so far accurately describes your content? After all, there are endless types of information in the world, but as you've seen, not all that many semantic elements. Fortunately, HTML provides two generic elements that can be customized to describe your content perfectly. The **div** element indicates a division of content, and **span** indicates a word or phrase for which no text-level element currently exists. The generic elements are given meaning and context with the **id** and **class** attributes, which we'll discuss in a moment.

The **div** and **span** elements have no inherent presentation qualities of their own, but you can use style sheets to format them however you like. In fact, generic elements are a primary tool in standards-based web design because they enable authors to accurately describe content and offer plenty of "hooks" for adding style rules. They also allow elements on the page to be accessed and manipulated by JavaScript.

We're going to spend a little time on **div** and **span** (as well as the **id** and **class** attributes) and learn how authors use them to structure content.

Divide it up with a div

The **div** element is used to create a logical grouping of content or elements on the page. It indicates that they belong together in some sort of conceptual unit or should be treated as a unit by CSS or JavaScript. By marking related content as a **div** and giving it a unique **id** identifier or indicating that it is part of a **class**, you give context to the elements in the grouping. Let's look at a few examples of **div** elements.

MARKUP TIP

It is possible to nest **div** elements within other **div** elements, but don't go overboard. You should always strive to keep your markup as simple as possible, so add a **div** element only if it is necessary for logical structure, styling, or scripting.

In this example, a **div** element is used as a container to group an image and two paragraphs into a product "listing."

```
<div class="listing">
  <img src="felici-cover.gif" alt="">
  <p><cite>The Complete Manual of Typography</cite>, James Felici</p>
  <p>A combination of type history and examples of good and bad type
design.</p>
</div>
```

By putting those elements in a **div**, I've made it clear that they are conceptually related. It will also allow me to style two **p** elements within listings differently than other paragraphs on the page.

Here is another common use of a **div** used to break a page into sections for layout purposes. In this example, a heading and several paragraphs are enclosed in a **div** and identified as the "news" division.

```
<div id="news">
  <h1>New This Week</h1>
  <p>We've been working on...</p>
  <p>And last but not least,... </p>
</div>
```

Now that I have an element known as "news," I could use a style sheet to position it as a column to the right or left of the page. You might be thinking, "Hey Jen, couldn't you use a **section** element for that?" You could! In fact, authors may turn to generic **div**s less now that we have better semantic grouping elements in HTML5.

Get inline with span

A **span** offers the same benefits as the **div** element, except it is used for phrase elements and does not introduce line breaks. Because spans are inline elements, they can only contain text and other inline elements (in other words, you cannot put headings, lists, content-grouping elements, and so on, in a **span**). Let's get right to some examples.

There is no **telephone** element, but we can use a **span** to give meaning to telephone numbers. In this example, each telephone number is marked up as a **span** and classified as "tel":

```
<ul>
  <li>John: <span class="tel">999.8282</span></li>
  <li>Paul: <span class="tel">888.4889</span></li>
  <li>George: <span class="tel">888.1628</span></li>
  <li>Ringo: <span class="tel">999.3220</span></li>
</ul>
```

You can see how the classified spans add meaning to what otherwise might be a random string of digits. As a bonus, the **span** element enables us to apply the same style to phone numbers throughout the site (for example, ensuring line breaks never happen within them, using a CSS **white-space: nowrap** declaration). It makes the information recognizable not only to humans but to computer programs that know that "tel" is telephone number information. In fact, some values—including "tel"—have been standardized in a markup system known as Microformats that makes web content more useful to software (see the Microformats and Metadata sidebar).

id and class attributes

In the previous examples, we saw the **id** and **class** attributes used to provide context to generic **div** and **span** elements. **id** and **class** have different purposes, however, and it's important to know the difference.

Identification with id

The **id** attribute is used to assign a *unique* identifier to an element in the document. In other words, the value of **id** must be used only once in the document. This makes it useful for assigning a name to a particular element, as though it were a piece of data. See the sidebar id and class Values for information on providing values for the **id** attribute.

This example uses the books' ISBN numbers to uniquely identify each listing. No two book listings may share the same **id**.

id and class Values

The values for **id** and **class** attributes should start with a letter (A–Z or a–z) or underscore (although Internet Explorer 6 and earlier have trouble with underscores, so they are generally avoided). They should not contain any character spaces or special characters. Letters, numbers, hyphens, underscores, colons, and periods are OK. Also, the values are case-sensitive, so "sectionB" is not interchangeable with "Sectionb."

```
<div id="ISBN0321127307">
  <img src="felici-cover.gif" alt="">
  <p><cite>The Complete Manual of Typography</cite>, James Felici</p>
  <p>A combination of type history and examples of good and bad type.
  </p>
</div>

<div id="ISBN0881792063">
  <img src="bringhurst-cover.gif" alt="">
  <p><cite>The Elements of Typographic Style</cite>, Robert Bringhurst
  </p>
  <p>This lovely, well-written book is concerned foremost
   with creating beautiful typography.</p>
</div>
```

Web authors also use **id** when identifying the various sections of a page. In the following example, there may not be more than one element with the **id** of "main," "links," or "news" in the document.

```
<section id="main">
  <!-- main content elements here -->
</section>

<section id="news">
  <!-- news items here -->
</section>

<aside id="links">
  <!-- list of links here -->
</aside>
```

Not Just for divs

The **id** and **class** attributes may be used with all elements in HTML5, not just **div** and **span**. For example, you could identify an ordered list as "directions" instead of wrapping it in a **div**.

```
<ol id="directions">
  <li>...</li>
  <li>...</li>
  <li>...</li>
</ol>
```

Note that in HTML 4.01, **id** and **class** may be used with all elements except **base**, **basefont**, **head**, **html**, **meta**, **param**, **script**, **style**, and **title**.

Microformats and Metadata

As you've seen, the elements in HTML fall short in describing every type of information. A group of developers decided that if **class** names could be standardized (for example, always using "tel" for telephone numbers), they could establish systems for describing data to make it more useful. This system is called Microformats. Microformats extend the semantics of HTML markup by establishing standard *values* for **id**, **class**, and **rel** attributes rather than creating whole new elements.

There are several Microformat "vocabularies" used to identify things such as contact information (hCard) or calendar items (hCalendar). The Microformats.org site is a good place to learn about them. To give you the general idea, the following example describes the parts of an event using the hCalendar Microformat vocabulary so the browser can automatically add it to your calendar program.

```
<section class="vevent">
 <span class="summary">O'Reilly Emerging
 Technology Conference</span>,
 <time class="dtstart" datetime="20110306">Mar 6
 </time> -
 <time class="dtend" datetime="20110310">10,
 2011</time>
 <div class="location">Manchester Grand Hyatt,
 San Diego, CA</div>
 <a class="url" href="http://events.example.com
 pub/e/403">Permalink</a>
</section>
```

The hCard vocabulary identifies components of typical contact information (stored in vCard format), including: address (**adr**), postal code (**postal-code**), states (**region**), and telephone numbers (**tel**), to name a few. The browser can then use a service to grab the information from the web page and automatically add it to an address book.

There is a lot more to say about Microformats than I can fit in this book. And not only that, but there are two additional, more complex systems for adding metadata to web pages in development at the W3C: RDFa and Microdata. It's not clear how they are all going to shake out in the long run, and I'm thinking that this metadata stuff is more than you want to take on right now anyway. But when you are ready to learn more, WebSitesMadeRight.com has assembled a great big list of introductory articles and tutorials on all three options: *websitesmaderight.com/2011/05/html5-microdata-microformats-and-rdfa-tutorials-and-resources/*.

Classification with class

The **class** attribute classifies elements into conceptual groups; therefore, unlike the **id** attribute, multiple elements may share a **class** name. By making elements part of the same class, you can apply styles to all of the labeled elements at once with a single style rule or manipulate them all with a script. Let's start by classifying some elements in the earlier book example. In this first example, I've added **class** attributes to classify each **div** as a "listing" and to classified paragraphs as "descriptions."

```
<div id="ISBN0321127307" class="listing">
  <header>
  <img src="felici-cover.gif" alt="">
  <p><cite>The Complete Manual of Typography</cite>, James Felici</p>
  </header>
  <p class="description">A combination of type history and examples of
good and bad type.</p>
</div>

<div id="ISBN0881792063" class="listing">
  <header>
  <img src="bringhurst-cover.gif" alt="">
  <p><cite>The Elements of Typographic Style</cite>, Robert Bringhurst
  </p>
  </header>
  <p class="description">This lovely, well-written book is concerned
foremost with creating beautiful typography.</p>
</div>
```

TIP

The **id** attribute is used to *identify.*
The **class** attribute is used to *classify.*

Notice how the same element may have both a **class** and an **id**. It is also possible for elements to belong to multiple classes. When there is a list of **class** values, simply separate them with character spaces. In this example, I've classified each **div** as a "book" to set them apart from possible "cd" or "dvd" listings elsewhere in the document.

```
<div id="ISBN0321127307" class="listing book">
  <img src="felici-cover.gif" alt="CMT cover">
  <p><cite>The Complete Manual of Typography</cite>, James Felici</p>
  <p class="description">A combination of type history and examples of
good and bad type.</p>
</div>

<div id="ISBN0881792063" class="listing book">
  <img src="bringhurst-cover.gif" alt="ETS cover">
  <p><cite>The Elements of Typographic Style</cite>, Robert Bringhurst
  </p>
  <p class="description">This lovely, well-written book is concerned
  foremost with creating beautiful typography.</p>
</div>
```

This should have given you a good introduction to how **div** and **span** elements with **class** and **id** attributes are used to add meaning and organization to documents. We'll work with them even more in the style sheet chapters in Part III.

Some Special Characters

There's just one more text-related topic before we close this chapter out.

Some common characters, such as the copyright symbol ©, are not part of the standard set of ASCII characters, which contains only letters, numbers, and a few basic symbols. Other characters, such as the less-than symbol (<), are available, but if you put one in an HTML document, the browser will interpret it as the beginning of a tag.

Characters such as these must be escaped in the source document. Escaping means that instead of typing in the character itself, you represent it by its numeric or named character reference. When the browser sees the character reference, it substitutes the proper character in that spot when the page is displayed.

There are two ways of referring to a specific character: by an assigned numeric value (numeric entity) or using a predefined abbreviated name for the character (called a named entity). All character references begin with an "&" and end with a ";".

Some examples will make this clear. I'd like to add a copyright symbol to my page. The typical Mac keyboard command, Option-G, which works in my word processing program, may not be understood properly by a browser or other software. Instead, I must use the named entity **©** (or its numeric equivalent, **©**) where I want the symbol to appear (Figure 5-17).

```
<p>All content copyright &copy; 2012, Jennifer Robbins</p>
```

or:

```
<p>All content copyright &#169; 2012, Jennifer Robbins</p>
```

HTML defines hundreds of named entities as part of the markup language, which is to say you can't make up your own entity. Table 5-2 lists some commonly used character references. If you'd like to see them all, the complete list of character references has been assembled online by the nice folks at the Web Standards Project at *www.webstandards.org/learn/reference/charts/entities/*.

All content copyright © 2007, Jennifer Robbins

Figure 5-17. The special character is substituted for the character reference when the document is displayed in the browser.

NOTE

In XHTML, every instance of an ampersand must be escaped so that it is not interpreted as the beginning of a character entity, even when it appears in the value of an attribute. For example:

```
<img src="sno.jpg" alt="Sifl
&amp; Olly Show" />
```

Non-breaking Spaces

One interesting character to know about is the non-breaking space (**` `**). Its purpose is to ensure that a line doesn't break between two words. So, for instance, if I mark up my name like this:

```
Jennifer Robbins
```

I can be sure that my first and last names will always stay together on a line.

Table 5-2. Common special characters and their character references

Character	Description	Name	Number
	Character space (nonbreaking space)	` `	` `
&	Ampersand	`&`	`&#038;`
‘	Apostrophe	`&apos;`	`&#039;`
<	Less-than symbol (useful for displaying markup on a web page)	`<`	`&#060;`
>	Greater-than symbol (useful for displaying markup on a web page)	`>`	`&#062;`
©	Copyright	`&copy;`	`&#169;`
®	Registered trademark	`&reg;`	`&#174;`
™	Trademark	`&trade;`	`&#8482;`
£	Pound	`&pound;`	`&#163;`
¥	Yen	`&yen;`	`&#165;`
€	Euro	`&euro;`	`&#8364;`
–	En-dash	`&ndash;`	`&#8211;`
—	Em-dash	`&mdash;`	`&#8212;`
‘	Left curly single quote	`&lsquo;`	`&#8216;`
’	Right curly single quote	`&rsquo;`	`&#8217;`
“	Left curly double quote	`&ldquo;`	`&#8220;`
”	Right curly double quote	`&rdquo;`	`&#8221;`
•	Bullet	`&bull;`	`&#8226;`
…	Horizontal ellipsis	`&hellip;`	`&#8230;`

Putting It All Together

TIP

Remember that indenting each hierarchical level in your HTML source consistently makes the document easier to scan and update later.

So far, you've learned how to mark up elements, and you've met all of the HTML elements for adding structure and meaning to text content. Now it's just a matter of practice. Exercise 5-3 gives you an opportunity to try out everything we've covered so far: document structure elements, block elements, inline elements, sectioning elements, and character entities. Have fun!

exercise 5-3 | The Black Goose Blog page

Now that you've been introduced to all of the text elements, you can put them to work by marking up the Blog page for the Black Goose Bistro site. The content is shown below (the second post is already marked up with the inline elements from Exercise 5-2). Get the starter text file online at *www.learningwebdesign.com/4e/materials.* The resulting markup is in Appendix A and included in the ***materials*** folder.

Once you have the text file, follow the instructions listed after the copy. The resulting page is shown in Figure 5-18.

```
The Black Goose Blog

Home
Menu
Blog
Contact

Summer Menu Items
posted by BGB, June 15, 2013
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.

Appetizers
Black bean purses
Spicy black bean and a blend of mexican cheeses
wrapped in sheets of phyllo and baked until
golden. $3.95

Southwestern napoleons with lump crab -- new
item!
Layers of light lump crab meat, bean and corn
salsa, and our handmade flour tortillas. $7.95

Main courses

Shrimp sate kebabs with peanut sauce
Skewers of shrimp marinated in lemongrass, garlic,
and fish sauce then grilled to perfection. Served
with spicy peanut sauce and jasmine rice. $12.95

Jerk rotisserie chicken with fried plantains --
new item!
Tender chicken slow-roasted on the rotisserie,
flavored with spicy and fragrant jerk sauce and
served with fried plantains and fresh mango.
$12.95

Low and Slow
posted by BGB, November 15, 2012
<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>


Location: Baker's Corner, Seekonk, MA
Hours: Tuesday to Saturday, 11am to midnight

All content copyright &copy; 2012, Black Goose
Bistro and Jennifer Robbins
```

Figure 5-18. The finished menu page.

1. Add all the document structure elements first (**`html`**, **`head`**, **`meta`**, **`title`**, and **`body`**). Give the document the title "Black Goose Bistro: Blog."

2. The first thing we'll do is identify the top-level heading and the list of links as the header for the document by wrapping them in a **header** element (don't forget the closing tag). Within the header, the headline should be an **h1** and the list of links should be an unordered list (**ul**). Don't worry about making the list items links; we'll get to linking in the next chapter. Give the list more meaning by identifying it as the primary navigation for the site (**nav**).
3. This blog page has two posts titled "Summer Menu Items" and "Low and Slow." Mark each one up as an **article**.
4. Now we'll get the first article into shape! Let's create a **header** for this article that contains the heading (**h2** this time because we've moved down in the document hierarchy) and the publication information (**p**). Identify the publication date for the article with the **time** element, just as you did in Exercise 5-2.
5. The content after the header is clearly a simple paragraph. However, the menu has some interesting things going on. It is divided into two conceptual sections (Appetizers and Main Courses), so mark those up as **section** elements. Be careful that the closing section tag (**</section>**) appears before the closing article tag (**</article>**) so the elements are nested correctly and don't overlap. Finally, let's identify the sections with **id** attributes. Name the first one "appetizers" and the second "maincourses."
6. With our sections in place, now we can mark up the content. We're down to **h3** for the headings in each section. Choose the most appropriate list elements to describe the menu item names and their descriptions. Mark up the lists and each item within the lists.
7. Now we can add a few fine details. *Classify* each price as "price" using **span** elements.
8. Two of the dishes are new items. Change the double hyphens to an em-dash character and mark up "new items!" as "strongly important." Classify the title of each new dish as "newitem" (hint, use the existing **dt** element; there is no need to add a **span** this time). This allows us to target menu titles with the "newitem" class and style them differently than other menu items.
9. That takes care of the first article. The second article is already mostly marked up from the previous exercise, but you should mark up the header with the appropriate heading and publication information.
10. So far so good, right? Now make the remaining content that applies to the whole page a **footer**. Mark each line of content within the footer as a paragraph.
11. Let's give the location and hours information some context by putting them in a **div** named "about." Make the labels "Location" and "Hours" appear on a line by themselves by adding line breaks after them. If you'd like, you could also mark up the hours with the **time** element.
12. Finally, copyright information is typically "small print" on a document, so mark it up accordingly. As the final touch, add a copyright symbol after the word "copyright."

Save the file, name it ***bistro_blog.html***, and check your page in a modern browser (remember that IE 8 and earlier won't know what to do with those new HTML5 sectioning elements). How did you do?

Markup tips:

- Choose the element that best fits the meaning of the selected text.
- Don't forget to close elements with closing tags.
- Put all attribute values in quotation marks for clarity
- "Copy and paste" is your friend when adding the same markup to multiple elements. Just be sure what you copied is correct before you paste it throughout the document.

Test Yourself

Were you paying attention? Here is a rapid-fire set of questions to find out.

1. Add the markup to add a thematic break between these paragraphs.

```
<p>People who know me know that I love to cook.</p>

<p>I've created this site to share some of my favorite
recipes.</p>
```

2. What's the difference between a **blockquote** and a **q** element?

3. Which element displays whitespace exactly as it is typed into the source document?

4. What is the difference between a `ul` and an `ol`?

5. How do you remove the bullets from an unordered list? (Be general, not specific.)

6. What element would you use to provide the full name of the W3C (World Wide Web Consortium) in the document? Can you write out the complete markup?

7. What is the difference between a `dl` and a `dt`?

8. What is the difference between `id` and `class`?

9. What is the difference between an `article` and a `section`?

10. Name and write the characters generated by these character entities:

`&mdash;`	_____	`&`	_____
` `	_____	`&copy;`	_____
`&bull;`	_____	`&trade;`	_____

Want More Practice?

Try marking up your own résumé. Start with the raw text, and then add document structure elements, content grouping elements, then inline elements as we've done in Exercise 5-3. If you don't see an element that matches your information just right, try creating one using a `div` or a `span`.

Element Review: Text

The following is a summary of the elements we covered in this chapter. New HTML5 elements are indicated by "(5)." The **data** element is included only in the WHATWG HTML version as of this writing.

Page sections

address	author contact information
article (5)	self-contained content
aside (5)	tangential content (sidebar)
footer (5)	related content
header (5)	introductory content
nav (5)	primary navigation
section (5)	conceptually related group of content

Heading content

h1...h6	headings, levels 1 through 6
hgroup	heading group

Grouping content

blockquote	blockquote
div	generic division
figure (5)	related image or resource
figcaption (5)	text description of a figure
hr	paragraph-level thematic break (horizontal rule)
p	paragraph
pre	preformatted text

List elements

dd	definition
dl	definition list
dt	term
li	list item (for ul and ol)
ol	ordered list
ul	unordered list

Breaks

br	line break
wbr (5)	word break

Phrasing elements

abbr	abbreviation
b	added visual attention (bold)
bdi (5)	possible direction change
bdo	bidirectional override
cite	citation
code	code sample
data (WHATWG)	machine-readable equivalent
del	deleted text
dfn	defining term
em	stress emphasis
i	alternate voice (italic)
ins	inserted text
kbd	keyboard text
mark (5)	highlighted text
q	short inline quotation
ruby (5)	section containing ruby text
rp (5)	parentheses in ruby text
rt (5)	ruby annotations
s	strike-through; incorrect text
samp	sample output
small	annotation; "small print"
span	generic phrase of text
strong	strong importance
sub	subscript
sub	superscript
time (5)	machine-readable time data
u	added attention (underline)
var	variable

ADDING LINKS

If you're creating a page for the Web, chances are you'll want it to point to other web pages and resources, whether on your own site or someone else's. Linking, after all, is what the Web is all about. In this chapter, we'll look at the markup that makes links work: to other sites, to your own site, and within a page. There is one element that makes linking possible: the anchor (**a**).

`<a>...</a>`

Anchor element (hypertext link)

To make a selection of text a link, simply wrap it in opening and closing **`<a>...</a>`** tags and use the **`href`** attribute to provide the URL of the target page. The content of the anchor element becomes the hypertext link. Here is an example that creates a link to the O'Reilly Media website:

```
<a href="http://www.oreilly.com">Go to the O'Reilly Media site</a>
```

To make an image a link, simply put the **`img`** element in the anchor element:

```
<a href="http://www.oreilly.com"><img src="orm.gif" alt="O'Reilly
tarsier logo"></a>
```

Nearly all graphical browsers display linked text as blue and underlined by default. Some older browsers put a blue border around linked images, but most current ones do not. Visited links generally display in purple. Users can change these colors in their browser preferences, and, of course, you can change the appearance of links for your sites using style sheets. I'll show you how in Chapter 13, Colors and Backgrounds.

WARNING

One word of caution: if you choose to change your link colors, keep them consistent throughout your site so as not to confuse your users.

When a user clicks or taps on the linked text or image, the page you specify in the anchor element loads in the browser window. The linked image markup sample shown previously might look like Figure 6-1.

IN THIS CHAPTER

AT A GLANCE

Anchor Syntax

The simplified structure of the anchor element is:

```
<a href="url">linked text
or element</a>
```

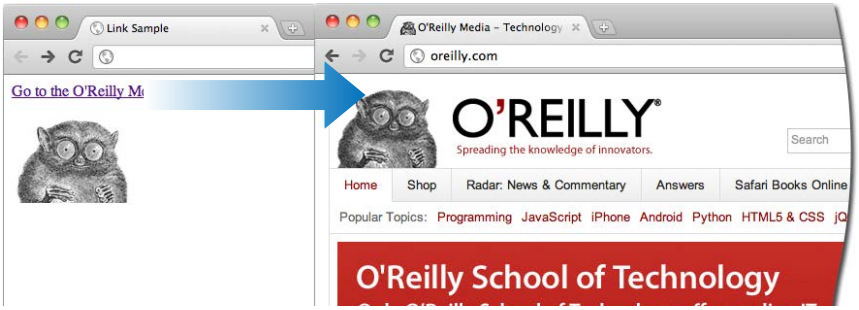


Figure 6-1. When a user clicks or taps on the linked text or image, the page you specified in the anchor element loads in the browser window.

In HTML5, you can put any element in an `a` *element—even block elements!*

Starting in HTML5, you can put any element in an `a` element—even block elements! In the HTML 4.01 spec and earlier, the anchor element could be used for inline content only.

The href Attribute

You'll need to tell the browser which document to link to, right? The `href` (hypertext reference) attribute provides the address of the page or resource (its URL) to the browser. The URL must always appear in quotation marks. Most of the time you'll point to other HTML documents; however, you can also point to other web resources, such as images, audio, and video files.

Because there's not much to slapping anchor tags around some content, the real trick to linking comes in getting the URL correct.

There are two ways to specify the URL:

- **Absolute URLs** provide the full URL for the document, including the protocol (`http://`), the domain name, and the pathname as necessary. You need to use an absolute URL when pointing to a document out on the Web (i.e., not on your own server).

  Example: `href="http://www.oreilly.com/"`

  Sometimes, when the page you're linking to has a long URL pathname, the link can end up looking pretty confusing (Figure 6-2). Just keep in mind that the structure is still a simple container element with one attribute. Don't let the pathname intimidate you.

- **Relative URLs** describe the pathname to a file *relative* to the current document. Relative URLs can be used when you are linking to another document on your own site (i.e., on the same server). It doesn't require the protocol or domain name—just the pathname.

  Example: `href="recipes/index.html"`

In this chapter, we'll add links using absolute and relative URLs to my cooking website, Jen's Kitchen (see Figure 6-3). Absolute URLs are easy, so let's get them out of the way first.

URL Versus URI

The W3C and the development community are moving away from the term URL (Uniform Resource Locator) and toward the more generic and technically accurate URI (Uniform Resource Identifier). On the street and even on the job, however, you're still likely to hear URL.

Here's the skinny on URL versus URI: A URL is one type of a URI that identifies the resource by its location (the L in URL) on the network. The other type of URI is a URN that identifies the resource by name or namespace (the N in URN).

Because it is more familiar, I will be sticking with URL throughout the discussions in this chapter. Just know that URLs are a subset of URIs, and the terms are often used interchangeably.

If you like to geek out on this kind of thing, I refer you to the URI Wikipedia entry: *en.wikipedia.org/wiki/Uniform_resource_identifier*.

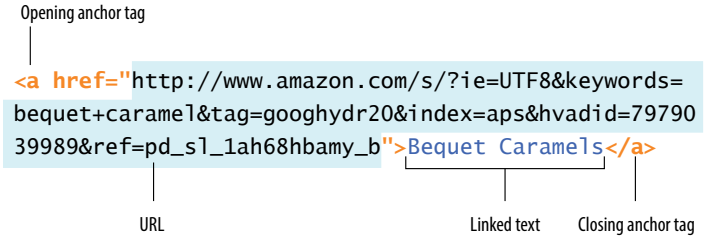


Figure 6-2. An example of a long URL. Although it may make the anchor tag look confusing, the structure is the same.

MARKUP TIP

URL Wrangling

If you are linking to a page with a long URL, it is helpful to copy the URL from the location toolbar in your browser and paste it into your document. That way, you avoid mistyping a single character and breaking the whole link.

Linking to Pages on the Web

Many times, you'll want to create a link to a page that you've found on the Web. This is known as an "external" link because it is going to a page outside of your own server or site. To make an external link, you need to provide the absolute URL, beginning with **http://** (the protocol). This tells the browser, "Go out on the Web and get the following document."

I want to add some external links to the Jen's Kitchen home page (Figure 6-3). First, I'll link the list item "The Food Network" to the site *www.foodtv.com*. I marked up the link text in an anchor element by adding opening and closing anchor tags. Notice that I've added the anchor tags *inside* the list item (**li**) element. That's because only **li** elements are permitted to be children of a **ul** element; placing an **a** element directly inside the **ul** would be invalid HTML.

```
<li><a>The Food Network</a></li>
```

Next, I add the **href** attribute with the complete URL for the site.

```
<li><a href="http://www.foodnetwork.com">The Food Network</a></li>
```

And *voila*! That's all there is to it. Now "The Food Network" will appear as a link and will take my visitors to that site when they click or tap it.

TRY IT

Work Along with Jen's Kitchen

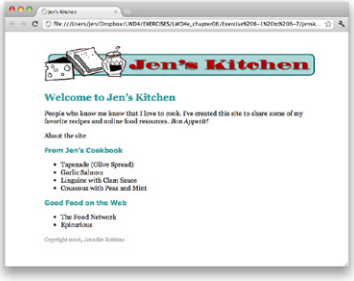


Figure 6-3. Finished Jen's Kitchen page

All the files for the Jen's Kitchen website are available online at *www.learningwebdesign.com/4e/materials*. Download the entire directory, making sure not to change the way its contents are organized.

The resulting markup for all of the exercises is provided in Appendix A.

The pages aren't much to look at, but they will give you a chance to develop your linking skills.

exercise 6-1 | Make an external link

Open the file ***index.html*** from the ***jenskitchen*** folder. Make the list item "Epicurious" link to its web page at *www.epicurious.com*, following my example.

```
<ul>
  <li><a href="http://www.foodnetwork.com/">The Food Network</a>
</li>
  <li>Epicurious</li>
</ul>
```

When you are done, you can save ***index.html*** and open it in a browser. If you have an Internet connection, you can click on your new link and go to the Epicurious site. If the link doesn't take you there, go back and make sure that you didn't miss anything in the markup.

Linking Within Your Own Site

NOTE

On PCs and Macs, files are organized into "folders," but in the web development world, it is more common to refer to the equivalent and more technical term, "directory." A folder is just a directory with a cute icon.

A large portion of the linking you'll do will be between pages of your own site: from the home page to section pages, from section pages to content pages, and so on. In these cases, you can use a relative URL—one that calls for a page on your own server.

Without "http://", the browser looks on the current server for the linked document. A pathname, the notation used to point to a particular file or directory, tells the browser where to find the file. Web pathnames follow the Unix convention of separating directory and filenames with forward slashes (/). A relative pathname describes how to get to the linked document starting from the location of the current document.

Relative pathnames can get a bit tricky. In my teaching experience, nothing stumps beginners like writing relative pathnames, so we'll take it one step at a time. There are exercises along the way that I recommend you do as we go along.

All of the pathname examples in this section are based on the structure of the Jen's Kitchen site shown in Figure 6-4. When you diagram the structure of the directories for a site, it generally ends up looking like an inverted tree with the root directory at the top of the hierarchy. For the Jen's Kitchen site, the root directory is named *jenskitchen*. For another way to look at it, there is also a view of the directory and subdirectories as they appear in the Finder on my Mac (Windows users see one directory at a time).

Important Pathname Don'ts

When you are writing relative pathnames, it is critical that you follow these rules to avoid common errors:

- Don't use backslashes (\). Web URL pathnames use forward slashes (/) only.
- Don't start with the drive name (D:, C:, etc.). Although your pages will link to each other successfully while they are on your own computer, once they are uploaded to the web server, the drive name is irrelevant and will break your links.
- Don't start with file://. This also indicates that the file is local and causes the link to break when it is on the server.

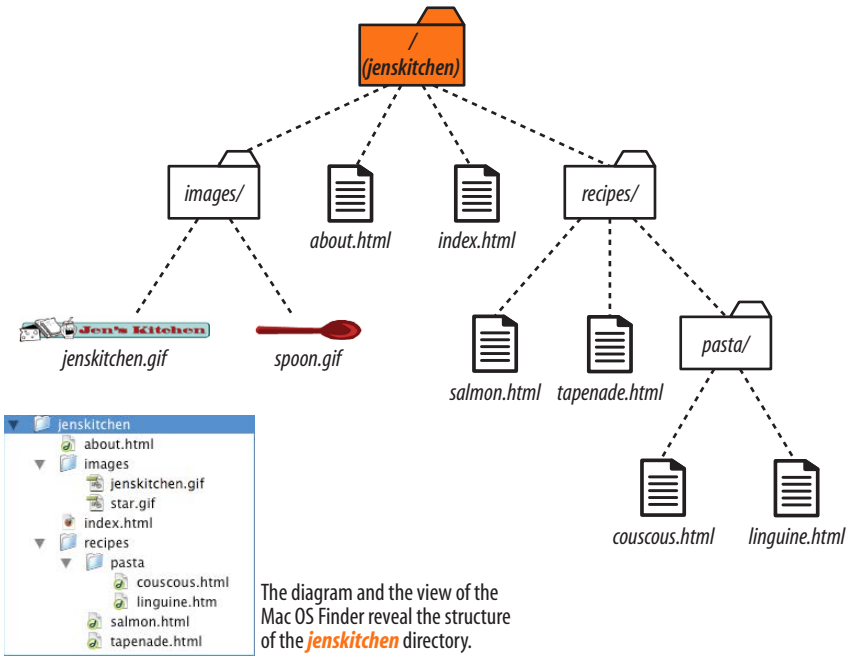


Figure 6-4. A diagram of the jenskitchen site structure.

Linking within a directory

The most straightforward relative URL points to another file within the same directory. When link to a file in the same directory, you only need to provide the name of the file (its filename). When the URL is a single filename, the server looks in the current directory (that is, the directory that contains the document with the link) for the file.

In this example, I want to make a link from my home page (*index.html*) to a general information page (*about.html*). Both files are in the same directory (*jenskitchen*). So from my home page, I can make a link to the information page by simply providing its filename in the URL (Figure 6-5):

A link to just the filename indicates the linked file is in the same directory as the current document.

```
<a href="about.html">About the site...</a>
```

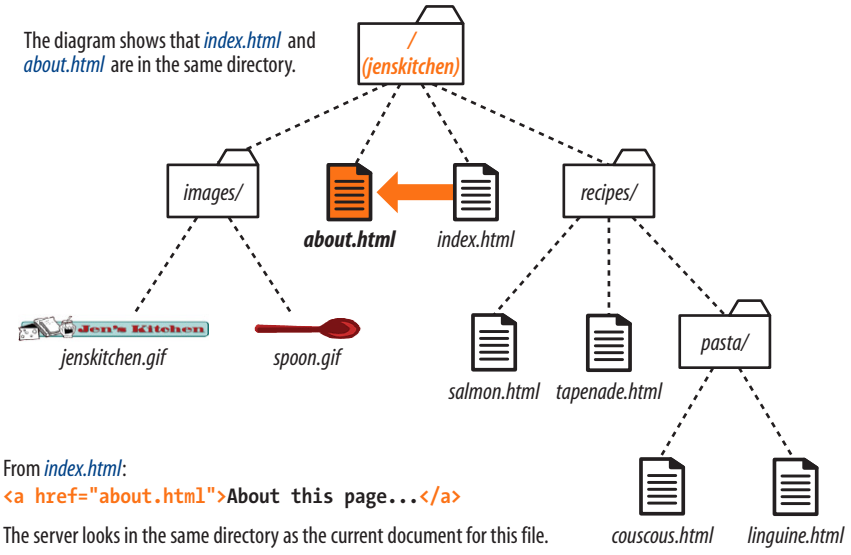


Figure 6-5. Writing a relative URL to another document in the same directory.

exercise 6-2 | Link in the same directory

Open the file ***about.html*** from the ***jenskitchen*** folder. Make the paragraph "Back to the home page" at the bottom of the page link back to ***index.html***. The anchor element should be contained in the **p** element.

```
<p>Back to the home page</p>
```

When you are done, you can save ***about.html*** and open it in a browser. You don't need an Internet connection to test links locally (that is, on your own computer). Clicking on the link should take you back to the home page.

Linking to a lower directory

But what if the files aren't in the same directory? You have to give the browser directions by including the pathname in the URL. Let's see how this works.

Getting back to our example, my recipe files are stored in a subdirectory called *recipes*. I want to make a link from *index.html* to a file in the *recipes* directory called *salmon.html*. The pathname in the URL tells the browser to look in the current directory for a directory called *recipes*, and then look for the file *salmon.html* (Figure 6-6):

```
<li><a href="recipes/salmon.html">Garlic Salmon</a></li>
```

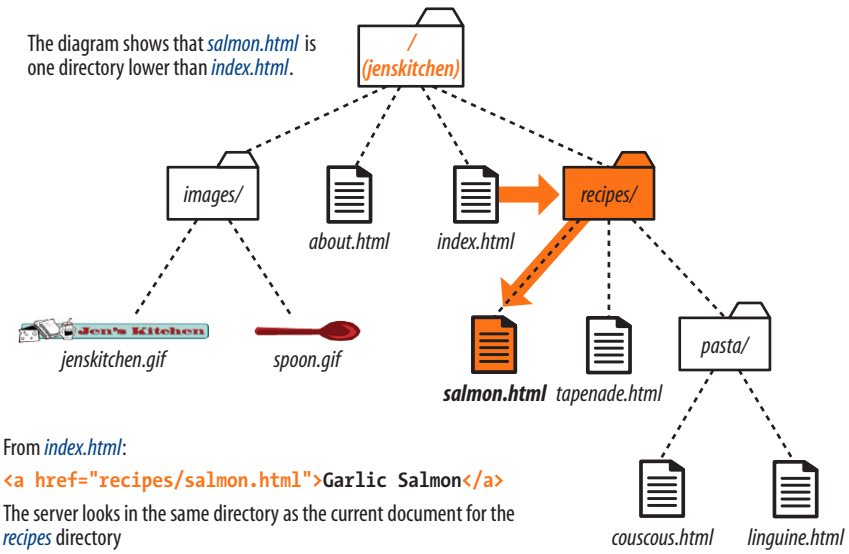


Figure 6-6. *Writing a relative URL to a document that is one directory level lower than the current document.*

exercise 6-3 | Link one directory down

Open the file ***index.html*** from the ***jenskitchen*** folder. Make the list item "Tapenade (Olive Spread)" link to the file ***tapenade.html*** in the ***recipes*** directory. Remember to nest the elements correctly.

```
<li>Tapenade (Olive Spread)</li>
```

When you are done, you can save ***index.html*** and open it in a browser. You should be able to click your new link and see the recipe page for tapenade. If not, make sure that your markup is correct and that the directory structure for ***jenskitchen*** matches the examples.

Now let's link down to the file called *couscous.html*, which is located in the *pasta* subdirectory. All we need to do is provide the directions through two subdirectories (*recipes*, then *pasta*) to *couscous.html* (Figure 6-7):

```
<li><a href="recipes/pasta/couscous.html">Couscous with Peas and Mint
</a></li>
```

When linking to a file in a lower directory, the pathname must contain the names of the subdirectories you go through to get to the file.

Directories are separated by forward slashes. The resulting anchor tag tells the browser, "Look in the current directory for a directory called *recipes*. There you'll find another directory called *pasta*, and in there is the file I'd like to link to, *couscous.html*."

Now that we've done two directory levels, you should get the idea of how pathnames are assembled. This same method applies for relative pathnames that drill down through any number of directories. Just start with the name of the directory that is in same location as the current file, and follow each directory name with a slash until you get to the linked filename.

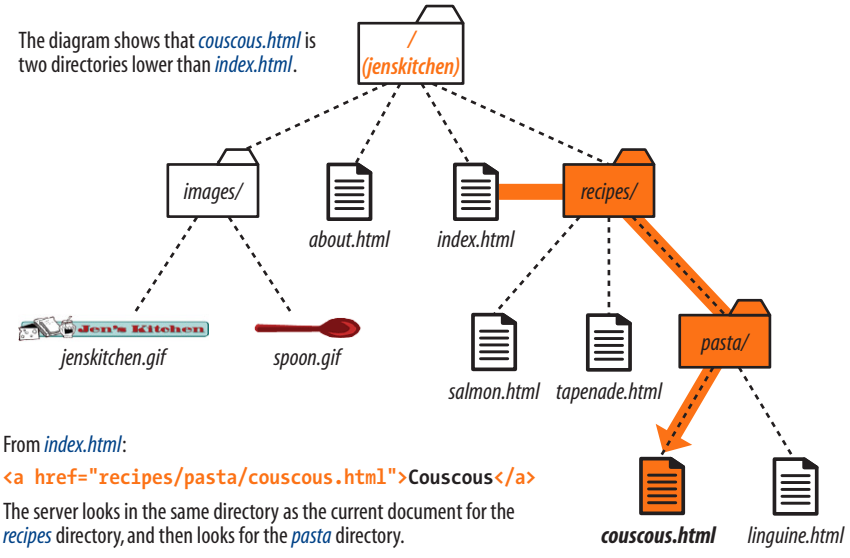


Figure 6-7. *Writing a relative URL to a document that is two directory levels lower than the current document.*

exercise 6-4 | Link two directories down

Open the file ***index.html*** from the ***jenskitchen*** folder. Make the list item "Linguine with Clam Sauce" link to the file ***linguine.html*** in the ***pasta*** directory.

```
<li>Linguine with Clam Sauce</li>
```

When you are done, you can save ***index.html*** and open it in a browser. Click on the new link to get the delicious recipe.

Linking to a higher directory

Each ../ at the beginning of the pathname tells the browser to go up one directory level to look for the file.

So far, so good, right? Here comes the tricky part. This time we're going to go in the other direction and make a link from the salmon recipe page back to the home page, which is one directory level up.

In Unix, there is a pathname convention just for this purpose, the "dot-dot-slash" (`../`). When you begin a pathname with `../`, it's the same as telling the browser "back up one directory level" and then follow the path to the specified file. If you are familiar with browsing files on your desktop, it is helpful to know that a "../" has the same effect as clicking the "Up" button in Windows Explorer or the left-arrow button in the Finder on Mac OS X.

Let's start by making a link from *salmon.html* back to the home page (*index.html*). Because *salmon.html* is in the *recipes* subdirectory, we need to back up a level to *jenskitchen* to find *index.html*. This pathname tells the browser to "go up one level," then look in that directory for *index.html* (Figure 6-8):

```
<p><a href="../index.html">[Back to home page]</a></p>
```

Note that we don't need to write out the name of the higher directory (*jenskitchen*) in the pathname. The `../` stands in for it.

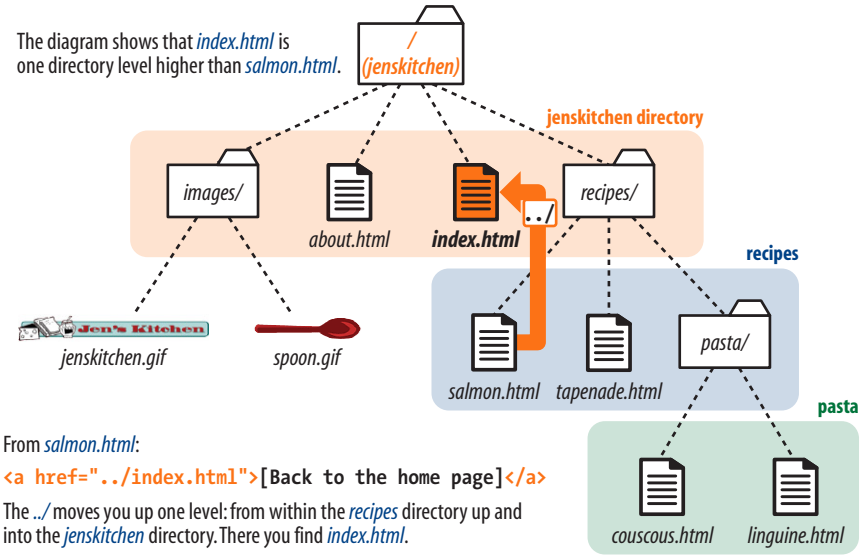


Figure 6-8. Writing a relative URL to a document that is one directory level higher than the current document.

exercise 6-5 | Link to a higher directory

Open the file ***tapenade.html*** from the ***recipes*** directory. At the bottom of the page, you'll find this paragraph:

```
<p>[Back to the home page]</p>
```

Using the notation described in this section, make this text link back to the home page (***index.html***), located one directory level up.

But how about linking back to the home page from *couscous.html*? Can you guess how you'd back your way out of two directory levels? Simple: just use the dot-dot-slash twice (Figure 6-9).

A link on the *couscous.html* page back to the home page (*index.html*) would look like this:

```
<p><a href="../../index.html">[Back to home page]</a></p>
```

The first `../` backs up to the *recipes* directory; the second `../` backs up to the top-level directory where *index.html* can be found. Again, there is no need to write out the directory names; the `../` does it all.

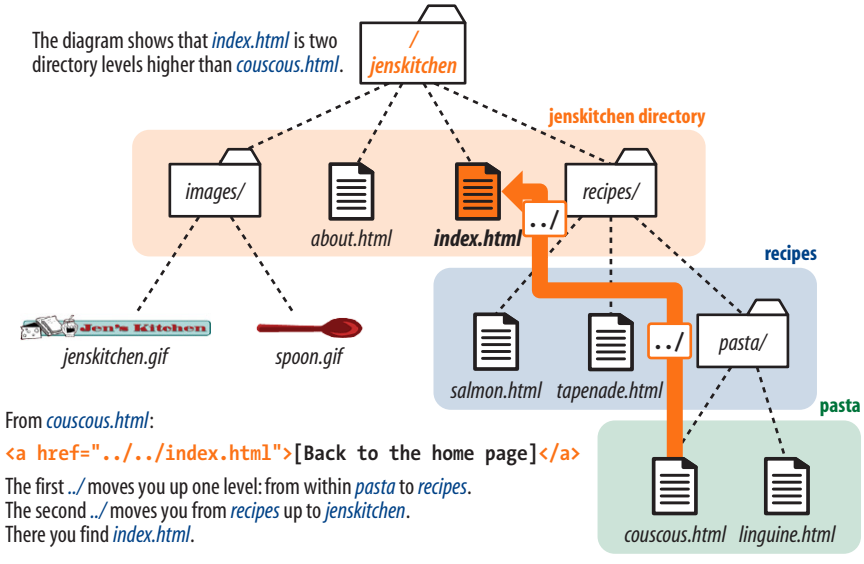


Figure 6-9. *Writing a relative URL to a document that is two directory levels higher than the current document.*

NOTE

I confess to still sometimes silently chanting "go-up-a-level, go-up-a-level" for each `../` when trying to decipher a complicated relative URL. It helps me sort things out.

exercise 6-6 | Link up two directory levels

OK, now it's your turn to give it a try. Open the file ***linguine.html*** and make the last paragraph link to back to the home page using `../../` as I have done.

```
<p>[Back to the home page]</p>
```

When you are done, save the file and open it in a browser. You should be able to link to the home page.

Site root relative pathnames

All websites have a root directory, which is the directory that contains all the directories and files for the site. So far, all of the pathnames we've looked at are relative to the document with the link. Another way to write a relative pathname is to start at the root directory and list the subdirectory names until you get to the file you want to link to. This kind of pathname is known as site root relative.

Site root relative links are generally preferred due to their flexibility.

In the Unix pathname convention, a forward slash (/) at the start of the pathname indicates the path begins at the root directory. The site root relative pathname in the following link reads, "Go to the very top-level directory for this site, open the *recipes* directory, then find the *salmon.html* file" (Figure 6-10):

```
<a href="/recipes/salmon.html">Garlic Salmon</a>
```

Note that you don't need to (and you shouldn't) write the name of the root directory (*jenskitchen*) in the path—just start it with a forward slash (/), and the browser will look in the top-level directory relative to the current file. From there, just specify the directories the browser should look in.

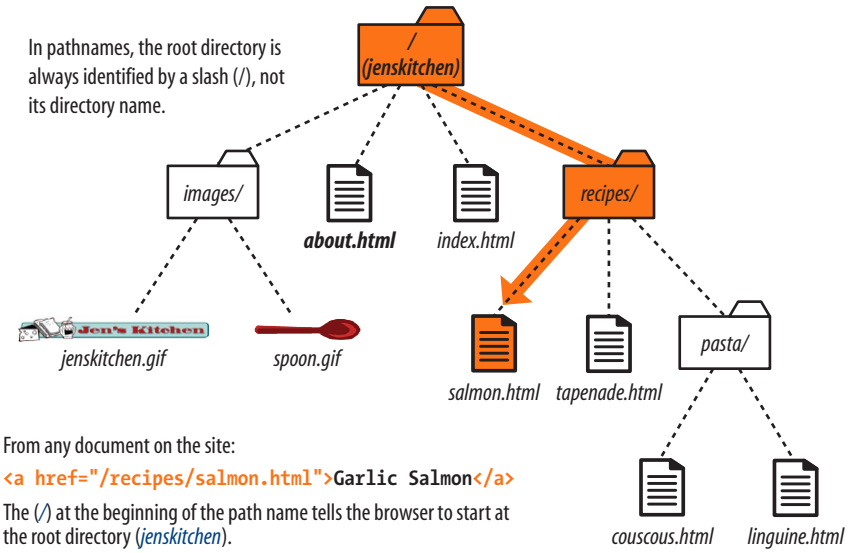


Figure 6-10. Writing a relative URL starting at the root directory.

Because this this type of link starts at the root to describe the pathname, it will work from any document on the server, regardless of which subdirectory it may be located in. Site root relative links are useful for content that might not always be in the same directory, or for dynamically generated material. They also make it easy to copy and paste links between documents.

On the downside, however, the links won't work on your local machine, because they will be relative to your hard drive. You'll have to wait until the site is on the final server to check that links are working.

It's the same for images

The **src** attribute in the **img** element works the same as the **href** attribute in anchors when it comes to specifying URLs. Since you'll most likely be using images from your own server, the **src** attributes within your image elements will be set to relative URLs.

Let's look at a few examples from the Jen's Kitchen site. First, to add an image to the *index.html* page, the markup would be:

```
<img src="images/jenskitchen.gif" alt="">
```

The URL says, "Look in the current directory (*jenskitchen*) for the *images* directory; in there you will find *jenskitchen.gif*."

Now for the *piece de résistance*. Let's add an image to the file *couscous.html*:

```
<img src="../../images/spoon.gif" alt="">
```

This is a little more complicated than what we've seen so far. This pathname tells the browser to go up two directory levels to the top-level directory and, once there, look in the *images* directory for an image called *spoon.gif*. Whew!

Of course, you could simplify that path by going the site root relative route, in which case the pathname to *spoon.gif* (and any other file in the *images* directory) could be accessed like this:

```
<img src="/images/spoon.gif" alt="">
```

The trade-off is that you won't see the image in place until the site is uploaded to the server, but it does make maintenance easier once it's there.

A Little Help from Your Tools

If you use a WYSIWYG authoring tool to create your site, the tool generates relative URLs for you. Programs such as Adobe Dreamweaver and Microsoft Expression Web have built-in site management functions that adjust your relative URLs even if you reorganize the directory structure.

exercise 6-7 | Try a few more

Before we move on, you may want to try your hand at writing a few more relative URLs to make sure you've really gotten it. You can just write your answers on the page, or if you want to test your markup to see whether it works, make changes in the actual files. You'll need to add the text to the files to use as the link (for example, "Go to the Tapenade recipe" for the first question). Answers are in Appendix A.

1. Create a link on ***salmon.html*** to ***tapenade.html***.

   `Go to the Tapenade recipe`

2. Create a link on ***couscous.html*** to ***salmon.html***.

   `Try this with Garlic Salmon.`

3. Create a link on ***tapenade.html*** to ***linguine.html***.

   `Try the Linguine with Clam Sauce`

4. Create a link on ***linguine.html*** to ***about.html.***

   `About Jen's Kitchen`

5. Create a link on ***tapenade.html*** to ***www.allrecipes.com***.

   `Go to Allrecipes.com`

NOTE

Any of the pathnames in Exercise 6-7 could be site root relative, but write them relative to the listed document for the practice.

Linking to a specific point in a page

Did you know you can link to a specific point in a web page? This is useful for providing shortcuts to information at the bottom of a long, scrolling page or for getting back to the top of a page with just one click or tap. Linking to a specific point in the page is also referred to as linking to a document fragment.

Linking to a particular spot within a page is a two-part process. First, you identify the destination, and then you make a link to it. In the following example, I create an alphabetical index at the top of the page that links down to each alphabetical section of a glossary page (Figure 6-11). When users click on the letter "H," they'll jump down on the page to the "H" heading lower on the page.

NOTE

Linking to another spot on the same page works well for long, scrolling pages, but the effect may be lost on a short web page.

Step 1: Identifying the destination

AUTHORING TIP

To the Top!

It is common practice to add a link back up to the top of the page when linking into a long page of text. This alleviates the need to scroll back after every link.

I like to think of this step as planting a flag in the document so I can get back to it easily. To create a destination, use the **id** attribute to give the target element in the document a unique name (that's "unique" as in the name may appear only once in the document, not "unique" as in funky and interesting). In web lingo, this is the fragment identifier.

You may remember the **id** attribute from Chapter 5, Marking Up Text, where we used it to name generic **div** and **span** elements. Here, we're going to use it to name an element so that it can serve as a fragment identifier—that is, the destination of a link.

NOTE

Remember that `id` *values must start with a letter or an underscore (although underscores may be problematic in some versions of IE).*

Here is a sample of the source for the glossary page. Because I want users to be able to link directly to the "H" heading, I'll add the **id** attribute to it and give it the value "startH" (Figure 6-11 ❶).

```
<h1 id="startH">H</h1>
```

Step 2: Linking to the destination

With the identifier in place, now I can make a link to it.

At the top of the page, I'll create a link down to the "startH" fragment ❷. As for any link, I use the **a** element with the **href** attribute to provide the location of the link. To indicate that I'm linking to a fragment, I use the octothorpe symbol (#), also called a hash or number symbol, before the fragment name.

```
<p>... F | G | <a href="#startH">H</a> | I | J ...</p>
```

Fragment names are preceded by an octothorpe symbol (#).

And that's it. Now when someone clicks on the "H" from the listing at the top of the page, the browser will jump down and display the section starting with the "H" heading ❸.

1. Identify the destination using the `id` attribute.

```
<h2 id="startH">H</h2>
<dl>
<dt>hexadecimal</dt>
<dd>A base-16 numbering system that uses the characters 0-9 and
A-F. It is used in CSS and HTML for specifying color values</dd>
```

2. Create a link to the destination. The `#` before the name is necessary to identify this as a fragment and not a filename.

```
<p>... | F | G | <a href="#startH">H</a> | I | J ...</p>
```

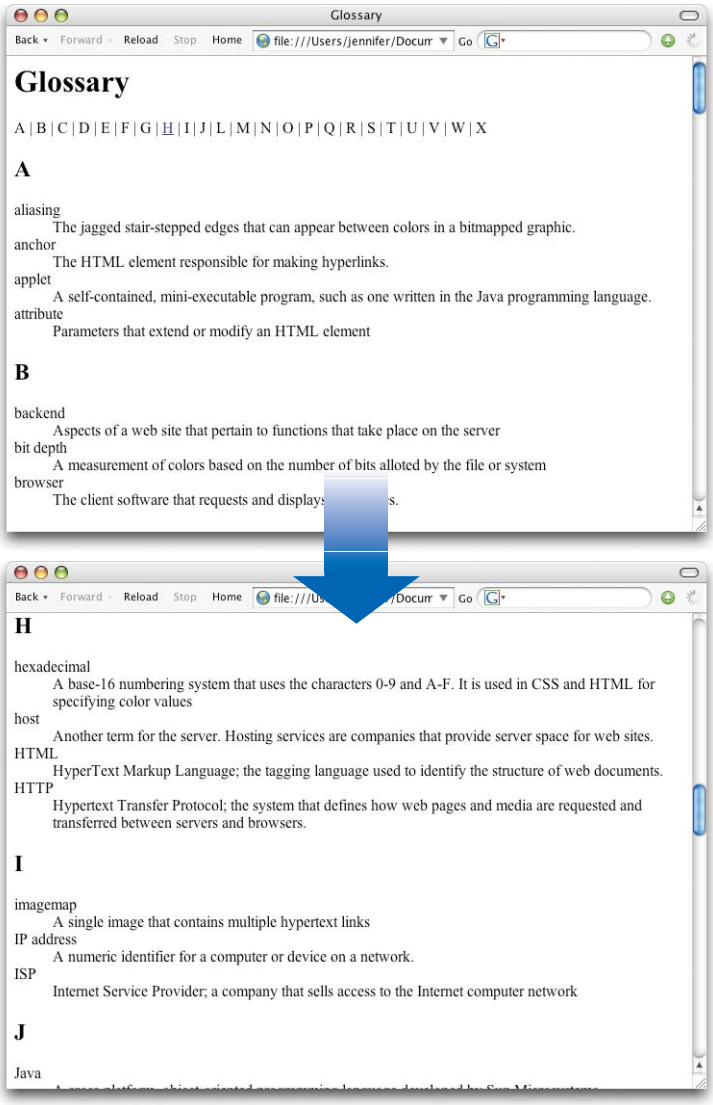
3. 

Figure 6-11. Linking to a specific destination within a single web page.

NOTE

Some developers help their brothers and sisters out by proactively adding `id`*s as anchors at the beginning of any thematic section of content (within a reasonable level, and depending on the site). That way other people can link back to any section in your content.*

exercise 6-8 | Linking to a fragment

Want some practice linking to specific destinations? Open the file ***glossary.html*** in the ***materials*** folder for this chapter. It looks just like the document in Figure 6-11.

1. Identify the **h2** "A" as a destination for a link by naming it "startA" with an **id** attribute.

   ```
   <h2 id="startA">A</h2>
   ```

2. Make the letter "A" at the top of the page a link to the identified fragment. Don't forget the **#**.

   ```
   <a href="#startA">A</a>
   ```

Repeat steps 1 and 2 for every letter across the top of the page until you really know what you are doing (or until you can't stand it anymore). You can help users get back to the top of the page, too.

3. Make the heading "Glossary" a destination named "top."

   ```
   <h1 id="top">Glossary</h1>
   ```

4. Add a paragraph element containing "TOP" at the end of each lettered section. Make "TOP" a link to the identifier that you just made at the top of the page.

   ```
   <p><a href="#top">TOP</a></
       p>
   ```

Copy and paste this code to the end of every letter section. Now your readers can get back to the top of the page easily throughout the document.

Linking to a fragment in another document

You can link to a fragment in another document by adding the fragment name to the end of the URL (absolute or relative). For example, to make a link to the "H" heading of the glossary page from another document in that directory, the URL would look like this:

```
<a href="glossary.html#startH">See the Glossary, letter H</a>
```

You can even link to specific destinations in pages on other sites by putting the fragment identifier at the end of an absolute URL, like so:

```
<a href="http://www.example.com/glossary.html#startH">See the Glossary,
letter H</a>
```

Of course, you don't have any control over the named fragments in other people's web pages (see the note). The destination points must be inserted by the author of those documents in order for them to be available to you. The only way to know whether they are there and where they are is to "View Source" for the page and look for them in the markup. If the fragments in external documents move or go away, the page will still load; the browser will just go to the top of the page as it does for regular links.

Targeting a New Browser Window

One problem with putting links on your page is that when people click on them, they may never come back. The traditional solution to this dilemma has been to make the linked page open in a new browser window. That way, your visitors can check out the link and still have your content available where they left it.

Before I provide the instructions for how to do it, I am going to strongly advise against it. First of all, tabbed browsers make it somewhat less likely that users will never find their way back to the original page. Furthermore, opening new windows is problematic for accessibility. New windows may be confusing to some users, particularly those who are accessing your site via a screen reader or even on a small-screen device. At the very least, new windows may be perceived as an annoyance rather than a convenience. Because it is common to configure your browser to block pop-up windows, you risk having the users miss out on the content in the new window altogether.

So consider carefully whether you need a separate window at all, and I'll tell you how in case you have a very good reason to do it. The method you use to open a link in a new browser window depends on whether you want to control its size. If the size of the window doesn't matter, you can use HTML markup alone. However, if you want to open the new window with particular pixel dimensions, then you need to use JavaScript.

A new window with markup

To open a new window using markup, use the **target** attribute in the anchor (**a**) element to tell the browser the name of the window in which you want the linked document to open. Set the value of target to **_blank** or to any name of your choosing. Remember that with this method, you have no control over the size of the window, but it will generally open as a new tab or in a new window the same size as the most recently opened window in the user's browser.

Setting **target="_blank"** always causes the browser to open a fresh window. For example:

```
<a href="http://www.oreilly.com" target="_blank">O'Reilly</a>
```

If you target "_blank" for every link, every link will launch a new window, potentially leaving your user with a mess of open windows.

A better method is to give the target window a specific name, which can then be used by subsequent links. You can give the window any name you like ("new," "sample," whatever), as long as it doesn't start with an underscore. The following link will open a new window called "display":

```
<a href="http://www.oreilly.com" target="display">O'Reilly</a>
```

If you target the "display" window from every link on the page, each linked document will open in the same second window. Unfortunately, if that second window stays hidden behind the user's current window, it may look as though the link simply didn't work.

Pop-up windows

It is possible to open a window with specific dimensions and various parts of the browser chrome (toolbars, scrollbars, etc.) turned on or off; however, it takes JavaScript to do it. These are known as pop-up windows, and they are commonly used for advertising. In fact, they've become such a nuisance that many browsers have preferences for turning them off completely. Furthermore, in a world where sites are accessed on small, mobile devices, popping up windows at specific pixel dimensions has no place.

That said, if you have a valid reason to open a new browser window at a specific size, I recommend this tutorial article by Peter-Paul Koch at Quirksmode: *www.quirksmode.org/js/popup.html*.

Mail Links

Here's a nifty little linking trick: the **mailto** link. By using the **mailto** protocol in a link, you can link to an email address. When the user clicks on a **mailto** link, the browser opens a new mail message preaddressed to that address in a designated mail program.

A sample **mailto** link is shown here:

```
<a href="mailto:alklecker@example.com">Contact Al Klecker</a>
```

As you can see, it's a standard anchor element with the **href** attribute. But the value is set to **mailto:***name@address.com*.

The browser has to be configured to launch a mail program, so the effect won't work for 100% of your audience. If you use the email itself as the linked text, nobody will be left out if the **mailto** function does not work (a nice little example of progressive enhancement).

Spam-Bots

Be aware that by putting an email address in your document source, you will make it susceptible to receiving unsolicited junk email (known as spam). People who generate spam lists sometimes use automated search programs (called bots) to scour the Web for email addresses.

If you want your email to display on the page in a way that humans can figure it out but robots can't, you can deconstruct the address in a way that is still understandable to people, for example, "jen [-at-] oreilly [dot] com."

That trick won't work in a **mailto** link, because the accurate email address must be provided as an attribute value. One solution is to encrypt the email address using JavaScript. The Enkoder Form at Hivelogic (*hivelogic.com/enkoder/*) does this for you. Simply enter the link text and the email address, and Enkoder generates code that you can copy and paste into your document.

Otherwise, if you don't want to risk getting spammed, keep your email address out of your HTML document.

Telephone Links

Keep in mind that the smartphones people are using to access your website can also be used to make phone calls! Why not save your visitors a step by letting them dial a phone number on your site simply by tapping on it on the page? The syntax uses the **tel:** scheme and is very simple.

```
<a href="tel:+18005551212">Call us free at (800) 555-1212</a>
```

When mobile users tap the link, they get an alert box asking them to confirm that they'd like to call the number. This feature is supported on most mobile devices, including iOS, Android, Blackberry, Symbian, Internet Explorer, and Opera Mini. The iPad and iPod Touch can't make a call, but they will offer to create a new contact from the number. Nothing happens when desktop users click the link. If that bothers you, you could use a CSS rule that hides the link for non-mobile devices (unfortunately, that is beyond the scope of this discussion).

There are a few best practices for using telephone links:

- It is recommended that you include the full international dialing number, including the country code, for the **tel:** value because there is no way of knowing where the user will be accessing your site.
- Also include the telephone number in the content of the link so that if the link doesn't work, the telephone number is still available.
- Android and iPhone have a feature that detects phone numbers and automatically turns them into links. Unfortunately, some 10-digit numbers that are not telephone numbers might get turned into links, too. If your document has strings of numbers that might get confused as phone numbers, you can turn auto-detection off by including the following **meta** element in the **head** of your document.

  ```
  <meta name="format-detection" content="telephone=no">
  ```

  For Blackberry devices, use the following:

  ```
  <meta http-equiv="x-rim-auto-match" content="none">
  ```

Test Yourself

The most important lesson in this chapter is how to write URLs for links and images. Here's another chance to brush up on your pathname skills.

Using the directory hierarchy shown in Figure 6-12, write out the markup for the following links and graphics. I filled in the first one for you as an example. The answers are located in Appendix A.

This diagram should provide you with enough information to answer the questions. If you need hands-on work to figure them out, the directory structure is available in the *test* directory in the materials for this chapter. The documents are just dummy files and contain no content.

TIP

The **../** (or multiples of them) always appears at the beginning of the pathname and never in the middle. If the pathnames you write have **../** in the middle, you've done something wrong.

1. In *index.html* (the site's home page), write the markup for a link to *tutorial.html*.

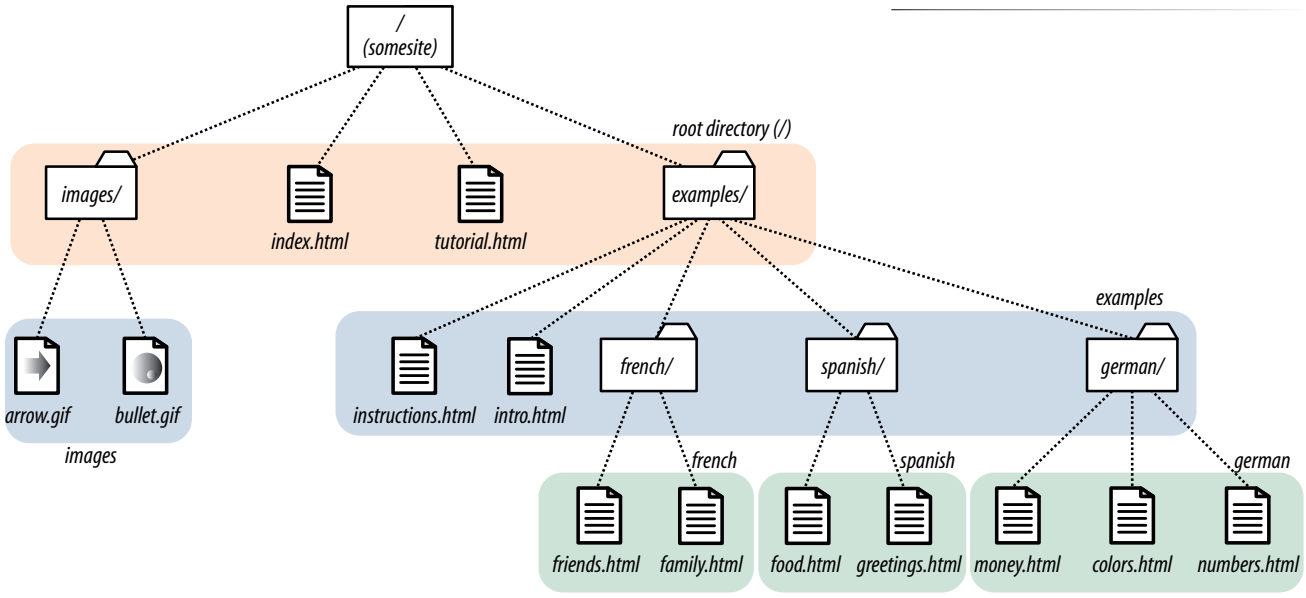
```
<a href="tutorial.html">...</a>
```

2. In *index.html*, write the anchor element for a link to *instructions.html*.

3. Create a link to *family.html* from the page *tutorial.html*.

4. Create a link to *numbers.html* from the *family.html* page, but this time, start with the root directory.

Figure 6-12. The directory structure for the Test Yourself questions.



5. Create a link back to the home page (*index.html*) from the page *instructions.html*.

6. In the file *intro.html*, create a link to the website for this book (*www.learningwebdesign.com/4e/materials*).

7. Create a link to *instructions.html* from the page *greetings.html*.

8. Create a link back to the home page (*index.html*) from *money.html*.

We haven't covered the image (`img`) element in detail yet, but you should be able to fill in the relative URLs after the `src` attribute to specify the location of the image files for these examples.

9. To place the graphic *arrow.gif* on the page *index.html*, the URL is:

```
<img src="                              " alt="">
```

10. To place the graphic *arrow.gif* on the page *intro.html*, the URL is:

```
<img src="                              " alt="">
```

11. To place the graphic *bullet.gif* on the *friends.html* page, the URL is:

```
<img src="                              " alt="">
```

Element Review: Links

There's really only one element relevant to creating hypertext links:

Element and attributes	Description
a	Anchor (hypertext link) element
`href="url"`	Location of the target file

ADDING IMAGES

IN THIS CHAPTER

Adding images to a web page

Using the `src`, `alt`, `width`, and `height` attributes

A web page with all text and no pictures isn't much fun. The Web's explosion into mass popularity was due in part to the fact that there were images on the page. Before images, the Internet was a text-only tundra.

Images appear on web pages in two ways: embedded in the inline content or as background images. Background images are added using Cascading Style Sheets and are talked about at length in Chapter 13, Colors and Backgrounds. With the emergence of standards-driven design and its mission to keep all matters of presentation out of the document structure, there has been a shift away from using inline images for purely decorative purposes. See the sidebar Images Move to the Background on the following page for more information on this trend.

In this chapter, we'll focus on embedding image content into the document using the **`img`** element. Use the **`img`** element when the image is the content, such as product shots, gallery images, ads, illustrations, and so on...I think you get the idea.

First, a Word on Image Formats

We'll get to the **`img`** element and markup examples in a moment, but first it's important to know that you can't put just any image on a web page. In order to be displayed inline, images must be in the GIF, JPEG, or PNG file format. Chapter 21, Web Graphics Basics explains these formats and the image types they handle best. In addition to being in an appropriate format, image files need to be named with the proper suffixes—*.gif*, *.jpg* (or *.jpeg*), and *.png*, respectively—in order to be recognized by the browser.

If you have a source image that is in another popular format, such as TIFF, BMP, or EPS, you'll need to convert it to a web format before you can add it to the page. If, for some reason, you must keep your graphic file in its original format (for example, a file for a CAD program or an image in a vector format), you can make it available as an external image by making a link directly to the image file, like this:

```
<a href="architecture.eps">Get the drawing</a>
```

Browsers use helper applications to display media they can't handle alone. The browser matches the suffix of the file in the link to the appropriate helper application. The external image may open in a separate application window or within the browser window if the helper application is a plug-in, such as the QuickTime plug-in. The browser may also ask the user to save the file or open an application manually. It is also possible that it won't be able to be opened at all.

Without further ado, let's take a look at the **img** element and its required and recommended attributes.

Images Move to the Background

Images that are used purely for decoration have more to do with presentation than document structure and content. For that reason, they should be controlled with a style sheet rather than the markup.

Using CSS, it is possible to place an image in the background of the page or in any text element (a **div**, **h1**, **li**… you name it). These techniques are introduced in Chapter 13, Colors and Backgrounds.

There are several benefits to specifying decorative images only in an external style sheet and keeping them out of the document structure. Not only does it make the document cleaner and more accessible, but it also makes it easier to make changes to the look and feel of a site than when presentational elements are interspersed in the content.

For inspiration on how visually rich a web page can be with no **img** elements at all, look at the examples in the "Select a Design" section of the CSS Zen Garden site at *www.csszengarden.com*.

The img Element

`<img>`

Adds an inline image

The **img** element tells the browser, "Place an image here." You've already gotten a glimpse of it used to place banner graphics in the examples in Chapters 4 and 5. You can also place an image element right in the flow of the text at the point where you want the image to appear, as in the following example. Images stay in the flow of text and do not cause any line breaks (HTML5 calls this a phrasing element), as shown in Figure 7-1.

```
<p>I had been wanting to go to Tuscany <img src="tuscany.jpg" alt="">
 for a long time, and I was not disappointed.</p>
```

Figure 7-1. By default, images are aligned with the baseline of the surrounding text, and they do not cause a line break.

When the browser sees the **img** element, it makes a request to the server and retrieves the image file before displaying it on the page. On a fast network with a fast computer, even though a separate request is made for each image file, the page usually appears to arrive instantaneously. On mobile devices with slow network connections, we may be well aware of the wait for images to be fetched one at a time. The same is true for users still using dial-up Internet connections or other slow networks, like the expensive WiFi at luxury hotels.

When designing mobile web experiences, it is wise to limit the number of server requests in general, which means carefully considering the number of images on the page.

The **src** and **alt** attributes shown in the sample are required. The **src** attribute tells the browser the location of the image file. The **alt** attribute provides alternative text that displays if the image is not available. We'll talk about **src** and **alt** a little more in upcoming sections.

The src and alt attributes are required in the img element.

There are a few other things of note about the **img** element:

- It is an empty element, which means it doesn't have any content. You just place it in the flow of text where the image should go.
- If you choose to write in the stricter XHTML syntax, you need to terminate (close) the empty **img** element with a slash like so: **<img />**.
- It is an inline element, so it behaves like any other inline element in the text flow. Figure 7-2 demonstrates the inline nature of image elements. When the browser window is resized, the line of images reflows to fill the new width.
- The **img** element is what's known as a replaced element because it is replaced by an external file when the page is displayed. This makes it different from text elements that have their content right there in the source (and thus are non-replaced).
- By default, the bottom edge of an image aligns with the baseline of text, as shown in Figures 7-1 and 7-2. Using CSS, you can float the image to the right or left margin and allow text to flow around it, control the space and borders around the image, and change its vertical alignment. We'll talk about those styles in Part III.

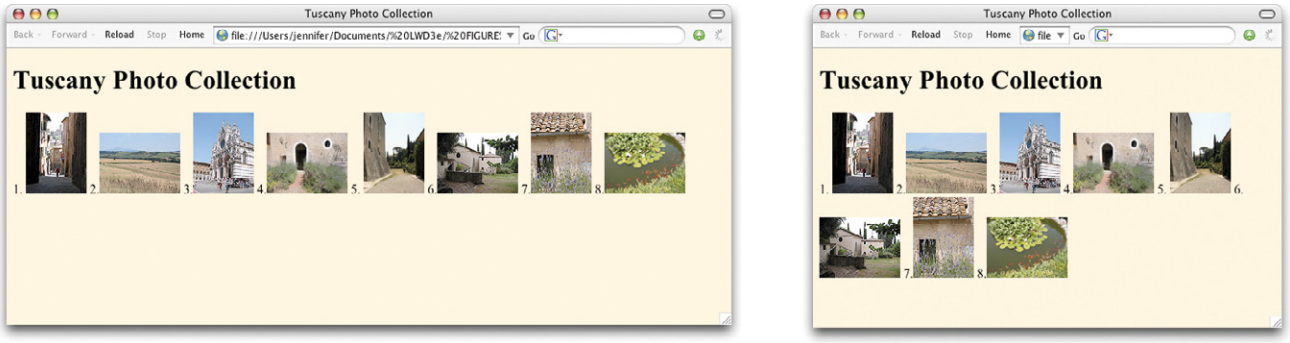


Figure 7-2. Inline images are part of the normal document flow. They reflow when the browser window is resized.

Providing the location with src

`src="URL"`

Source (location) of the image

The value of the **src** attribute is the URL of the image file. In most cases, the images you use on your pages will reside on your server, so you will use

relative URLs to point to them. If you just read Chapter 6, Adding Links, you should be pretty handy with writing relative URLs by now. In short, if the image is in the same directory as the HTML document, you can just refer to the image by name in the **src** attribute:

```
<img src="icon.gif" alt="">
```

Developers usually organize the images for a site into a directory called *images*, *assets*, or *graphics*. There may even be separate image directories for each section of the site. If an image is not in the same directory as the document, you need to provide the pathname to the image file.

```
<img src="/images/arrow.gif" alt="">
```

Of course, you can place images from other websites as well (just be sure that you have permission to do so). Just use an absolute URL, like this:

```
<img src="http://www.example.com/images/smile.gif" alt="">
```

TIP

Take Advantage of Caching

Here's a tip for making images display more quickly and reducing the traffic to your server. If you use the same image in multiple places on your site, be sure each **img** element is pointing to the same image file on the server.

When a browser downloads an image file, it stores it in the disk cache (a space for temporarily storing files on the hard disk). That way, if it needs to redisplay the page, it can just pull up a local copy of the source document and image files without making a new trip out to the remote server.

When you use the same image repetitively in a page or a site, the browser only needs to download the image once. Every subsequent instance of the image is grabbed from the local cache, which means less traffic for the server and faster display for the end user.

The browser recognizes an image by its entire pathname, not just the filename, so if you want to take advantage of file caching, be sure that each instance of your image is pointing to the same image file on the server, not multiple copies of the same image file in different directories.

Providing alternate text with alt

`alt="text"`

Alternative text

Every **img** element must also contain an **alt** attribute that is used to provide a brief description of the image for those who are not able to see it, such as users with screen readers, braille, or even small mobile devices. Alternate text (also referred to as alt text) should serve as a substitute for the image content—serving the same purpose and presenting the same information.

```
<p>If you're <img src="happyface.gif" alt="happy"> and you know it clap
    your hands.</p>
```

A screen reader might indicate the image by reading its **alt** value this way:

> *"If you're image happy and you know it clap your hands."*

If an image does not add anything meaningful to the text content of the page, it is recommended that you leave the value of the **alt** attribute empty, as shown in the following example and other examples in this chapter (you may also consider whether it is more appropriately handled as a background image with CSS, but I digress). Note that there is no character space between the quotation marks.

```
<img src="logo.gif" alt="">
```

Do not omit the **alt** attribute altogether, however, because it will cause the document to be invalid (validating documents is covered in Chapter 3, Some Big Concepts You Need to Know). For each inline image on your page, consider what the alternative text would sound like when read aloud and whether that enhances or is just obtrusive to a screen-reader user's experience.

Alternative text may benefit users with graphical browsers as well. If a user has opted to turn images off in the browser preferences or if the image simply fails to load, the browser may display the alternative text to give the user

an idea of what is missing. The handling of alternative text is inconsistent among modern browsers, however, as shown in Figure 7-3.

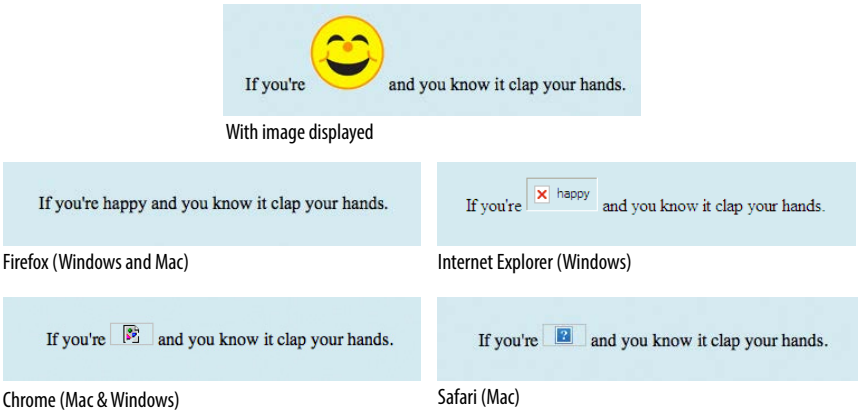


Figure 7-3. Most browsers display alternative text in place of the image (either with an icon or as inline text) if the image is not available. Safari for Macintosh OS X is a notable exception.

Image Accessibility

Images and other non-text content are a challenge for users accessing the Web with screen readers. Alternative text allows you to provide a short description of what is in an image for those who can't see it. However, there are some types of images, such as data charts and diagrams, that require longer descriptions than are practical as an **alt** value.

For extremely long descriptions, consider writing the description elsewhere on the page or in a separate document and making a reference or link to it near the image.

HTML 4.01 included the **longdesc** (long description) attribute, but it was dropped in HTML5 due to lack of support. The **longdesc** attribute points to a separate HTML document containing a lengthy description of the image, as in this example:

```
<img src="executivesalaries.png" alt="Executive salaries 1999-2009"
longdesc="salaries-ld.html">
```

In HTML5, the **figcaption** element allows a long description of an image when it is placed in a **figure**.

There is more to say about image accessibility than I can fit in this chapter. I encourage you to start your research with these resources:

- "Creating Accessible Images" at WebAIM (*webaim.org/techniques/images/longdesc*) provides alternatives to the **longdesc** attribute.
- "Chapter 6, The Image Problem" from the book *Building Accessible Websites* by Joe Clark (*joeclark.org/book/sashay/serialization/Chapter06.html*)
- The Web Content Accessibility Guidelines (WCAG 2.0) at the W3C include techniques for improving accessibility across all web content (*www.w3.org/TR/WCAG20-TECHS/*). Warning: it's pretty dense.

NOTE

Serving different image files for an `img` *element based on device size is handled by JavaScript or a program running on the server. It is beyond the scope of this chapter, but see the Responsive Images sidebar in Chapter 18, CSS Techniques.*

TIP

Using a Browser to Find Pixel Dimensions

You can find the pixel dimensions of an image by opening it in an image editing program, of course, but did you know you can also use a web browser?

Using Chrome, Firefox, or Safari (but, sorry, not Internet Explorer), simply open the image file, and its pixel dimensions display in the browser's title bar along with the filename. It's a handy shortcut I use all the time because I always seem to have a browser running.

Providing width and height dimensions

`width="number"`

Image width in pixels

`height="number"`

Image height in pixels

The **width** and **height** attributes indicate the dimensions of the image in number of pixels. Sounds mundane, but these attributes can speed up the time it takes to display the final page by seconds. Browsers use the specified dimensions to hold the right amount of space in the layout while the images are loading rather than reconstructing the page each time a new image arrives.

And that's great if you are designing one version of your page with one fixed image size. However, in these days of responsive web design, it is common to create several versions of the same image and send a small one to small mobile devices and a larger image for large-screen devices (and rescale the images to fit for sizes in between). If you are scaling images in a responsive design or delivering multiple image sizes, do not use width and height attributes in the markup.

With this caveat in mind, let's look at how **width** and **height** work for those cases when it is appropriate to use them.

Match values with actual pixel size

Be sure that the pixel dimensions you specify are the actual dimensions of the image. If the pixel values differ from the actual dimensions of your image, the browser resizes the image to match the specified values (Figure 7-4).

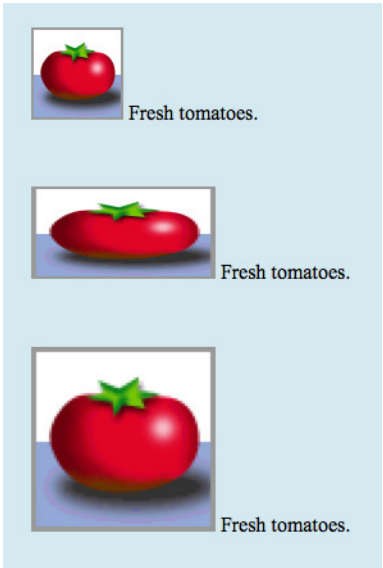


Figure 7-4. Browsers resize images to match the provided `width` *and* `height` *values. It is strongly recommended not to resize images in this way*

Although it may be tempting to resize images in this manner, you should avoid doing so. Even though the image may appear small on the page, the large image with its corresponding large file size still needs to download. It is better to resize the image in an image-editing program and then place it at actual size on the page. Not only that, but resizing with attributes usually results in a blurry and deformed image. In fact, if your images ever look fuzzy when viewed in a browser, the first thing to check is that the **width** and **height** values match the dimensions of the image exactly.

.

exercise 7-1 | **Adding and linking images**

You're back from Italy and it's time to post about some of your travels. In this exercise, you'll add thumbnail images to a travelog and make them link to pages with full-sized versions.

All the thumbnails and photos you need have been created for you, and I've given you a head start on the HTML files as well. Everything is available at *www.learningwebdesign.com/4e/materials.* Put a copy of the ***tuscany*** folder on your hard drive, making sure to keep it organized as you find it. As always, the resulting markup is listed in Appendix A.

This little site is made up of a main page (***index.html***) and three separate HTML documents containing each of the larger image views (Figure 7-5). First, we'll add the thumbnails, and then we'll add the full-size versions to their respective pages. Finally, we'll make the thumbnails link to those pages. Let's get started. Open the file ***index.html***, and add the small thumbnail images to this page to accompany the text. I've done the first one for you:

```
<h2>Pozzarello</h2>
<p><img src="thumbnails/window_thumb.jpg"
alt="view from bedroom window" width="75"
height="100"> The house we stayed in was called
Pozzarello…
```

I've put the image at the beginning of the paragraph, just after the opening **<p>** tag. Because all of the thumbnail images are located in the *thumbnails* directory, I provided the pathname in the URL. I also added a description of the image and the width and height dimensions in pixels (px).

Now it's your turn. Add the image ***countryside_thumb.jpg*** (100px wide x 75px tall) and ***sienna_thumb.jpg*** (75 x 100) at the beginning of the paragraphs in their respective sections. Be sure to include the pathname, an alternative text description, and pixel dimensions.

When you are done, save the file and then open it in the browser to be sure that the images are visible and appear at the right size.

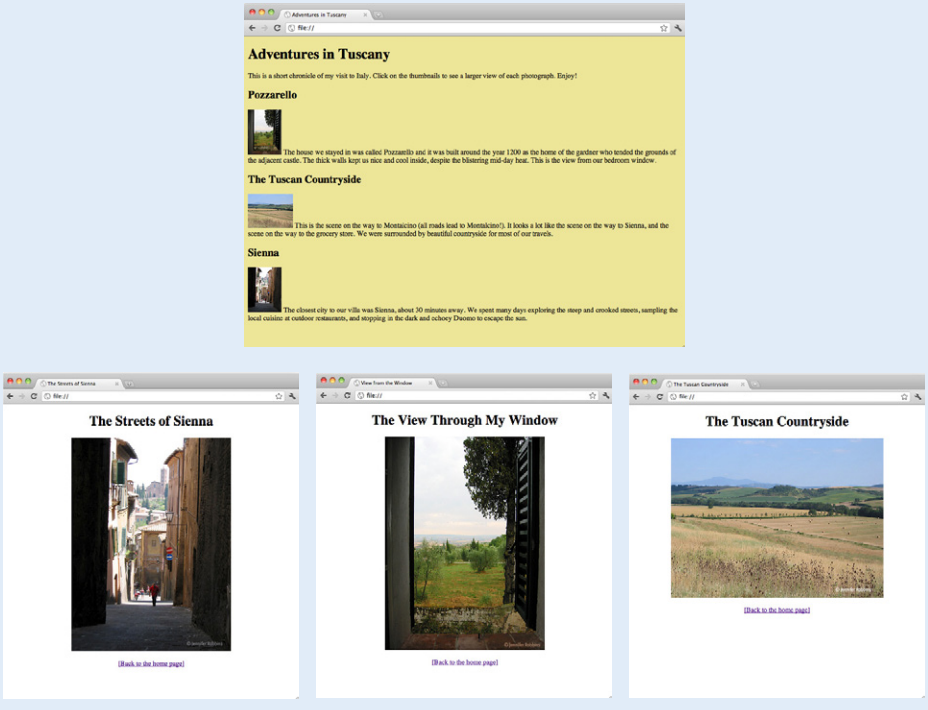


Figure 7-5. Travel photo site.

1. Next, add the images to the individual HTML documents. I've done ***window.html*** for you:

   ```
   <h1>The View Through My Window</h1>
   <p><img src="photos/window.jpg" alt="view out the window of the rolling Tuscan hills" width="375" height="500"></p>
   ```

   Notice that the full-size images are in a directory called ***photos***, so that needs to be reflected in the pathnames.

   Add images to ***countryside.html*** and ***sienna.html***, following my example. Hint: all of the images are 500 pixels on their widest side and 375 pixels on their shortest side, although the orientation varies.

   Save each file, and check your work by opening them in the browser window.

2. Back in ***index.html***, link the thumbnails to their respective files. I've done the first one here.

   ```
   <h2>Pozzarello</h2>
   <p><a href="window.html"><img src="thumbnails/window_thumb.jpg" alt="view from the bedroom window" width="75" height="100"></a></p>
   ```

   Notice that the URL is relative to the current document (***index.html***), not to the location of the image (the ***thumbnails*** directory).

   Make the remaining thumbnail images links to each of the documents. If all the images are visible and you are able to link to each page and back to the home page again, then congratulations, you're done!

Like a little more practice?

If you'd like more practice, you'll find three additional images (***sweets.jpg***, ***cathedral.jpg***, and ***lavender.jpg***) with their thumbnail versions (***sweets_thumb.jpg***, ***cathedral_thumb.jpg***, and ***lavender_thumb.jpg***) in their appropriate directories. This time, you'll need to add your own descriptions to the home page and create the HTML documents for the full-size images from scratch.

For an added challenge, create a new directory called ***photopages*** in the ***tuscany*** directory. Move ***countryside.html*** and ***sienna.html*** into that directory, and then update the URLs on those pages so that the images are visible again.

A Window in a Window

As long as we're talking about embedding things on a page, I thought I'd tell you about the `iframe` element that lets you to embed a separate HTML document or other resource in a document. What you see on the page is a floating or inline "frame" that displays the document with its own set of scrollbars if the embedded document is too long to fit (Figure 7-6).

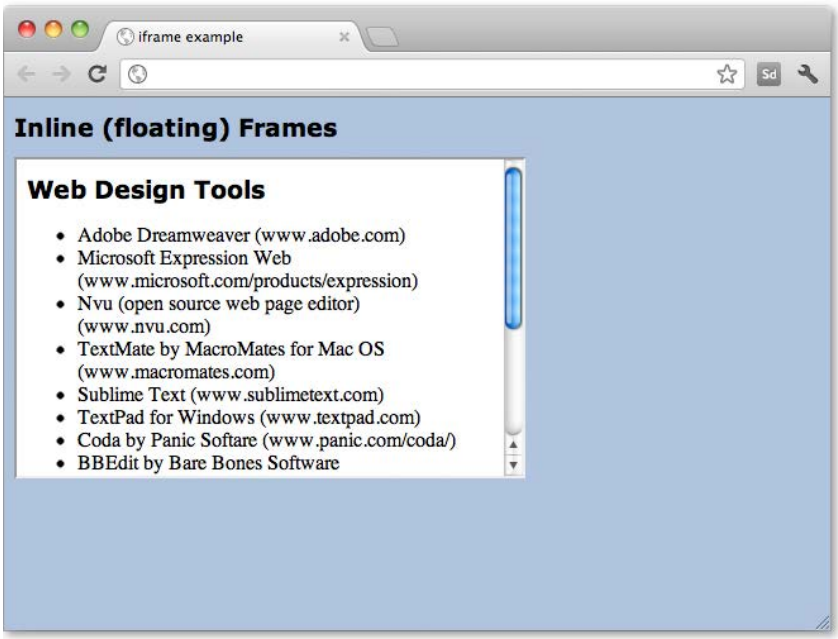


Figure 7-6. Inline frames (added with the `iframe` *element) are like a browser window within the browser that displays external HTML documents and resources.*

You place an inline frame on a page similarly to an image, specifying the source (`src`) of its content as well as its width and height. The content in the `iframe` element itself displays on browsers that don't support the element. This example displays a document called *list.html* in an inline frame.

```
<h1>Inline (floating) Frames</h1>
<iframe src="list.html" width="400" height="250">
Your browser does not support inline frames.Read the <a href="list.
    html">list</a>.
</iframe>
```

You don't see inline frames much in the wild, but developers sometimes use them to keep third party content such as interactive ads or other widgets quarantined so they don't interfere with the scripting and contents of the rest of the page.

Test Yourself

Images are a big part of the web experience. Answer these questions to see how well you've absorbed the key concepts of this chapter. The correct answers can be found in Appendix A.

1. Which attributes must be included in every **`img`** element?

2. Write the markup for adding an image called *furry.jpg* that is in the same directory as the current document.

3. Why is it necessary to include alternative text? Name two reasons.

4. What is the advantage of including **`width`** and **`height`** attributes for every graphic on the page? When should you leave them out?

5. What might be going wrong if your images don't appear when you view the page in a browser? There are three possible explanations.

Element Review: Images

We covered just one element in this chapter:

Element and attributes	Description
`img`	Inserts an inline image.
`src="url"`	The location of the image file.
`alt="text"`	Alternative text.
`width="number"`	Width of the graphic.
`height="number"`	Height of the graphic.
`usemap="usemap"`	Indicates a client-side image map.
`title="text"`	Provides a "tool tip" when the user mouses over the image. Can be used for supplemental information about the image.
`iframe`	Inserts an inline browsing context (window)
`height="number"`	Height of the frame in pixels
`src="url"`	Resource of the display in the frame
`width="number"`	Width of the frame in pixels

CHAPTER 8

TABLE MARKUP

Before we launch into the markup for tables, let's check in with our progress so far. We've covered a lot of territory: how to establish the basic structure of an HTML document, how to mark up text to give it meaning and structure, how to make links, and how to embed images on the page.

This chapter and the next, Chapter 9, Forms, describe the markup for specialized content that you might not have a need for right away. If you're getting antsy to make your pages look good, skip right to Part III and start playing with Cascading Style Sheets. The tables and forms chapters will be here when you're ready for them.

Are you still with me? Great. Let's talk tables. We'll start out by reviewing how tables should be used, then learn the elements used to create them with markup. Remember, this is an HTML chapter, so we're going to focus on the markup that structures the content into tables, and we won't be concerned with how the tables look. Like any web content, the appearance (or presentation, as we say in the web dev biz) of tables should be handled with style sheets, which you'll learn about in Chapter 18, CSS Techniques.

IN THIS CHAPTER

How Tables Are Used

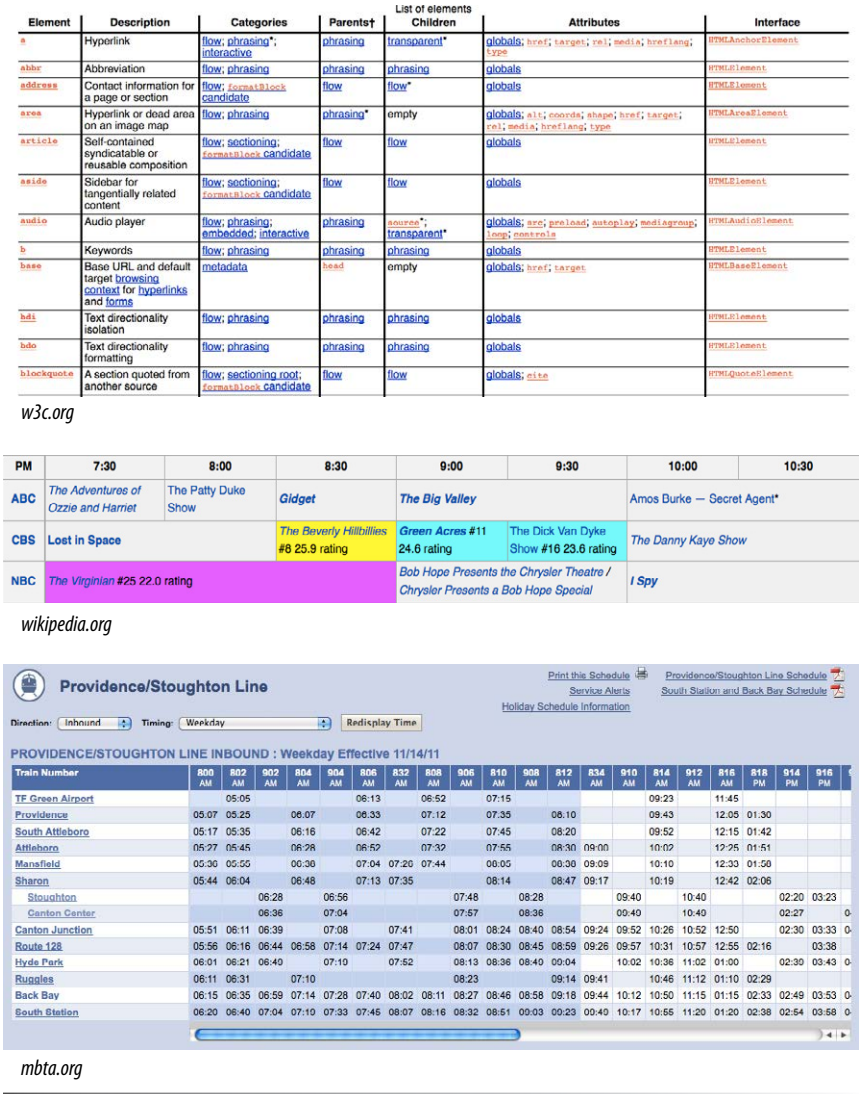
HTML tables were created for instances when you need to add tabular material (data arranged into rows and columns) to a web page. Tables may be used to organize calendars, schedules, statistics, or other types of information, as shown in Figure 8-1. Note that "data" doesn't necessarily mean numbers. A table cell may contain any sort of information, including numbers, text elements, and even images and multimedia objects.

The Trouble with Tables

Large tables, such as those shown in Figure 8-1, can be difficult to use on small-screen devices. By default, they are shrunk to fit the screen width, rendering the text in the cells too small to be read. Users can zoom in to read the cells, but then only a few cells may be visible at a time and it is difficult to parse the organization of headings and columns.

To be honest, as of this writing, we are just starting to figure out how best to handle tabular material on small screens. One approach is to replace the table with a graphic representation, such as a pie chart, on mobile devices. Of course, this will work only for certain types of tables. For simple two- or three-column tables, consider using a `dl` list to represent the information instead for more flexibility. Another approach is to put an indication of the table (such as an image of the top of it) that links to a separate screen with the full table for those who are interested. Chris Coyier proposes a clever solution in his article "Responsive Data Tables" (*css-tricks.com/9096-responsive-data-tables/*) that describes how to use CSS to reformat the table as a long, narrow list that fits better in a smartphone screen. See also the clever solution proposed by Filament Group (think of them as the Super Friends of responsive design) at *filamentgroup.com/lab/responsive_design_approach_for_complex_multicolumn_data_tables/*.

There may be new solutions by the time you read this, but it is important to always keep the mobile, small-screen experience in mind as you design any web content.



Element	Description	Categories	Parents†	List of elements Children	Attributes	Interface
a	Hyperlink	flow; phrasing*; interactive	phrasing	transparent*	globals; href; target; rel; media; hreflang; type	HTMLAnchorElement
abbr	Abbreviation	flow; phrasing	phrasing	phrasing	globals	HTMLElement
address	Contact information for a page or section	flow; formatBlock candidate	flow	flow*	globals	HTMLElement
area	Hyperlink or dead area on an image map	flow; phrasing	phrasing*	empty	globals; alt; coords; shape; href; target; rel; media; hreflang; type	HTMLAreaElement
article	Self-contained syndicatable or reusable composition	flow; sectioning; formatBlock candidate	flow	flow	globals	HTMLElement
aside	Sidebar for tangentially related content	flow; sectioning; formatBlock candidate	flow	flow	globals	HTMLElement
audio	Audio player	flow; phrasing; embedded; interactive	phrasing	source*; transparent*	globals; src; preload; autoplay; mediagroup; loop; controls	HTMLAudioElement
b	Keywords	flow; phrasing	phrasing	phrasing	globals	HTMLElement
base	Base URL and default target browsing context for hyperlinks and forms	metadata	head	empty	globals; href; target	HTMLBaseElement
bdi	Text directionality isolation	flow; phrasing	phrasing	phrasing	globals	HTMLElement
bdo	Text directionality formatting	flow; phrasing	phrasing	phrasing	globals	HTMLElement
blockquote	A section quoted from another source	flow; sectioning root; formatBlock candidate	flow	flow	globals; cite	HTMLQuoteElement

w3c.org

PM	7:30	8:00	8:30	9:00	9:30	10:00	10:30
ABC	*The Adventures of Ozzie and Harriet*	The Patty Duke Show	*Gidget*	*The Big Valley*		Amos Burke — Secret Agent*	
CBS	*Lost in Space*		*The Beverly Hillbillies* #8 25.9 rating	*Green Acres* #11 24.6 rating	The Dick Van Dyke Show #16 23.6 rating	*The Danny Kaye Show*	
NBC	*The Virginian* #25 22.0 rating			*Bob Hope Presents the Chrysler Theatre* / *Chrysler Presents a Bob Hope Special*		*I Spy*	

wikipedia.org

mbta.org

Figure 8-1. *Examples of tables used for tabular information, such as charts, calendars, and schedules.*

In visual browsers, the arrangement of data in rows and columns gives readers an instant understanding of the relationships between data cells and their respective header labels. Bear in mind when you are creating tables, however, that some readers will be hearing your data read aloud with a screen reader or reading braille output. Later in this chapter, we'll discuss measures you can take to make table content accessible to users who don't have the benefit of visual presentation.

In the days before style sheets, tables were the only option for creating multicolumn layouts or controlling alignment and whitespace. Layout tables, particularly the complex nested table arrangements that were once standard web design fare, have gone the way of the dodo. This chapter focuses on HTML tables as they are intended to be used.

Minimal Table Structure

Let's take a look at a simple table to see what it's made of. Here is a small table with three rows and three columns that lists nutritional information.

Menu item	Calories	Fat (g)
Chicken noodle soup	120	2
Caesar salad	400	26

`<table>...</table>`
Tabular content (rows and columns)

`<tr>...</tr>`
Table row

`<th>...</th>`
Table header

`<td>...</td>`
Table cell data

Figure 8-2 reveals the structure of this table according to the HTML table model. All of the table's content goes into cells that are arranged into rows. Cells contain either header information (titles for the columns, such as "Calories") or data, which may be any sort of content.

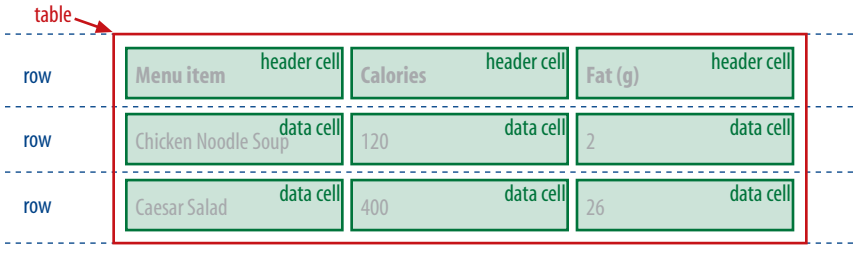


Figure 8-2. *Tables are made up of rows that contain cells. Cells are the containers for content.*

Simple enough, right? Now let's look at how those parts translate into elements (Figure 8-3).

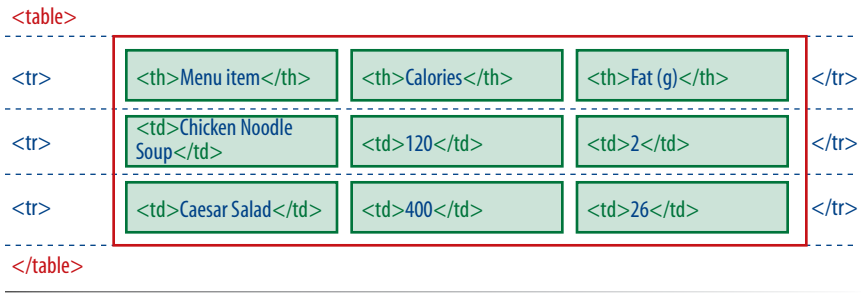


Figure 8-3. *The elements that make up the basic structure of a table.*

Figure 8-3 shows the elements that identify the table (**`table`**), rows (**`tr`**, for "table row"), and cells (**`th`**, for "table headers," and **`td`**, for "table data"). Cells are the heart of the table, because that's where the actual content goes. The other elements just hold things together.

What we don't see are column elements (see note). The number of columns in a table is determined by the number of cells in each row. This is one of the things that make HTML tables potentially tricky. Rows are easy—if you want the table to have three rows, just use three **tr** elements. Columns are different. For a table with four columns, you need to make sure that every row has four **td** or **th** elements; the columns are implied.

NOTE

There are two column-related elements in HTML5: `col` *for identifying a column and* `colgroup` *for establishing related groups of columns. They were created to add a layer of information about the table that can potentially speed up its display, but they are not part of HTML's row-centric table model. See the sidebar Advanced Table Elements for more information.*

Written out in a source document, the markup for the table in Figure 8-3 would look like the following sample. It is common to stack the **th** and **td** elements in order to make them easier to find in the source. This does not affect how they are rendered by the browser.

```
<table>
   <tr>
      <th>Menu item</th>
      <th>Calories</th>
      <th>Fat (g)</th>
   </tr>
   <tr>
      <td>Chicken noodle soup</td>
      <td>120</td>
      <td>2</td>
   </tr>
   <tr>
      <td>Caesar salad</td>
      <td>400</td>
      <td>26</td>
   </tr>
</table>
```

Remember, all the content must go in cells, that is, within **td** or **th** elements. You can put any content in a cell: text, a graphic, even another table.

Start and end **table** tags are used to identify the beginning and end of the tabular material. The **table** element may directly contain only some number of **tr** (row) elements. The only thing that can go in the **tr** element is some number of **td** or **th** elements. In other words, there may be no text content within the **table** and **tr** elements that isn't contained within a **td** or **th**.

Finally, Figure 8-4 shows how the table would look in a simple web page, as displayed by default in a browser. I know it's not exciting. Excitement happens in the CSS chapters. What is worth noting is that tables always start on new lines by default in browsers.

Advanced Table Elements

The sample table in this section has been stripped down to its bare essentials to make its structure clear while you learn how tables work. It is worth noting, however, that there are other table elements and attributes that offer more complex semantic descriptions and improve the accessibility of tabular content. A thoroughly marked-up version of the sample table might look like this:

```
<table>
<caption>Nutritional Information (Calorie and Fat
    Content)</caption>

<col span="1" class="itemname">
<colgroup id="data">
  <col span="1" class="calories">
  <col span="1" class="fat">
</colgroup>

<thead>
  <tr>
    <th scope="col">Menu item</th>
    <th scope="col">Calories</th>
    <th scope="column">Fat (g)</th>
  </tr>
</thead>

<tbody>
  <tr>
    <td>Chicken noodle soup</td>
    <td>120</td>
    <td>2</td>
  </tr>
  <tr>
    <td>Caesar salad</td>
    <td>400</td>
    <td>26</td>
  </tr>
</tbody>

</table>
```

Row group elements

You can describe rows or groups of rows as belonging to a header, footer, or the body of a table using the **`thead`**, **`tfoot`**, and **`tbody`** elements, respectively. Some user agents (another word for a browsing device) may repeat the header and footer rows on tables that span multiple pages. Authors may also use these elements to apply styles to various regions of a table.

Column group elements

Columns may be identified with the **`col`** element or put into groups using the **`colgroup`** element. This is useful for adding semantic context to information in columns and may be used to calculate the width of tables more quickly. Notice that there is no content in the column elements; it just describes the columns before the actual table data begins.

Accessibility features

Accessibility features such as captions for providing descriptions of table content and the **`scope`** and **`headers`** attributes for explicitly connecting headers with their respective content are discussed later in this chapter.

An in-depth exploration of the advanced table elements are beyond the scope of this book, but you may want to do more research at the W3C site (*www.w3.org/TR/html5*) if you anticipate working with data-heavy tables.

NOTE

According to the HTML5 spec, a table may contain "in this order: optionally a `caption` *element, followed by zero or more* `colgroup` *elements, followed optionally by a* `thead` *element, followed optionally by a* `tfoot` *element, followed by either zero or more* `tbody` *elements or one or more* `tr` *elements, followed optionally by a* `tfoot` *element (but there can only be one* `tfoot` *element child in total)." Got all that?*

Stylin' Tables

Once you build the structure of the table in the markup, it's no problem adding a layer of style to customize its appearance.

Style sheets can and should be used to control these aspects of a table's visual presentation. We'll get to all the formatting tools you'll need in the following chapters:

In Chapter 12, Formatting Text:

- Font settings for cell contents
- Text color in cells

In Chapter 14, Thinking Inside the Box:

- Table dimensions (width and height)
- Borders
- Cell padding (space around cell contents)
- Margins around the table

In Chapter 13, Colors and Backgrounds:

- background colors
- Tiling background images

In Chapter 18, CSS Techniques:

- Special properties for controlling borders and spacing between cells

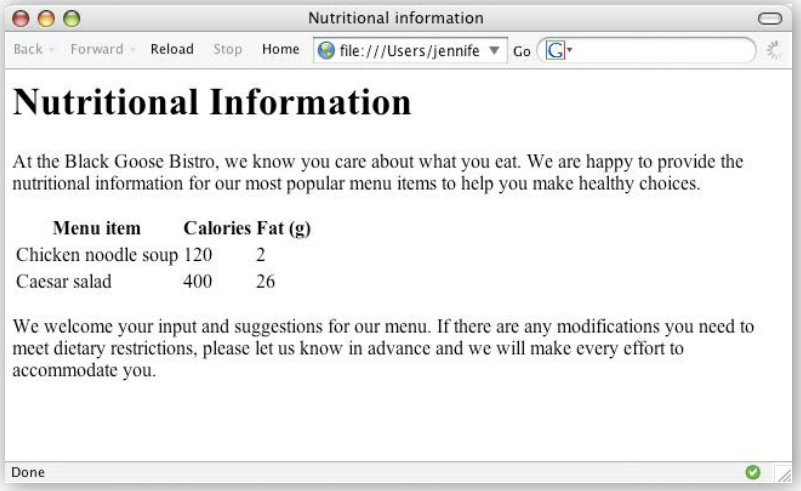


Figure 8-4. The default rendering of our sample table in a browser.

Here is the source for another table. Can you tell how many rows and columns it will have when it is displayed in a browser?

```
<table>
   <tr>
      <td>Sufjan Stevens</td>
      <td>Illinoise</td>
      <td>Asthmatic Kitty Records</td>
   </tr>
   <tr>
      <td>The Shins</td>
      <td>Oh Inverted World</td>
      <td>Sub-pop Records</td>
   </tr>
</table>
```

If you guessed that it's a table with two rows and three columns, you're correct! Two **tr** elements create two rows; three **td** elements in each row create three columns.

Table Headers

As you can see in Figure 8-4, the text marked up as headers (**th** elements) is displayed differently from the other cells in the table (**td** elements). The difference, however, is not purely cosmetic. Table headers are important because they provide information or context about the cells in the row or column they precede. The **th** element may be handled differently than **td**s by alternative browsing devices. For example, screen readers may read the header aloud before each data cell ("Menu item, Caesar salad, Calories, 400, Fat-g, 26").

In this way, they are a key tool for making table content accessible. Don't try to fake headers by formatting a row of **td** elements differently than the rest of the table. Conversely, don't avoid using **th** elements because of their default rendering (bold and centered). Mark up the headers semantically and change the presentation later with a style rule.

That covers the basics. Before we get fancier, try your hand at Exercise 8-1.

exercise 8-1 | Making a simple table

Try writing the markup for the table shown in Figure 8-5. You can open a text editor or just write it down on paper. The finished markup is provided in Appendix A.

(Note that I've added a 1-pixel border around cells with a style rule just to make the structure clear. You won't include this in your version.)

Be sure to close all table elements. Technically, you are not *required* to close **tr**, **th**, and **td** elements, but I want you to get in the habit of writing tidy source code for maximum predictability across all browsing devices. If you choose to write documents using XHTML syntax, closing table elements is required in order for the document to be valid.

Album	Year
Rubber Soul	1968
Revolver	1966
Sgt. Pepper's	1967
The White Album	1968
Abbey Road	1969

Figure 8-5. Write the markup for this table.

Spanning Cells

One fundamental feature of table structure is cell spanning, which is the stretching of a cell to cover several rows or columns. Spanning cells allows you to create complex table structures, but it has the side effect of making the markup a little more difficult to keep track of. You make a header or data cell span by adding the **colspan** or **rowspan** attributes, as we'll discuss next.

Column spans

Column spans, created with the **colspan** attribute in the **td** or **th** element, stretch a cell to the right to span over the subsequent columns (Figure 8-6). Here a column span is used to make a header apply to two columns. (I've added a border around cells to reveal the table structure in the screenshot.)

WARNING

Be careful with **colspan** *values. If you specify a number that exceeds the number of columns in the table, most browsers will add columns to the existing table, which typically screws things up.*

```
<table>
   <tr>
      <th colspan="2">Fat</th>
   </tr>
   <tr>
      <td>Saturated Fat (g)</td>
      <td>Unsaturated Fat (g)</td>
   </tr>
</table>
```

Fat	
Saturated Fat (g)	Unsaturated Fat (g)

Figure 8-6. The **colspan** *attribute stretches a cell to the right to span the specified number of columns.*

Notice in the first row (**tr**) that there is only one **th** element, while the second row has two **td** elements. The **th** for the column that was spanned over is no longer in the source; the cell with the **colspan** stands in for it. Every row should have the same number of cells or equivalent **colspan** values. For example, there are two **td** elements and the **colspan** value is 2, so the implied number of columns in each row is equal.

exercise 8-2 | Column spans

Try writing the markup for the table shown in Figure 8-7. You can open a text editor or just write it down on paper. I added borders to reveal the cell structure in the figure, but your table won't have them. Check Appendix A for the final markup.

Some hints:

- For simplicity's sake, this table uses all td elements.
- The second row shows you that the table has a total of three columns.
- When a cell is spanned over, its td element does not appear in the table.

7:00pm	7:30pm	8:00pm
The Sunday Night Movie		
Perry Mason	Candid Camera	What's My Line
Bonanza	The Wackiest Ship in the Army	

Figure 8-7. Practice column spans by writing the mvwarkup for this table.

Row spans

Row spans, created with the **rowspan** attribute, work just like column spans, but they cause the cell to span downward over several rows. In this example, the first cell in the table spans down three rows (Figure 8-8).

```
<table>
   <tr>
      <th rowspan="3">Serving Size</th>
      <td>Small (8oz.)</td>
   </tr>
   <tr>
      <td>Medium (16oz.)</td>
   </tr>
   <tr>
      <td>Large (24oz.)</td>
   </tr>
</table>
```

Again, notice that the **td** elements for the cells that were spanned over (the first cells in the remaining rows) do not appear in the source. The **rowspan="3"** implies cells for the subsequent two rows, so no **td** elements are needed.

Serving Size	Small (8oz.)
	Medium (16oz.)
	Large (24oz.)

Figure 8-8. The `rowspan` *attribute stretches a cell downward to span the specified number of rows.*

exercise 8-3 | Row spans

Try writing the markup for the table shown in Figure 8-9. Remember that cells that are spanned over do not appear in the table code. Rows always span downward, so the "oranges" cell is part of the first row even though its content is vertically centered.

If you're working in text editor, don't worry if your table doesn't look exactly like the one shown here. The resulting markup is provided in Appendix A.

Some hints:

- Rows always span downward, so the "oranges" cell is part of the first row
- Cells that are spanned over do not appear in the code

apples	oranges	pears
bananas		pineapple
lychees		

Figure 8-9. Practice row spans by writing the markup for this table.

Space In and Between Cells

By default, cells are sized just large enough to fit their contents, but often you'll want to add a little breathing room around tabular content (Figure 8-10). Because spacing is a matter of presentation, it is a job for style sheets.

Cell padding is the space inside the cell, between the content and the edge of the cell. To add cell padding, apply the CSS **`padding`** property to the **`td`** or **`th`** element.

Cell spacing, the area between cells, is a little more complicated. First, set the **`border-collapse`** property for the **`table`** to **`separate`**, then use the **`border-spacing`** property to specify the amount of space between borders. Unfortunately, this technique won't work in Internet Explorer 6, but hopefully IE6 usage will be inconsequential by the time you're reading this.

In the past, cell padding and spacing were handled by the **`cellpadding`** and **`cellspacing`** attributes in the **`table`** element, respectively, but they have been made obsolete in HTML5 due to their presentational nature.

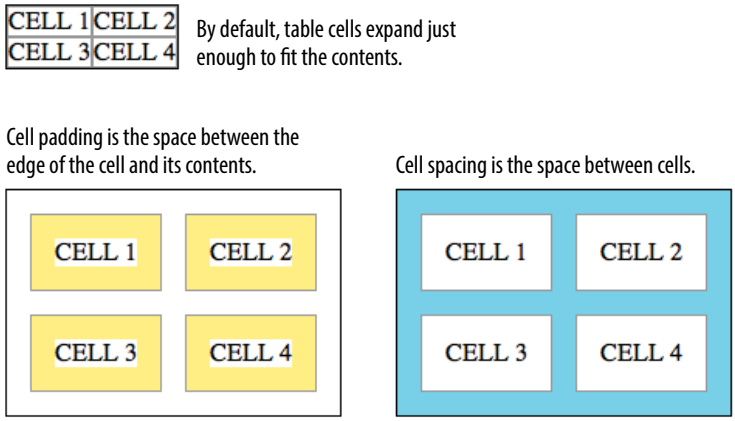


Figure 8-10. Cell padding and cell spacing.

Table Accessibility

As a web designer, it is important that you always keep in mind how your site's content is going to be used by non-sighted visitors. It is especially challenging to make sense of tabular material using a screen reader, but there are measures you can take to improve the experience and make your content more understandable.

Describing table content

The first step is to simply provide a description of your table's contents and perhaps the way it is structured if it is out of the ordinary.

Use the **caption** element to give a table a title or brief description that displays next to the table. You can use it to describe the table's contents or provide hints on how it is structured. When used, the **caption** element must be the first thing within the **table** element, as shown in this example that adds a caption to the nutritional chart from earlier in the chapter.

```
<table>
   <caption>Nutritional Information (Calorie and Fat Content)</caption>
   <tr>
      <th>Menu item</th>
      <th>Calories</th>
      <th>Fat (g)</th>
   </tr>

   …table continues…
</table>
```

The caption is displayed above the table by default, as shown in Figure 8-11, although you can use a style sheet property (**caption-side**) to move it below the table.

Nutritional Information

Menu item	Calories	Fat (g)
Chicken noodle soup	120	2
Caesar salad	400	26

Figure 8-11. The table caption is displayed above the table by default.

For longer descriptions, you could consider putting the table in a **figure** element and using the **figcaption** element for the description. The HTML5 specification has a number of suggestions for providing table descriptions, which you can find at *www.w3.org/TR/html5/tabular-data.html#table-descriptions-techniques*.

Connecting cells and headers

We discussed headers briefly as a straightforward method for improving the accessibility of table content, but sometimes it may be difficult to know which header applies to which cells. For example, headers may be at the left or right edge of a row rather than at the top of a column. And although it may be easy for sighted users to understand a table structure at a glance, for users hearing the data as text, the overall organization is not as clear. HTML 4.01 introduced a few attributes that allow authors to explicitly associate headers and their respective content.

NOTE

HTML 4.01 included a **summary** *attribute for the* **table** *element that was used for providing long descriptions to assistive devices while hiding them from visual browsers. However, it was omitted from HTML5 and will trigger validation errors.*

scope

The **scope** attribute associates a table header with the row, column, group of rows (such as **tbody**), or column group in which it appears using the values **row**, **column**, **rowgroup**, or **colgroup**, respectively. This example uses the **scope** attribute to declare that a header cell applies to the current row.

```
<tr>
   <th scope="row">Mars</th>
   <td>.95</td>
   <td>.62</td>
   <td>0</td>
</tr>
```

headers

For really complicated tables in which **scope** is not sufficient to associate a table data cell with its respective header (such as when the table contains multiple spanned cells), the **headers** attribute is used in the **td** element to explicitly tie it to a header's **id** value. In this example, the cell content ".38" is tied to the header "Diameter measured in earths":

```
<th id="diameter">Diameter measured in earths</th>

…many other cells…
<td headers="diameter">.38</td>
…many other cells…
```

This section is obviously only the tip of the iceberg of table accessibility. In-depth instruction on authoring accessible tables is beyond the scope of this beginner book. If you'd like to learn more, I recommend "Creating Accessible Tables" at WebAIM (*www.webaim.org/techniques/tables*) as an excellent starting point.

Wrapping Up Tables

This chapter gave you a good overview of the components of HTML tables. Exercise 8-4 puts most of what we covered together to give you a little more practice at authoring tables.

After just a few exercises, you're probably getting the sense that writing table markup manually, although not impossible, gets tedious and complicated quickly. Fortunately, web-authoring tools such as Dreamweaver provide interfaces that make the process much easier and time-efficient. Still, you'll be glad that you have a solid understanding of table structure and terminology, as well as the preferred methods for changing a table's appearance.

exercise 8-4 | **The table challenge**

Now it's time to put together the table writing skills you've acquired in this chapter. Your challenge is to write out the source document for the table shown in Figure 8-12.

I'll walk you through it one step at a time.

1. First, open a new document in your text editor and set up its overall structure (**html**, **head**, **title**, and **body** elements). Save the document as ***table.html*** in the directory of your choice.
2. Next, in order to make the boundaries of the cells and table clearer when you check your work, I'm going to have you add some simple style sheet rules to the document. Don't worry about understanding exactly what's happening here (although it's fairly intuitive); just insert this **style** element in the **head** of the document exactly as you see it here.

```
<head>
  <title>Table Challenge</title>
  <style type="text/css">
    td, th { border: 1px solid #CCC; }
    table {border: 1px solid black; }
  </style>
</head>
```

3. Now it's time to start building the table. I usually start by setting up the table and adding as many empty row elements as I'll need for the final table as placeholders, as shown here (it should be clear that there are five rows in this table).

```
<body>
<table>
  <tr></tr>
  <tr></tr>
  <tr></tr>
  <tr></tr>
  <tr></tr>
</table>
</body>
```

4. Start with the top row, and fill in the **th** and **td** elements from left to right, including any row or column spans as necessary. I'll help with the first row.

The first cell (the one in the top left corner) spans down the height of two rows, so it gets a **rowspan** attribute. I'll use a **th** here to keep it consistent with the rest of the row. This cell has no content.

```
<table>
  <tr>
    <th rowspan="2"></th>
  </tr>
```

The cell in the second column of the first row spans over the width of two columns, so it gets a **colspan** attribute:

```
<table>
  <tr>
    <th rowspan="2"></th>
    <th colspan="2">A common header for two
    subheads</th>
  </tr>
```

The cell in the third column has been spanned over by the **colspan** we just added, so we don't need to include it in the markup. The cell in the fourth column also spans down two rows.

```
<table>
  <tr>
    <th rowspan="2"></th>
    <th colspan="2">A common header for two
    subheads</th>
    <th rowspan="2">Header 3</th>
  </tr>
```

5. Now it's your turn. Continue filling in the **th** and **td** elements for the remaining four rows of the table. Here's a hint: the first and last cells in the second row have been spanned over. Also, if it's bold in the example, make it a header.
6. To complete the content, add the title over the table using the **caption** element.
7. Finally, use the **scope** attribute to make sure that the Thing A, Thing B, and Thing C headers are associated with their respective rows.
8. Save your work and open the file in a browser. The table should look just like the one on this page. If not, go back and adjust your markup. If you're stumped, the final markup for this exercise is listed in Appendix A.

Your Content Here

	A common header for two subheads		Header 3
	Header 1	Header 2	
Thing A	data A1	data A2	data A3
Thing B	data B1	data B2	data B3
Thing C	data C1	data C2	data C3

Figure 8-12. *The table challenge.*

Test Yourself

The answers to these questions are in Appendix A.

1. What are the parts (elements) of a basic HTML table?

2. Why don't professional web designers use tables for layout anymore?

3. When would you use the **col** (column) element?

4. Find five errors in this table markup.

```
<caption>Primetime Television
    1965</caption>
<table>
  Thursday Night
  <tr></tr>
    <th>7:30</th>
    <th>8:00</th>
    <th>8:30</th>
  <tr>
    <td>Shindig</td>
    <td>Donna Reed Show</td>
    <td>Bewitched</td>
  <tr>
    <colspan>Laredo</colspan>
    <td>Daniel Boone</td>
  </tr>
</table>
```

Element Review: Tables

The following is a summary of the elements we covered in this chapter:

Element and attributes	Description
`table`	Establishes a table element
`td`	Establishes a cell within a table row
`colspan="number"`	Number of columns the cell should span
`rowspan="number"`	Number of rows the cell should span
`headers="header name"`	Associates the data cell with a header
`th`	Table header associated with a row or column
`colspan="number"`	Number of columns the cell should span
`rowspan="number"`	Number of rows the cell should span
`headers="header name"`	Associates a header with another header
`scope="row|col|rowgroup|colgroup"`	Associates the header with a row, row group, column, or column group
`tr`	Establishes a row within a table
`caption`	Gives the table a title that displays in the browser
`col`	Declares a column
`colgroup`	Declares a group of columns
`tbody`	Identifies the table body row group
`tfoot`	Identifies the table footer grow group
`thead`	Identifies the table header row group

FORMS

It didn't take long for the web to shift from a network of pages to read to a place where you went to get things *done*—making purchases, booking plane tickets, signing petitions, searching a site, posting a tweet...the list goes on! All of these interactions are handled by forms.

In fact, in response to this shift from page to application, HTML5 introduced a bonanza of new form controls and attributes that make it easier for users to fill out forms and for developers to create them. Tasks that have traditionally relied on JavaScript may be handled by markup and native browser behavior alone. HTML5 introduces a number of new form-related elements, 13 new input types, and many new attributes (they are listed in Table 9-1 at the end of this chapter). Some of these features are waiting for browser implementation to catch up, so I will be sure to note which controls may not be universally supported.

This chapter introduces web forms, how they work, and the markup used to create them. I'll also briefly discuss the importance of web form design.

How Forms Work

There are two parts to a working form. The first part is the form that you see on the page itself that is created using HTML markup. Forms are made up of buttons, input fields, and drop-down menus (collectively known as form controls) used to collect information from the user. Forms may also contain text and other elements.

The other component of a web form is an application or script on the server that processes the information collected by the form and returns an appropriate response. It's what makes the form *work*. In other words, posting an HTML document with form elements isn't enough. Web applications and scripts require programming know-how that is beyond the scope of this book, but the Getting Your Forms to Work sidebar later in this chapter provides some options for getting the scripts you need.

A Word About Encoding

Form data is encoded using the same method used for URLs in which spaces and other characters that are not permitted are translated into their hexadecimal equivalents. For example, each space character in the collected form data is represented by the character string **%20**, and a slash (/) character is replaced with **%2F**. You don't need to worry about this; the browser handles it automatically.

From data entry to response

If you are going to be creating web forms, it is beneficial to understand what is happening behind the scenes. This example traces the steps of a transaction using a simple form that gathers names and email addresses for a mailing list; however, it is typical of the process for many forms.

1. Your visitor, let's call her Sally, opens the page with a web form in the browser window. The browser sees the form control elements in the markup and renders them with the appropriate form controls on the page, including two text entry fields and a submit button (shown in Figure 9-1).
2. Sally would like to sign up for this mailing list, so she enters her name and email address into the fields and submits the form by hitting the "Submit" button.
3. The browser collects the information she entered, encodes it (see the sidebar A Word About Encoding), and sends it to the web application on the server.

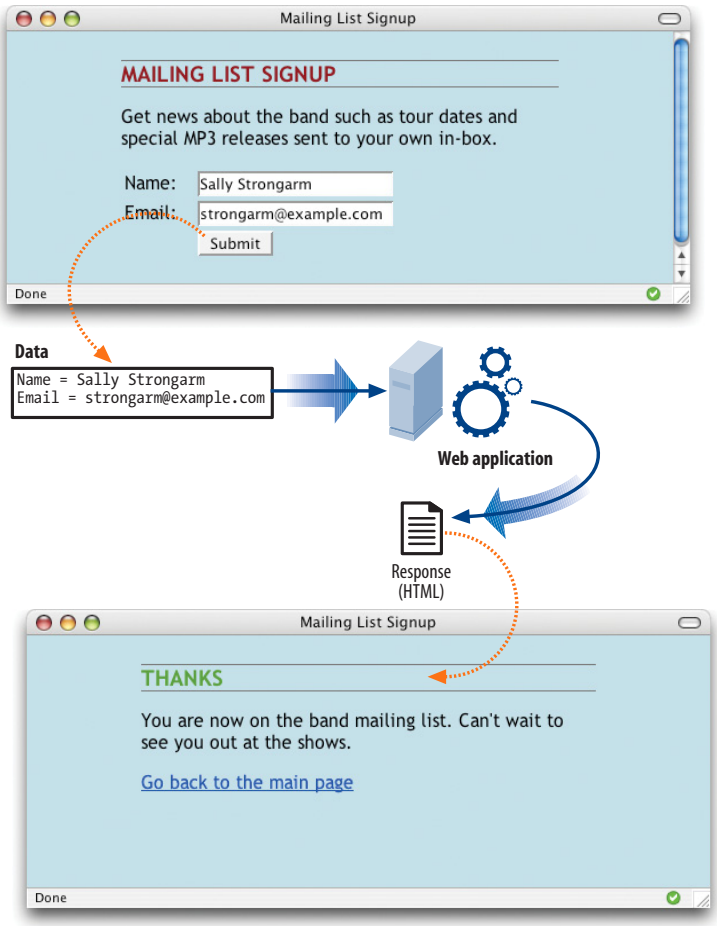


Figure 9-1. What happens behind the scenes when a web form is submitted

4. The web application accepts the information and processes it (that is, does whatever it is programmed to do with it). In this example, the name and email address are added to a database.
5. The web application also returns a response. The kind of response sent back depends on the content and purpose of the form. Here, the response is a simple web page that contains a thank you for signing up for the mailing list. Other applications might respond by reloading the HTML form page with updated information, by moving the user on to another related form page, or by issuing an error message if the form is not filled out correctly, to name only a few examples.
6. The server sends the web application's response back to the browser where it is displayed. Sally can see that the form worked and that she has been added to the mailing list.

The form Element

`<form>...</form>`

Interactive form

Forms are added to web pages using (no surprise here) the **`form`** element. The **`form`** element is a container for all the content of the form, including some number of form controls, such as text entry fields and buttons. It may also contain block elements (**`h1`**, **`p`**, and lists, for example). However, it may *not* contain another **`form`** element.

TIP

Be careful not to nest **`form`** elements or allow them to overlap. A **`form`** element must be closed before the next one begins.

This sample source document contains a form similar to the one shown in Figure 9-1:

```
<!DOCTYPE html>
<html>
<head>
  <title>Mailing List Signup</title>
  <meta charset="utf-8">
</head>
<body>
  <h1>Mailing List Signup</h1>

  <form action="/mailinglist.php" method="post">
    <fieldset>
    <legend>Join our email list</legend>
    <p>Get news about the band such as tour dates and special MP3
releases sent to your own in-box.</p>
  <ol>
    <li><label for="firstlast">Name:</label>
        <input type="text" name="username" id="firstlast"></li>
    <li><label for="email">Email:</label>
        <input type="text" name="email" id="email"></li>
    </ol>
    <input type="submit" value="Submit">
    </fieldset>
  </form>

</body>
</html>
```

NOTE

It is current best practice to wrap form controls in semantic HTML elements such as lists or **`div`***s. Ordered lists, as shown in this example, are a popular solution, but know that there are often default styles that need to be cleared out before styling them, particularly on mobile browsers.*

In addition to being a container for form control elements, the **`form`** element has some attributes that are necessary for interacting with the form-processing program on the server. Let's take a look at each.

The action attribute

The **`action`** attribute provides the location (URL) of the application or script (sometimes called the action page) that will be used to process the form. The **`action`** attribute in this example sends the data to a script called *mailinglist.php*.

```
<form action="/mailinglist.php" method="post">...</form>
```

Getting Your Forms to Work

If you aren't a programmer, don't fret. You have a few options for getting your forms operational.

Use hosting plan goodies

Many site hosting plans include access to scripts for simple functions such as mailing lists. More advanced plans may even provide everything you need to add a full shopping cart system to your site as part of your monthly hosting fee. Documentation or a technical support person should be available to help you use them.

Hire a programmer

If you need a custom solution, you may need to hire a programmer who has server-side programming skills. Tell your programmer what you are looking to accomplish with your form and he or she will suggest a solution. Again, you need to make sure you have permission to install scripts on your server under your current hosting plan, and that the server supports the language you choose.

The *.php* suffix indicates that this form is processed by a script written in the PHP scripting language, but web forms may be processed using one of the following technologies:

- PHP (*.php*) is an open source scripting language most commonly used with the Apache web server.
- Microsoft's ASP.NET (Active Server Pages) (*.asp*) is a programming environment for the Microsoft Internet Information Server (IIS).
- Ruby on Rails. Ruby is the programming language that is used with the Rails platform. Many popular web applications are built with it.
- JavaServer Pages (*.jsp*) is a Java-based technology similar to ASP.
- Python is a popular scripting language for web and server applications.

There are other forms processing options that may have their own suffixes or none at all (as is the case for the Ruby on Rails platform). Check with your programmer, server administrator, or script documentation for the proper name and location of the program to be provided by the **`action`** attribute.

Sometimes there is form-processing code such as PHP embedded right in the HTML file. In that case, leave the action empty and the form will post to the page itself.

The method attribute

The **`method`** attribute specifies how the information should be sent to the server. Let's use this data gathered from the sample form in Figure 9-1 as an example.

```
username = Sally Strongarm
email = strongarm@example.com
```

When the browser encodes that information for its trip to the server, it looks like this (see the earlier sidebar if you need a refresher on encoding):

```
username=Sally%20Strongarm&email=strongarm%40example.com
```

There are only two methods for sending this encoded data to the server: POST or GET, indicated using the **`method`** attribute in the **`form`** element. The method is optional and will default to GET if omitted. We'll look at the difference between the two methods in the following sections. Our example uses the POST method, as shown here:

```
<form action="/cgi-bin/mailinglist.pl" method="POST">...</form>
```

The POST method

When the form's method is set to POST, the browser sends a separate server request containing some special headers followed by the data. Only the server sees the content of this request, thus it is the best method for sending secure information such as credit card or other personal information.

The POST method is also preferable for sending a lot of data, such as a lengthy text entry, because there is no character limit as there is for GET.

The GET method

With the GET method, the encoded form data gets tacked right onto the URL sent to the server. A question mark character separates the URL from the following data, as shown here:

```
get http://www.bandname.com/cgi-bin/mailinglist.pl?name=Sally%20Strongarm&email=strongarm%40example.com
```

The GET method is appropriate if you want users to be able to bookmark the results of a form submission (such as a list of search results). Because the content of the form is in plain sight, GET is not appropriate for forms with private personal or financial information. In addition, GET may not be used when the form is used to upload a file.

In this chapter, we'll stick with the more prevalent POST method. Now that we've gotten through the technical aspects of the **`form`** element, we can take on the real meat of forms: form controls.

NOTE

POST and GET are not case-sensitive and are commonly listed in all uppercase by convention. In XHTML documents, however, the value of the `method` *attribute (post or get) must be provided in all lowercase letters.*

Variables and Content

Web forms use a variety of controls that allow users to enter information or choose options. Control types include various text entry fields, buttons, menus, and a few controls with special functions. They are added to the document using a collection of form control elements that we'll be examining one by one in the upcoming Great Form Control Roundup section.

As a web designer, it is important to be familiar with control options to make your forms easy and intuitive to use. It is also useful to have an idea of what form controls are doing behind the scenes.

The name attribute

The job of a form control is to collect one bit of information from a user. In the form example a few pages back, text entry fields collect the visitor's name and email address. To use the technical term, "username" and "email" are two variables collected by the form. The data entered by the user ("Sally Strongarm" and "strongarm@example.com") is the value or content of the variable.

The **`name`** attribute provides the variable name for the control. In this example, the text gathered by a **`textarea`** element is defined as the "comment" variable:

```
<textarea name="comment" rows="4" cols="45" placeholder="Leave us a comment."></textarea>
```

When a user enters a comment in the field ("This is the best band ever!"), it would be passed to the server as a name/value (variable/content) pair like this:

```
comment=This%20is%20the%20best%20band%20ever%21
```

All form control elements must include a **name** attribute so the form-processing application can sort the information. You may include a **name** attribute for **submit** and **reset** button elements, but they are not required, because they have special functions (submitting or resetting the form) not related to data collection.

Naming your variables

You can't just name controls willy-nilly. The web application that processes the data is programmed to look for specific variable names. If you are designing a form to work with a preexisting application or script, you need to find out the specific variable names to use in the form so they are speaking the same language. You can get the variable names from the developer you are working with, your system administrator, or from the instructions provided with a ready-to-use script on your server.

If the script or application will be created later, be sure to name your variables simply and descriptively and to document them well. In addition, each variable must be named uniquely, that is, the same name may not be used for two variables. You should also avoid putting character spaces in variable names; use an underscore or hyphen instead.

We've covered the basics of the **form** element and how variables are named. Now we can get to the real meat of form markup: the controls.

The Great Form Control Roundup

This is the fun part—playing with the markup that adds form controls to the page. This section introduces the elements used to create:

- Text entry controls
- Specialized text entry controls
- Submit and reset buttons
- Radio and checkbox buttons
- Pull-down and scrolling menus
- File selection and upload control
- Hidden controls
- Dates and times (HTML5)
- Numerical controls (HTML5)
- Color picker control (HTML5)

We'll pause along the way to allow you to try them out by constructing the questionnaire form shown in Figure 9-2.

As you will see, the majority of controls are added to a form using the `input` element. The functionality and appearance of the `input` element changes based on the value of the `type` attribute in the tag. In HTML5, there are *twenty-three* different types of input controls. We'll take a look at them all.

NOTE

The attributes associated with each input type are listed in Table 9-1 at the end of this chapter.

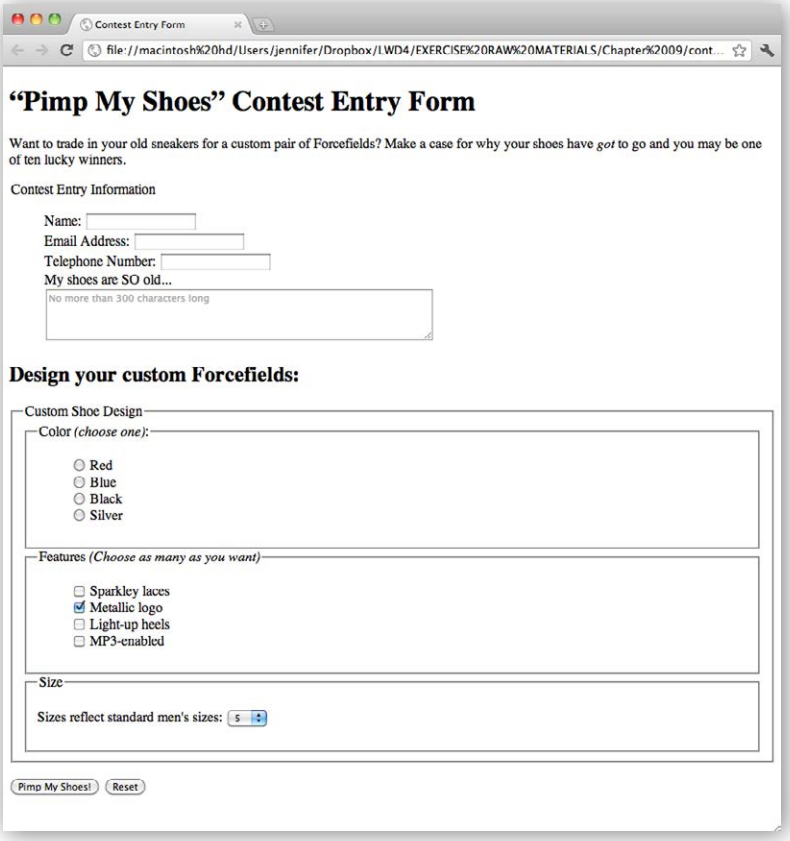


Figure 9-2. The contest entry form we'll be building in the exercises in this chapter.

Text entry controls

One of the most common tasks in a web form is to enter text information. Which element you use depends on whether users are asked to enter a single line of text (`input`) or multiple lines (`textarea`).

NOTE

The markup examples throughout this section include the `label` *element, which is used to improve accessibility. We will discuss* `label` *in more detail in the Form Accessibility Features section later in this chapter, but in the meantime, I want you to get used to seeing proper form markup.*

Single-line text field

`<input type="text">`
Single-line text entry control

One of the most straightforward form input types is the text entry field used for entering a single word or line of text. In fact, it is the default input type, which means it is what you'll get if you forget to include the **type** attribute or include an unrecognized value. Add a text input field to a form with the **input** element with its **type** attribute set to **text**, as shown here and in Figure 9-3.

```
<li><label>City <input type="text" name="city" id="form-city"
value="Your Hometown" maxlength="50"></label></li>
```

There are a few attributes in there I'd like to point out.

`name`

The **name** attribute is required for indicating the variable name.

`value`

The **value** attribute specifies default text that appears in the field when the form is loaded. When you reset a form, it returns to this value.

`maxlength`

By default, users can type an unlimited number of characters in a text field regardless of its size (the display scrolls to the right if the text exceeds the character width of the box). You can set a maximum character limit using the **maxlength** attribute if the forms processing program you are using requires it.

Multiline text entry field

`<textarea>...</textarea>`
Multiline text entry control

At times, you'll want your users to be able enter more than just one line of text. For these instances, use the **textarea** element that is replaced by a multiline, scrollable text entry box when displayed by the browser (Figure 9-3).

NOTE

The specific rendering style of form controls varies by operating system and browser version.

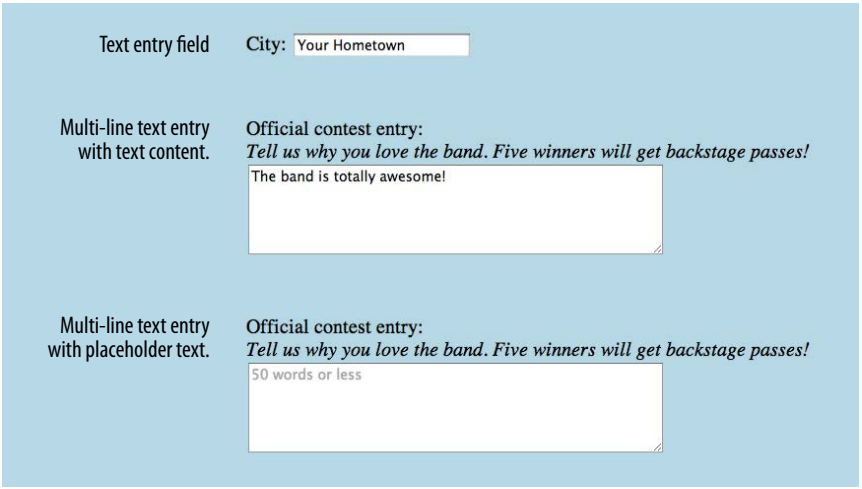


Figure 9-3. Examples of the text entry control options for web forms.

Unlike the empty **input** element, you can put content between the opening and closing tags in the **textarea** element. The content of the **textarea** element will show up in the text box when the form is displayed in the browser. It will also get sent to the server when the form is submitted, so carefully consider what goes there. It is not uncommon for developers to put nothing between the opening and closing tags, and provide a hint of what should go there with a **title** or **placeholder** attribute instead. The new HTML5 **placeholder** attribute can be used with **textarea** and other text-based **input** types and is used to provide a short hint of how to fill in the field. It is not supported on Android, older versions of Firefox (versions earlier than 3.6), or IE as of this writing.

```
<p><label>Official contest entry <br>
<em>Tell us why you love the band. Five winners will get backstage
passes!</em><br>
<textarea name="contest_entry" rows="5" cols="50">The band is totally
awesome!</textarea></label></p>

<p>Official contest entry:<br>
<em>Tell us why you love the band. Five winners will get backstage
passes!</em><br>
<textarea name="contest_entry" placeholder="50 words or less">
</textarea>
</p>
```

The **rows** and **cols** attributes are a way of specifying the size of the **textarea** using markup, but it is more commonly sized with CSS. **rows** specifies the number of lines the text area should display, and **cols** specifies the width in number of characters. Scrollbars will be provided if the user types more text than fits in the allotted space.

There are also a few attributes not shown in the example. The **wrap** attribute specifies whether the text should keep its line breaks when submitted. A value of **soft** (the default) does not preserve line breaks, and **hard** does. The **maxlength** attribute (new in HTML5) sets a limit on the number of characters that can by typed into the field.

Specialized text entry fields

In addition to the generic single-line text entry, there are a number of input types for entering specific types of information such as passwords, search terms, email addresses, telephone numbers, and URLS.

Password entry field

```
<input type="password">
```

Password text control

A password field works just like a text entry field, except the characters are obscured from view using asterisk (*) or bullet (•) characters, or another character determined by the browser.

disabled and readonly

The **disabled** and **readonly** attributes can be added to any form control element to prevent users from selecting them. When a form element is disabled, it cannot be selected. Visual browsers may render the control as grayed-out by default (which you can change with CSS, of course). The disabled state can only be changed with a script. This is a useful attribute for restricting access to some form fields based on data entered earlier in the form.

The **readonly** attribute prevents the user from changing the value of the form control (although it can be selected). This enables developers to use scripts to set values for controls contingent on other data entered earlier in the form. Inputs that are **readonly** should have strong visual cues that they are somehow different than other inputs, or they could be confusing to users who are trying to change their values.

WARNING

iOS ignores **disabled** *on* **option** *elements as of this writing (iOS 5 and earlier).*

It's important to note that although the characters entered in the password field are not visible to casual onlookers, the form does not encrypt the information, so it should not be considered a real security measure.

Here is an example of the markup for a password field. Figure 9-4 shows how it might look after the user enters a password in the field.

```
<li><label for="form-pswd">Log in:</label><br>
 <input type="password" name="pswd" maxlength="8" id="form-pswd"></li>
```

Figure 9-4. Passwords are converted to bullets in the browser display.

HTML5 text inputs

`<input type="search">`
Search field
NEW IN HTML5

`<input type="email">`
Email address
NEW IN HTML5

`<input type="tel">`
Telephone number
NEW IN HTML5

`<input type="url">`
Location (URL)
NEW IN HTML5

Until HTML5, the only way to collect email addresses, telephone numbers, URLs, or search terms was to insert a generic text input field. In HTML5, the **email**, **tel**, **url**, and **search** input types give the browser a heads-up as to what type of information to expect in the field. These new input types use the same attributes as the generic text input type described earlier (**name**, **maxlength**, **size**, and **value**), as well as a number of new HTML5 attributes.

All of these input types are typically displayed as single-line text inputs. But browsers that support them can do some interesting things with the extra semantic information. For example, Safari on iOS uses the input type to provide a keyboard well-suited to the entry task, such as the keyboard featuring a "Search" button for the **search** input type or a ".com" button when the input type is set to **url** (Figure 9-5). Browsers usually add a one-click "clear field" icon (usually a little X) in search fields. A supporting browser could check the user's input to see that it is valid, such as making sure text entered in an **email** input follows standard email address structure (in the past, you needed JavaScript for validation). For example, the Opera (Figure 9-6) and Chrome browsers display a warning if the input does not match the expected format.

Not all browsers support the new HTML5 input types or support them in the same way, but the good news is that if the type isn't recognized, the default generic text input is displayed instead, which works perfectly fine. There is no reason not to start using them right away as a progressive enhancement, even if you can't reap the benefits of easy user input and browser (client-side) validation.

Figure 9-5. Safari on iOS provides custom keyboards based on the input type.

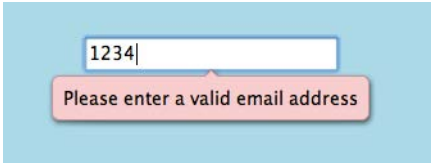


Figure 9-6. Opera displays a warning when input does not match the expected `email` *format as part of its client-side validation support.*

The datalist Element

The **`datalist`** element (new in HTML5) allows the author to provide a drop-down menu of suggested values for any type of text input. It gives the user some shortcuts to select from, but if none are selected, the user can still type in her own text. Within the **`datalist`** element, suggested values are marked up as option elements. Use the **`list`** attribute in the **`input`** element to associate it with the **`id`** of its respective **`datalist`**.

In the following example (Figure 9-7), a **`datalist`** suggests several education level options for a text input.

```
<p>Education completed: <input type="text"
    list="edulevel" name="education">

<datalist id="edulevel">
  <option value="High School">
  <option value="Bachelors Degree">
  <option value="Masters Degree">
  <option value="PhD">
</datalist>
```

As of this writing, only the Opera browser has implemented the **`datalist`** element. Other browsers will ignore it and present a simple text input, which is a perfectly acceptable fallback. You could also use JavaScript to create **`datalist`** functionality (i.e., a polyfill).

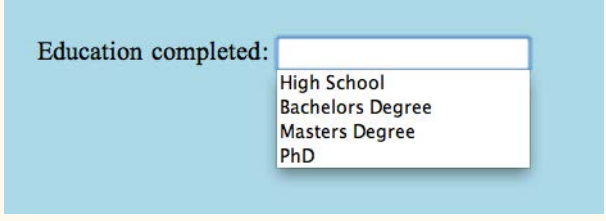


Figure 9-7. A `datalist` *creates a pop-up menu of suggested values for a text entry field.*

Submit and reset buttons

`<input type="submit">`

Submits the form data to the server

`<input type="reset">`

Resets the form controls to their default settings

There are several different kinds of buttons that can be added to web forms. The most fundamental is the submit button. When clicked or tapped, the submit button immediately sends the collected form data to the server for processing. A reset button returns the form controls to the state they were in when the form initially loaded. In other words, resetting the form doesn't simply clear all the fields.

Both submit and reset buttons are added using the **input** element. As mentioned earlier, because these buttons have specific functions that do not include the entry of data, they are the only form control elements that do not require the **name** attribute, although it is OK to add one if you need it.

Submit and reset buttons are straightforward to use. Just place them in the appropriate place in the form, which in most cases is at the very end. By default, the submit button displays with the label "Submit" or "Submit Query" and the reset button is labeled "Reset." Change the text on the button using the **value** attribute, as shown in the reset button in this example (Figure 9-8).

```
<p><input type="submit"> <input type="reset" value="Start over"></p>
```

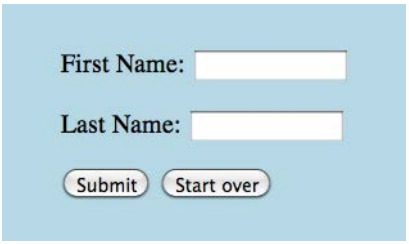


Figure 9-8. Submit and reset buttons.

The reset button is not used in forms as commonly as it used to be. That iss because in contemporary form development, we use JavaScript to check the validity of form inputs along the way, so the users get feedback as they go along. With thoughtful design and assistance, fewer users should get to the end of the form and need to reset the whole thing. Still, it is a good function to be aware of.

At this point, you know enough about form markup to start building the questionnaire shown in Figure 9-2. Exercise 9-1 walks you through the first steps.

A Few More Buttons

There are a handful of custom button elements that are a little off the beaten path for beginners, but in the interest of thoroughness, here they are tucked off in a sidebar.

Image buttons

`<input type="image">`

This type of **input** control allows you to replace the submit button with an image of your choice. The image will appear flat, not like a 3-D button. Unfortunately, this type of button has accessibility issues, so be sure to include a carefully chosen **alt** value.

Custom input button

`<input type="button">`

Setting the type of the **input** element to "button" creates a button that can be customized with JavaScript. It has no predefined function on its own, unlike submit and reset buttons.

The button element

`<button>...</button>`

The **button** element is a flexible element for creating custom buttons similar to those created with the **input** element. The content of the button element (text and/or images) is what gets displayed on the button.

For more information on what you can do with the **button** element, read "Push My Button" by Aaron Gustafson at *digital-web.com/articles/push_my_button.*

exercise 9-1 | Starting the contest form

Here's the scenario. You are the web designer in charge of creating the entry form for the Forcefield Sneakers "Pimp My Shoes!" Contest. The copy editor has handed you a sketch (Figure 9-9) of the form's content, complete with notes of how some controls should work. There are sticky notes from the programmer with information about the script and variable names you need to use.

Your challenge is to turn the sketch into a functional online form. I've given you a head start by creating a bare-bones document containing the text content and some minimal markup and styles. This document, ***contest_entry.html***, is available online at *www.learningwebdesign.com/4e/materials*. The source for the entire finished form is provided in Appendix A if you want to check your work.

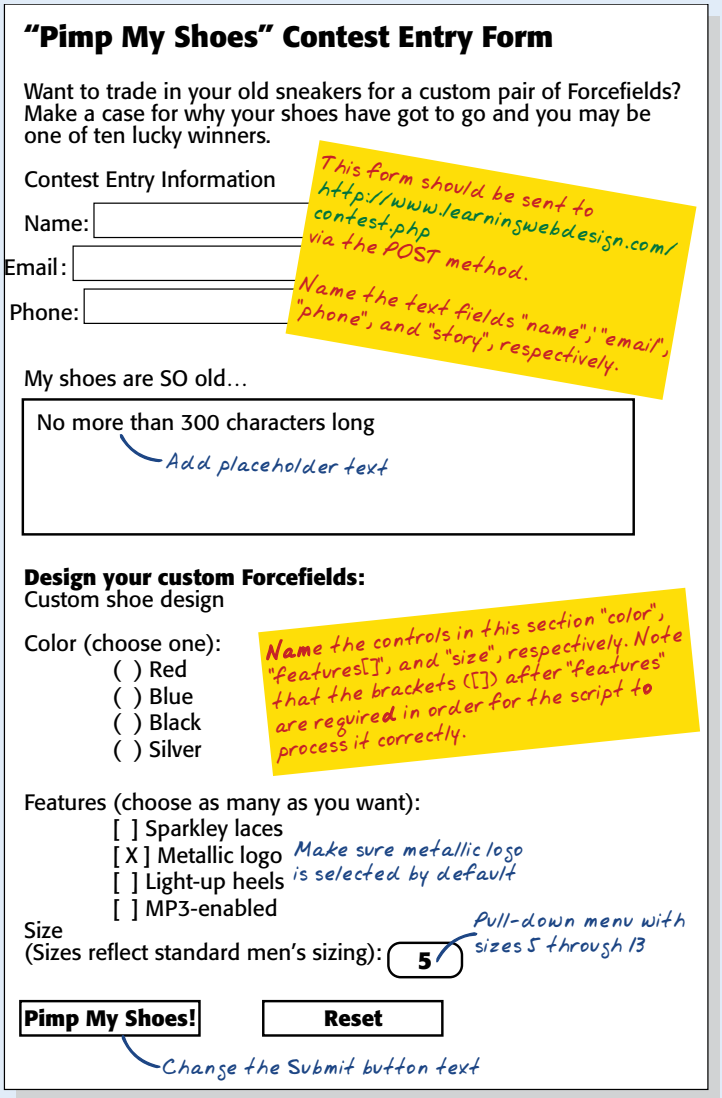


Figure 9-9. A sketch of the contest entry form.

1. Open *contest_entry.html* in a text editor.
2. The first thing we'll do is put everything after the intro paragraph into a **form** element. The programmer has left a note specifying the **action** and the **method** to use for this form. The resulting **form** element should look like this:

```
<form action="http://www.learningwebdesign.com/contest.php"
method="post">
...
</form>
```

3. In this exercise, we'll work on the "Contest Entry Information" section of the form. Start with the first three short text entry form controls that are marked up appropriately as an unordered list. Here's the first one; you insert the other two.

```
<li>Name: <input type="text" name="username"></li>
```

   Hints: Choose the most appropriate input type for each entry field. Be sure to name the input elements as specified in the programmer's note.

4. Now add a multiline text area for the shoe description on a new line. Because we aren't writing a style sheet for this form, use markup to make it four rows long and 60 characters wide (in the real world, CSS is preferable because it gives you more fine-tuned control).

```
<li>My shoes are SO old...<br>
<textarea name="story" rows="4" cols="60"
maxlength="300" placeholder="No more than 300
characters long"></textarea></li>
```

5. We'll skip the rest of the form for now until we get a few more controls under our belt, but we can add the submit and reset buttons at the end, just before the **</form>** tag. Note that we need to change the text on the submit button.

```
<p><input type="submit" value="Pimp my shoes!">
<input type="reset"></p>
</form>
```

6. Now, save the document and open it in a browser. The parts that are finished should generally match Figure 9-3. If it doesn't, then you have some more work to do.

Once it looks right, take it for a spin by entering some information and submitting the form. You should get a response like the one shown in Figure 9-10 (yes, *contact.php* actually works, but sorry, the contest is make-believe.)

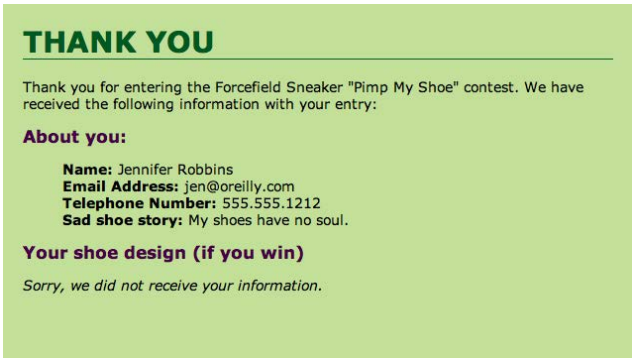


Figure 9-10. You should see a response page like this if your form is working.

Radio and checkbox buttons

Both checkbox and radio buttons make it simple for your visitors to choose from a number of provided options. They are similar in that they function like little on/off switches that can be toggled by the user and are added using the **input** element. They serve distinct functions, however.

A form control made up of a collection of radio buttons is appropriate when only one option from the group is permitted—in other words, when the selections are mutually exclusive (such as Yes or No, or Male or Female). When one radio button is "on," all of the others must be "off," sort of the way buttons used to work on old radios: press one button in and the rest pop out.

When checkboxes are grouped together, however, it is possible to select as many or as few from the group as desired. This makes them the right choice for lists in which more than one selection is okay.

NOTE

I have omitted the `fieldset` *and* `label` *elements from the code examples for radio buttons, checkboxes, and menus in order to keep the markup structure as simple and clear as possible. In the upcoming Form Accessibility Features section, you will learn why it is important to include them in your markup for all form elements.*

Radio buttons

`<input type="radio">`
Radio button

Radio buttons are added to a form using the **input** element with the **type** attribute set to **radio**. Here is the syntax for a minimal radio button:

```
<input type="radio" name="variable" value="value">
```

The **name** attribute is required and plays an important role in binding multiple radio inputs into set. When you give a number of radio button inputs the same **name** value (**age** in the following example), they create a group of mutually exclusive options.

In this example, radio buttons are used as an interface for users to enter their age group (a person can't belong to more than one age group, so radio buttons are the right choice). Figure 9-11 shows how radio buttons are rendered in the browser.

```
<p>How old are you?</p>
<ol>
  <li><input type="radio" name="age" value="under24" checked> under
24</li>
  <li><input type="radio" name="age" value="25-34"> 25 to 34</li>
  <li><input type="radio" name="age" value="35-44"> 35 to 44</li>
  <li><input type="radio" name="age" value="over45"> 45+</li>
</ol>
```

Notice that all of the **input** elements have the same variable name ("age"), but their values are different. Because these are radio buttons, only one button can be checked at a time, and therefore, only one value will be sent to the server for processing when the form is submitted.

NOTE

XHTML syntax, the value of the **checked** *attribute must be explicitly set to* **checked**, *as shown in the example.*

```
<input type="radio" name="foo"
checked="checked" />
```

But in HTML syntax, you don't need to write out the value for the **checked** *attribute. It can be minimized, as shown here:*

```
<input type="radio" name="foo"
checked >
```

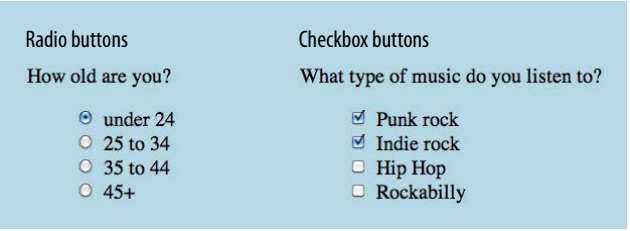


Figure 9-11. *Radio buttons (left) are appropriate when only one selection is permitted. Checkboxes (right) are best when users may choose any number of choices, from none to all of them.*

You can decide which button is checked when the form loads by adding the **checked** attribute to the **input** element. In this example, the button next to "under 24" will be checked by default (see the note).

Checkbox buttons

`<input type="checkbox">`

Checkbox button

Checkboxes are added using the **input** element with its type set to **checkbox**. As with radio buttons, you create groups of checkboxes by assigning them the same **name** value. The difference, as we've already noted, is that more than one checkbox may be checked at a time. The value of every checked button will be sent to the server when the form is submitted. Here is an example of a group of checkbox buttons used to indicate musical interests. Figure 9-11 shows how they look in the browser:

```
<p>What type of music do you listen to?</p>
<ul>
  <li><input type="checkbox" name="genre" value="punk" checked> Punk
rock</li>
  <li><input type="checkbox" name="genre" value="indie" checked> Indie
rock</li>
  <li><input type="checkbox" name="genre" value="hiphop"> Hip Hop</li>
  <li><input type="checkbox" name="genre" value="rockabilly">
Rockabilly</li>
</ul>
```

Checkboxes don't necessarily need to be used in groups, of course. In this example, a single checkbox is used to allow visitors to opt in for special promotions. The value of the control will be passed along to the server only if the user checks the box.

```
<p><input type="checkbox" name="OptIn" value="yes"> Yes, send me news
and special promotions by email.</p>
```

In Exercise 9-2, you'll get a chance to add both radio and checkbox buttons to the contest entry form.

exercise 9-2 | Adding radio buttons and checkboxes

The next two questions in the sneaker contest entry form use radio buttons and checkboxes for selecting options. Open the ***contest_entry.html*** document and follow these steps.

1. In the Custom Shoe Design section, there are lists of color and feature options. The Color options should be radio buttons because shoes can be only one color. Insert a radio button before each option. Follow this example for the remaining color options.

   ```
   <li><input type="radio"
   name="color" value="red">
   Red</li>
   ```

2. Mark up the Features options as you did the Color options, but this time, the **type** should be **checkbox**. Be sure the variable name for each is **features[]**, and that the metallic logo option is preselected, as noted on the sketch.
3. Save the document and check your work by opening it in a browser to make sure it looks right, then submit the form to make sure it's functioning properly.

Menus

Another way to provide a list of choices is to put them in a drop-down or scrolling menu. Menus tend to be more compact than groups of buttons and checkboxes.

`<select>...</select>`
Menu control

`<option>...</option>`
An option within a menu

`<optgroup>...</optgroup>`
A logical grouping of options within a menu

You add both drop-down and scrolling menus to a form with the **select** element. Whether the menu pulls down or scrolls is the result of how you specify its size and whether you allow more than one option to be selected. Let's take a look at both menu types.

Drop-down menus

The **select** element displays as a drop-down menu (also called a pull-down menu) by default when no size is specified or if the **size** attribute is set to 1. In pull-down menus, only one item may be selected. Here's an example (shown in Figure 9-12):

```
<p>What is your favorite 80s band?
<select name="EightiesFave">
   <option>The Cure</option>
   <option>Cocteau Twins</option>
   <option>Tears for Fears</option>
   <option>Thompson Twins</option>
   <option value="EBTG">Everything But the Girl</option>
   <option>Depeche Mode</option>
   <option>The Smiths</option>
   <option>New Order</option>
</select>
</p>
```

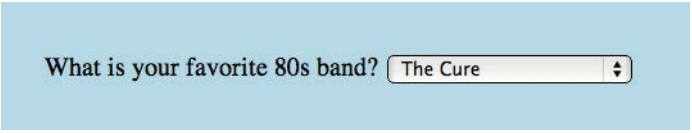


Figure 9-12. Pull-down menus pop open when the user clicks on the arrow or bar.

You can see that the **select** element is just a container for a number of **option** elements. The content of the chosen **option** element is what gets passed to the web application when the form is submitted. If for some reason you want to send a different value than what appears in the menu, use the **value** attribute to provide an overriding value. For example, if someone selects "Everything But the Girl" from the sample menu, the form submits the value "EBTG" for the "EightiesFave" variable. For the others, the content between the **option** tags will be sent as the value.

You will make a menu like this one for selecting a shoe size in Exercise 9-3.

Scrolling menus

To make the menu display as a scrolling list, simply specify the number of lines you'd like to be visible using the **size** attribute. This example menu has

the same options as the previous one, except it has been set to display as a scrolling list that is six lines tall (Figure 9-13).

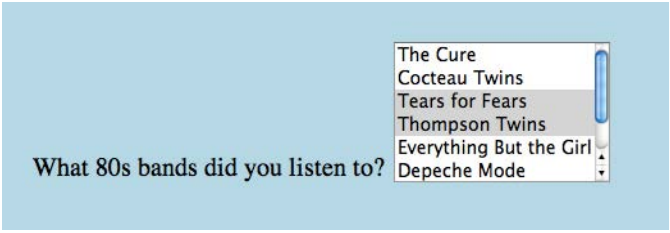


Figure 9-13. A scrolling menu with multiple options selected.

```
<p>What 80s bands did you listen to?
<select name="EightiesBands" size="6" multiple>
   <option>The Cure</option>
   <option>Cocteau Twins</option>
   <option selected>Tears for Fears</option>
   <option selected>Thompson Twins</option>
   <option value="EBTG">Everything But the Girl</option>
   <option>Depeche Mode</option>
   <option>The Smiths</option>
   <option>New Order</option>
</select>
</p>
```

You may notice a few new attributes tucked in there. The **multiple** attribute allows users to make more than one selection from the scrolling list. Note that pull-down menus do not allow multiple selections; when the browser detects the **multiple** attribute, it displays a small scrolling menu automatically by default.

Use the **selected** attribute in an **option** element to make it the default value for the menu control. Selected options are highlighted when the form loads. The **selected** attribute can be used with pull-down menus as well.

NOTE

The `label` *attribute in the* `option` *element is not the same as the* `label` *element used to improve accessibility (discussed later in this chapter).*

Grouping menu options

You can use the **optgroup** element to create conceptual groups of options. The required **label** attribute in the **optgroup** element provides the heading for the group. Figure 9-14 shows how option groups are rendered in modern browsers.

```
<select name="icecream" size="7" multiple>
  <optgroup label="traditional">
      <option>vanilla</option>
      <option>chocolate</option>
  </optgroup>
  <optgroup label="fancy">
    <option>Super praline</option>
    <option>Nut surprise</option>
    <option>Candy corn</option>
  </optgroup>
</select>
```

Figure 9-14. Option groups as rendered in a modern browser.

exercise 9-3 | Adding a menu

The only other control that needs to be added to the contest entry is a pull-down menu for selecting a shoe size.

1. Insert a **select** menu element with the shoe sizes (5 to 13).

```
<p>Size (sizes reflect men's sizing):
  <select name="size" size="1">
    <option>5</option>
    ...insert more options here...
  </select>
</p>
```

2. Save the document and check it in a browser. You can submit the form, too, to be sure that it's working. You should get the Thank You response page listing all of the information you entered in the form.

Congratulations! You've built your first working web form. In Exercise 9-4, we'll add markup that makes it more accessible to assistive devices. But first, we have a few more control types to cover.

File selection control

`<input type="file">`
File selection field

Web forms can collect more than just data. They can also be used to transmit external documents from a user's hard drive. For example, a printing company could use a web form to upload artwork for a business card order. A magazine could use a form on their site to collect digital photos for a photo contest.

The file selection control makes it possible for users to select a document from the hard drive to be submitted with the form data. It is added to the form using our old friend the **input** element with its **type** set to **file**.

The markup sample here and Figure 9-15 show a file selection control used for photo submissions.

```
<form action="/client.php" method="POST" enctype="multipart/form-data">
  <label>Send a photo to be used as your online icon
  <em>(optional)</em><br>
  <input type="file" name="photo" size="28"><label>
</form>
```

The file upload widget varies by browser and operating system. It may be a text field with a button to browse the hard drive, as Firefox does (Figure 9-15, top) or it might be just a button, which is how Chrome displays it (bottom).

It is important to note that when a form contains a file selection input element, you must specify the encoding type (**enctype**) of the form as **multipart/form-data** and use the POST method. The **size** attribute in this example sets the character width of the text field (although it could also be controlled with a CSS rule) if the browser displays one.

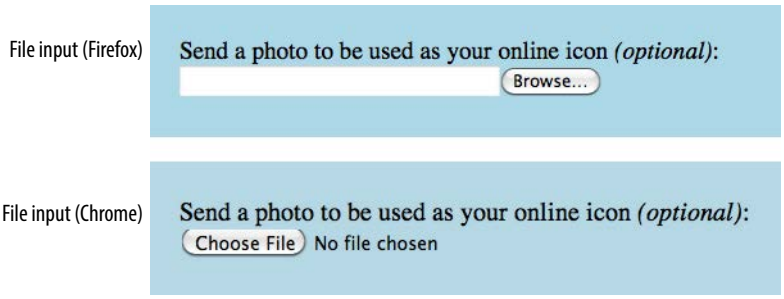


Figure 9-15. A file selection form field.

Hidden controls

`<input type="hidden">`
Hidden control field

There may be times when you need to send information to the form processing application that does not come from the user. In these instances, you can use a hidden form control that sends data when the form is submitted, but is not visible when the form is displayed in a browser.

Hidden controls are added using the **input** element with the **type** set to **hidden**. Its sole purpose is to pass a name/value pair to the server when the form is submitted. In this example, a hidden form element is used to provide the location of the appropriate thank-you document to display when the transaction is complete.

```
<input type="hidden" name="success-link" value="http://www.example.com/
littlechair_thankyou.html">
```

WARNING

It is possible for users to access and manipulate hidden form controls. If you should become a professional web developer, you will learn to program defensively for this sort of thing.

I've worked with forms that have had dozens of hidden controls in the **form** element before getting to the parts that the user actually fills out. This is the kind of information you get from the application programmer, system administrator, or whoever is helping you get your forms processed. If you are using a canned script, be sure to check the accompanying instructions to see if any hidden form variables are required.

Date and time controls (HTML5)

If you've ever booked a hotel or a flight online, you've no doubt used a little calendar widget for choosing the date. Chances are that little calendar was created using JavaScript. HTML5 introduced six new input types that make date and time selection widgets part of a browser's standard built-in display capabilities (just as they can display checkboxes, pop-up menus, and other widgets today). The date and time pickers are implemented on only a few browsers as of this writing, such as Opera, shown in Figure 9-16, but on non-supporting browsers, the date and time input types display as a perfectly usable text entry field instead.

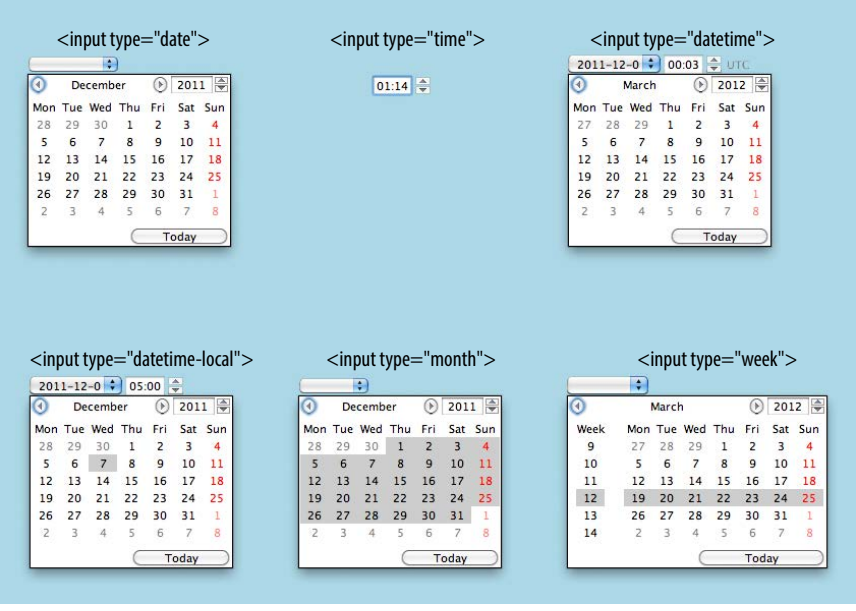


Figure 9-16. Date and time picker inputs in (Opera 11 on Mac OS X).

`<input type="date">`
Date input control
NEW IN HTML5

`<input type="time">`
Time input control
NEW IN HTML5

`<input type="datetime">`
Date/time control with time zone
NEW IN HTML5

`<input type="datetime-local">`
Date/time control with no time zone
NEW IN HTML5

`<input type="month">`
Specifies a month in a year
NEW IN HTML5

`<input type="week">`
Specifies a particular week in a year
NEW IN HTML5

The new date- and time-related input types are as follows:

`<input type="date" name="`*name*`" value="`*2004-01-14*`">`

Creates a date input control, such as a pop-up calendar, for specifying a date (year, month, day). The initial value must be provided in ISO date format (**`YYYY-MM-DD`**).

`<input type="time" name="`*name*`" value="`*03:13:00*`">`

Creates a time input control for specifying a time (hour, minute, seconds, fractional sections) with no time zone indicated. The value is provided as **`hh:mm:ss`**.

`<input type="datetime" name="name" value="2004-01-14T03:13:00-5:00">`

Creates a combined date/time input control that includes time zone information. The value is an ISO-formatted date and time with time zone relative to GMT, as we saw for the **time** element in Chapter 5 (**YYYY-MM-DDThh:mm:ssTZD**).

`<input type="datetime-local" name="name" value="2004-01-14T03:13:00">`

Creates a combined date/time input control with no time zone information (**YYYY-MM-DDThh:mm:ss**).

`<input type="month" name="name" value="2004-01">`

Creates a date input control specifying a particular month in a year (**YYYY-MM**).

`<input type="week" name="name" value="2004-W2">`

Creates a date input control for specifying a particular week in a year using an ISO week numbering format (**YYYY-W#**).

Numerical inputs (HTML5)

`<input type="number">`

Number input

NEW IN HTML5

`<input type="range">`

Slider input

NEW IN HTML5

The **number** and **range** input types collect numerical data. For the **number** input, the browser may supply a spinner widget for selecting a specific numerical value (a text input may display in user agents that don't support the input type). The **range** input is typically displayed as a slider (Figure 9-17) that allows the user to select a value within a specified range.

```
<label>Number of guests <input type="number" name="guests" min="1"
max="6"></label>

<label>Satisfaction (0 to 10) <input type="range" name="satis" min="0"
max="10" step="1"></label>
```

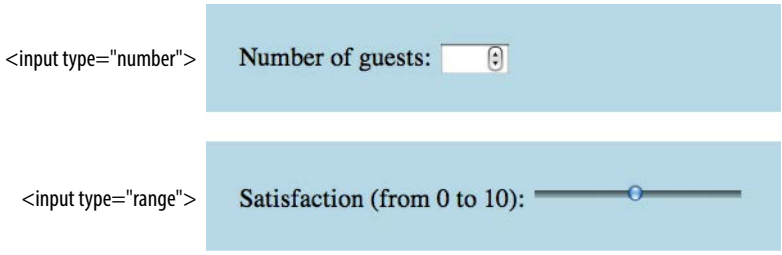


Figure 9-17. The number *and* range *HTML5 input types (in Opera 11 on Mac OS X).*

Both the **number** and **range** input types accept the **min** and **max** attributes for specifying the minimum and maximum values allowed for the input (again, the browser could check that the user input complies with the constraint). Both **min** and **max** are optional, and you can also set one without the other.

The **step** attribute allows developers to specify the acceptable increments for numerical input. The default is 1. A value of .5 would permit values 1, 1.5, 2, 2.5, etc.; a value of 100 would permit 100, 200, 300, and so on. You can also set the **step** attribute to **any** to explicitly accept any value increment.

Again, browsers that don't support these new input types display a standard text input field instead, which is a fine fallback.

Color selector (HTML5)

`<input type="color">`

Color picker

NEW IN HTML5

The intent of the color control type is to create a pop-up color picker for visually selecting a color value similar to those used in operating systems or image-editing programs. Values are provided in hexadecimal RGB values (#RRGGBB). Figure 9-18 shows the color picker widget in Opera 11. Non-supporting browsers display the default text input instead.

```
<label>Your favorite color <input type="color" name="favorite"></label>
```

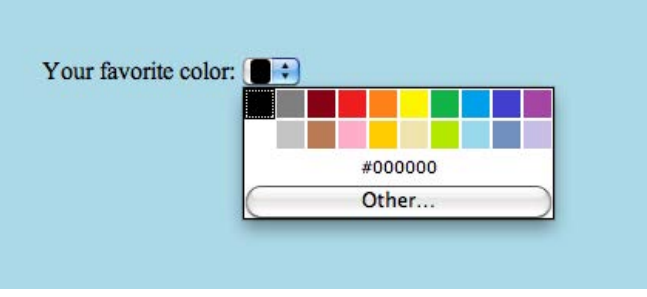


Figure 9-18. The `color` input type (in Opera 11 on Mac OS X).

That wraps up the form control roundup. Learning how to insert form controls is one part of the forms production process, but any web developer worth her salt will take the time to make sure the form is as accessible as possible. Fortunately, there are a few things we can do in markup to describe the form's structure.

A Few More HTML5 Form Elements

For the sake of completeness, let's look at the remaining form elements that are new in HTML5. As of this writing, they are poorly supported, and are somewhat esoteric anyway, so you may wait a while to add these to your HTML toolbox. We've already covered the **datalist** element for providing suggested values for text inputs. HTML5 also introduced the following elements:

progress

`<progress>…</progress>`

Indicates the state of an ongoing process

NEW IN HTML5

The **progress** element gives users feedback on the state of an ongoing process, such as a file download. It can have a specific end value (provided with the **max** attribute) or just indicate that something is happening (such as waiting for a server to respond).

```
Percent downloaded: <progress max="100" name="fave">0</progress>
```

meter

`<meter>…</meter>`

Indicates the state of an ongoing process

NEW IN HTML5

meter is similar to **progress**, but it always represents a measurement within a known range of values (also known as a gauge). It has a number of attributes: **min** and **max** indicate the highest and lowest values for the range; **low** and **high** could be used to trigger warnings at undesirable levels; and **optimum** specifies a preferred value. The values would most likely be updated with JavaScript dynamically during the process.

```
<meter min="0" max="100" name="download">50%</meter>
```

output

`<output>…</output>`

Calculated output value

NEW IN HTML5

Simply put, the **output** element provides a way to indicate the results of a calculation by a script or program and associate it with inputs that affected the calculation.

keygen

`<keygen>`

Key pair generator

NEW IN HTML5

The **keygen** element represents a control for making a key pair (used to ensure privacy). When the form is submitted, the private key is stored locally, and the public key is packaged and sent to the server. Don't worry; I'm a little foggy on what this all means, too. You can read about public-key cryptography (*en.wikipedia.org/wiki/Public-key_cryptography*) and explain it to me when you figure it out.

Form Accessibility Features

It is essential to consider how users without the benefit of visual browsers will be able to understand and navigate through your web forms. The **label**, **fieldset**, and **legend** form elements improve accessibility by making the semantic connections between the components of a form clear. The resulting markup is not only more semantically rich, but there are also more elements available to act as "hooks" for style sheet rules. Everybody wins!

Labels

Although we may see the label "Address" right next to a text field for entering an address in a visual browser, in the source, the label and field input may be separated. The **label** element associates descriptive text with its respective form field. This provides important context for users with speech-based browsers.

Each **label** element is associated with exactly one form control. There are two ways to use it. One method, called implicit association, nests the control and its description within a **label** element. In the following example, **label**s are assigned to individual checkboxes and their related text descriptions. (By the way, this is the way to label radio buttons and checkboxes. You can't assign a label to the entire group.)

```
<ul>
  <li><label><input type="checkbox" name="genre" value="punk"> Punk
rock</label></li>
  <li><label><input type="checkbox" name="genre" value="indie"> Indie
rock</label></li>
  <li><label><input type="checkbox" name="genre" value="hiphop"> Hip
Hop</label></li>
  <li><label><input type="checkbox" name="genre" value="rockabilly">
Rockabilly</label></li>
</ul>
```

The other method, called explicit association, matches the label with the control's **id** reference. The **for** attribute says which control the label is for. This approach is useful when the control is not directly next to its descriptive text in the source. It also offers the potential advantage of keeping the label and the control as two distinct elements, which may come in handy when aligning them with style sheets.

```
<label for="form-login-username">Login account</label>
<input type="text" name="login" id="form-login-username">

<label for="form-login-password">Password</label>
<input type="password" name="password" id="form-login-password">
```

Another advantage to using labels is that users can click or tap anywhere on them to select the form element. Users with touch devices will appreciate the larger tap target.

WARNING

iOS devices as of this writing do not make implicit labels clickable, so that behavior needs to be created with JavaScript. I know we haven't done any JavaScript yet, but if you are wondering, the fix looks like this:

```
document.getElementsByTagName
('label').setAttribute
('onclick','');
```

TIP

To keep form-related **id**s distinct from other **id**s on the page, consider prefacing them with "form-" as shown in the examples.

Another technique for keeping forms organized is to give the form element an ID name and include it as a prefix in the IDs for the controls it contains as follows:

```
<form id="form-login">

<input id="form-login-
username">

<input id="form-login-
password">
```

fieldset and legend

The **fieldset** element indicates a logical group of form controls. A **fieldset** may also include a **legend** element that provides a caption for the enclosed fields.

Figure 9-19 shows the default rendering of the following example, but you could use style sheets to change the way the **fieldset** and **legend** appear.

```
<fieldset>
  <legend>Mailing List Sign-up</legend>
  <ul>
    <li><label>Add me to your mailing list <input type="radio"
    name="list" value="yes" checked="checked"></label></li>
    <li><label>No thanks <input type="radio" name="list" value="no">
  </label></li>
  </ul>
</fieldset>

<fieldset>
  <legend>Customer Information</legend>
  <ol>
    <li><label>Full name: <input type="text" name="username"></label></
li>
    <li><label>Email: <input type="text" name="email"></label></li>
    <li><label>State: <input type="text" name="state"></label></li>
  </ol>
</fieldset>
```

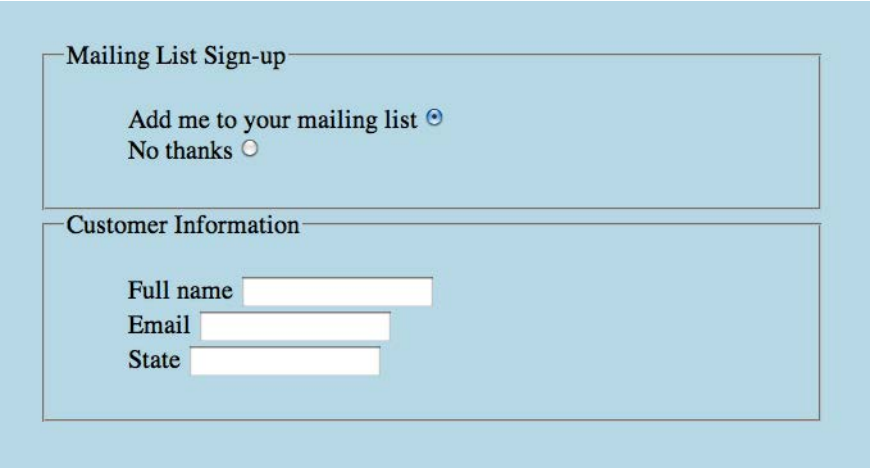


Figure 9-19. The default rendering of fieldsets and legends.

WARNING

Fieldsets and legends tend to throw some curveballs when it comes to styling. For example, background colors in fieldsets are handled differently from browser to browser. Legends are unique in that their text doesn't wrap. The solution is to put a **span** *or* **b** *element in them and control presentation of the contained element without sacrificing accessibility. Be sure to do lots of testing if you style these form elements.*

exercise 9-4 | labels and fieldsets

Our contest form is working, but we need to label it appropriately and create some **fieldset**s to make it more usable on assistive devices. Once again, open the ***contest_entry.html*** document and follow these steps.

I like to start with the broad strokes and fill in details later, so we'll begin this exercise by organizing the form controls into fieldsets, and then we'll do all the labeling. You could do it the other way around, and ideally, you'd just mark up the labels and fieldsets as you go along instead of adding them all later.

1. The "Contest Entry Information" at the top of the form is definitely conceptually related, so let's wrap it all in a **fieldset** element. Change the markup of the section title from a paragraph (**p**) to a legend for the fieldset.

```
<fieldset>
  <legend>Contest Entry Information</legend>
  <ul>
    <li>Name: <input type="text"
name="username"></li>
    …
  </ul>
</fieldset>
```

2. Next, group the Color, Features, and Size questions in a big fieldset with the legend "Custom Shoe Design" (the text is there; you just need to change it from a **p** to a **legend**).

```
<h2>Design your custom Forcefields:</h2>
<fieldset>
<legend>Custom Shoe Design</legend>
  Color…
  Features…
  Size…
</fieldset>
```

3. Create another fieldset just for the Color options, again changing the description in a paragraph to a **legend**. Do the same for the Features and Size sections. In the end, you will have three fieldsets contained within the larger "Custom Shoe Design" fieldset. When you are done, save your document and open it in a browser. It should now look very close to the final form shown in Figure 9-2, given the expected browser differences.

```
<fieldset>
<legend>Color <em>(choose one)</em>:</legend>
   <ul>…</ul>
</fieldset>
```

4. OK, now let's get some labels in there. In the Contest Entry Information fieldset, explicitly tie the label to the text input using the **for**/**id** label method. I've done the first one for you; you do the other three.

```
<li><label for="form-name">Name:</label> <input
type="text" name="username" id="form-name"></li>
```

5. For the radio and checkbox buttons, wrap the **label** element around the **input** and its value label. In this way, the button will be selected when the user clicks or taps anywhere inside the label element. Here is the first one; you do the other seven.

```
<li><label><input type="radio" name="color"
value="red"> Red</label></li>
```

Save your document, and you're done! Labels don't have any effect on how the form looks by default, but you can feel good about the added semantic value you've added and maybe even use them to apply styles at another time.

Form Layout and Design

I can't close this chapter without saying a few words about form design, even though this chapter is about markup, not presentation.

Usable forms

A poorly designed form can ruin a user's experience on your site and negatively impact your business goals. Badly designed forms mean lost customers, so it is critical to get it right—both on the desktop and for small-screen devices with their special requirements. You want the path to a purchase or other action to be as frictionless as possible.

The topic of good web form design is a rich one that could fill a book in itself. In fact, there is such a book: *Web Form Design* (Rosenfeld Media, 2008) by web form expert Luke Wroblewski, and I recommend it highly. Luke's subsequent book, *Mobile First* (A Book Apart, 2011), includes tips for how to format forms in a mobile context. You can browse over a hundred articles about forms on his site here: *www.lukew.com/ff?tag=forms*.

Here I'll offer just a very small sampling of tips from *Web Form Design* to get you started, but the whole book is worth a read.

Avoid unnecessary questions.

Help your users get through your form easily as possible by not including questions that are not absolutely necessary to the task at hand. Extra questions, in addition to slowing things down, may make a user wary of your motivations for asking. If you have another way of getting the information (for example, the type of credit card can be determined from the first four numbers of the account), then use alternative means and don't put the burden on the user. If there is information that might be nice to have but is not required, consider asking at a later time, after the form has been submitted and you have built a relationship with the user.

Consider impact of label placement.

The position of the label relative to the input affects the time it takes to fill out the form. The less the user's eye needs to bounce around the page, the quicker the form completion. Putting the labels above their respective fields creates a single alignment for faster scans and completion, particularly when asking for familiar information (username, address, etc.). Top-positioned labels can also accommodate labels of varying lengths and work best on narrow, small-screen devices. They do result in a longer form, however, so if vertical space is a concern, you can position the labels to the left of the inputs. Left alignment of labels results in the slowest form completion, but it may be appropriate if you want the user to slow down or be able to scan and consider the types of information required in the form.

Choose input types carefully.

As you've seen in this chapter, there are quite a few input types to choose from, and sometimes it's not easy to decide which one to use. For example, a list of options could be presented as a pull-down menu or a number of choices with checkboxes. Weigh the pros and cons of each control type carefully, and follow up with user testing.

Group related inputs.

It is easier to parse the many fields, menus, and buttons in a form if they are visually grouped by related topic. For example, a user's contact information could be presented in a compact group so that five or six inputs are perceived as one unit. Usually, all you need is a very subtle indication, such as a fine horizontal rule and some extra space. Don't overdo it.

Clarify primary and secondary actions.

The primary action at the end of the form is usually some form of Submit button ("Buy," "Register," etc.) that signals the completion of the form and the readiness to move forward. You want that button to be visually dominant and easy to find (aligning it along the main axis of the form alignment is helpful as well). Secondary actions tend to take you a step back, such as clearing or resetting the form. If you must include a secondary action, make sure that it is styled to look different and less important than the primary action. It is also a good idea to provide an opportunity to undo the action.

Styling Forms

As we've seen in this chapter, the default rendering of form markup is not up to par with the quality we see on most professional web forms today. As for other elements, you can use style sheets to create a clean form layout as well as change the appearance of most form controls. Something as simple as nice alignment and a look that is consistent with the rest of your site can go a long way toward improving the impression you make on a user.

Keep in mind that form widgets are drawn by the browser and are informed by operating system conventions. However, you can still apply dimensions, margins, fonts, colors, borders, and background effects to form elements such as text inputs, select menus, textareas, fieldsets, labels, and legends. Just be sure to test in a variety of browsers to check for unpleasant surprises. Chapter 18, CSS Techniques in Part III lists some specific techniques once you have more experience with CSS. For more help, a web search for "CSS for forms" will turn up a number of tutorials.

Test Yourself

Ready to put your web form know-how to the test? Here are a few questions to make sure you've gotten the basics.

1. Decide whether each of these forms should be sent via the GET or POST method:
   a. A form for accessing your bank account online _____
   b. A form for sending t-shirt artwork to the printer _____
   c. A form for searching archived articles _____
   d. A form for collecting long essay entries _____

2. Which form control element is best suited for the following tasks? When the answer is "input," be sure to also include the type. Some tasks may have more than one correct answer.

   a. Choose your astrological sign from 12 signs.

   b. Indicate whether you have a history of heart disease (yes or no).

   c. Write up a book review.

   d. Select your favorite ice cream flavors from a list of eight flavors.

   e. Select your favorite ice cream flavors from a list of 25 flavors.

3. Each of these markup examples contains an error. Can you spot what it is?

   a. `<input name="country" value="Your country here.">`

   b. `<checkbox name="color" value="teal">`

   c.
```
<select name="popsicle">
  <option value="orange">
  <option value="grape">
  <option value="cherry">
</select>
```

   d. `<input type="password">`

   e. `<textarea name="essay" height="6" width="100">Your story.</textarea>`

Element Review: Forms

We covered this impressive list of elements and attributes related to forms in this chapter. Elements marked with (HTML5) are new in the HTML5 specification.

Element and attributes	Description
`button`	Generic input button
`name="`*`text`*`"`	Supplies a unique variable name for the control
`type="submit|reset|button"`	The type of custom button
`value="`*`text`*`"`	Specifies the value to be sent to the server
`datalist` *[HTML5]*	Provides a list of options for text inputs
`fieldset`	Groups related controls and labels
`form`	Form element
`action="`*`url`*`"`	Location of forms processing program (*required*)
`method="get|post"`	The method used to submit the form data
`enctype="`*`content type`*`"`	The encoding method, generally either **`application/x-www-form-urlencoded`** (default) or **`multipart/form-data`**

Element and attributes	Description
`input`	Creates a variety of controls, based on the **`type`** value
`autofocus`	Indicates the control should be ready for input when the document loads
`type="submit|reset|button|text|password|checkbox|radio|image|file|hidden|email|tel|search|url|date|time|datetime|datetime-local|month|week|number|range|color "`	The type of input
`disabled`	Associates the control with a specified form
`form="`*`form id value`*`"`	Disables the input so it cannot be selected
See Table 9-1 for a full list of attributes associated with each input type.	
`keygen` *[HTML5]*	Generates key pairs for secure transaction certificates
`autofocus`	Indicates the control should be highlighted and ready for input when the document loads
`challenge="`*`challenge string`*`"`	Provides a challenge string to be submitted with the key
`disabled`	Disables the control so it cannot be selected
`form="`*`form id value`*`"`	Associates the control with a specified form
`keytype="`*`keyword`*`"`	Identifies the type of key to be generated (e.g., **`rsa`** or **`ec`**)
`name="`*`text`*`"`	Gives control an identifying name
`label`	Attaches information to controls
`for="`*`text`*`"`	Identifies the associated control by its **`id`** reference
`form="`*`form id value`*`"`	Associates the control with a specified form
`legend`	Assigns a caption to a **`fieldset`**
`meter` *[HTML5]*	Represents a fractional value within a known range
`form="`*`form id value`*`"`	Associates the control with a specified form
`high="`*`number`*`"`	Indicates the range that is considered "high" for the gauge
`low="`*`number`*`"`	Indicates the range that is considered "low" for the gauge
`max="`*`number`*`"`	Specifies the highest value for the range
`min="`*`number`*`"`	Specifies the lowest value for the range
`optimum="`*`number`*`"`	Indicates the number considered to be "optimum"
`value="`*`number`*`"`	Specifies the actual or measured value
`optgroup`	Defines a group of options
`disabled`	Disables the **`optgroup`** so it cannot be selected
`label="`*`text`*`"`	Supplies label for a group of options
`option`	An option within a select menu control
`disabled`	Disables the **`option`** so it cannot be selected
`label="`*`text`*`"`	Supplies an alternate label for the option
`selected`	Preselects the option
`value="`*`text`*`"`	Supplies an alternate value for the option

Element and attributes	Description
`output` *[HTML5]*	Represents the results of a calculation
`for="`*text*`"`	Creates relationship between output and another element
`form="`*form id value*`"`	Associates the control with a specified form
`name="`*text*`"`	Supplies a unique variable name for the control
`progress` *[HTML5]*	Represents the completion progress of a task (can be used even if the maximum value of the task is not known)
`form="`*form id value*`"`	Associates the control with a specified form
`max="`*number*`"`	Specifies the total value or final size of the task
`value="`*number*`"`	Specifies how much of the task has been completed
`select`	Pull-down menu or scrolling list
`autofocus`	Indicates the control should be highlighted and ready for input when the document loads
`disabled`	Indicates the control is nonfunctional. Can be activated with a script.
`form="`*form id value*`"`	Associates the control with a specified form
`multiple`	Allows multiple selections in a scrolling list
`name="`*text*`"`	Supplies a unique variable name for the control
`readonly`	Makes the control unalterable by the user
`required`	Indicates the user input is required for this control
`size="`*number*`"`	The height of the scrolling list in text lines
`textarea`	Multiline text entry field
`autofocus`	Indicates the control should be highlighted and ready for input when the document loads
`cols="`*number*`"`	The width of the text area in characters
`dirname="`*text*`"`	Allows text directionality to be specified
`disabled`	Disables the control so it cannot be selected
`form="`*form id value*`"`	Associates the control with a specified form
`maxlength="`*text*`"`	Specifies the maximum number of characters the user can enter
`name="`*text*`"`	Supplies a unique variable name for the control
`placeholder="`*text*`"`	Provides a short hint to help user enter the correct data
`readonly`	Makes the control unalterable by the user
`required`	Indicates user input is required for this control
`rows="`*number*`"`	The height of the text area in text lines
`wrap="hard|soft"`	Controls whether line breaks in the text input are returned in the data. **`hard`** preserves line breaks; **`soft`** does not.

Table 9-1. Available attributes for each input type

	submit	reset	button	text	password	checkbox	radio	image	file	hidden
accept									•	
alt								•		
checked						•	•			
disabled	•	•	•	•	•	•	•	•	•	•
maxlength				•	•				•	
name	•	•	•	•	•	•	•	•	•	•
readonly				•	•	•	•		•	
size				•	•				•	
src								•		
value	•	•	•	•	•	•	•		•	•
HTML5-only										
autocomplete				•	•					
autofocus	•	•	•	•	•	•	•	•	•	
form	•	•	•	•	•	•	•	•	•	•
formaction	•							•		
formenctype	•							•		
formmethod	•							•		
formnovalidate	•							•		
formtarget	•							•		
height								•		
list				•						
max										
min										
multiple									•	
pattern				•	•					
placeholder				•	•					
required				•	•	•	•		•	
step										
width								•		

	email	telephone, search, url	number	range	date, time, datetime, datetime-local, month, week	color
accept						
alt						
checked						
disabled	•	•	•	•	•	•
maxlength	•	•				
name	•	•	•	•	•	•
readonly	•	•	•		•	
size	•	•				
src						
value	•	•	•	•	•	•
HTML5-only						
autocomplete	•	•	•	•	•	•
autofocus	•	•	•	•	•	•
form	•	•	•	•	•	•
formaction						
formenctype						
formmethod						
formnovalidate						
formtarget						
height						
list	•	•	•	•	•	•
max			•	•	•	
min			•	•	•	
multiple	•					
pattern	•	•				
placeholder	•	•				
required	•	•	•		•	
step			•	•	•	
width						

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>
```

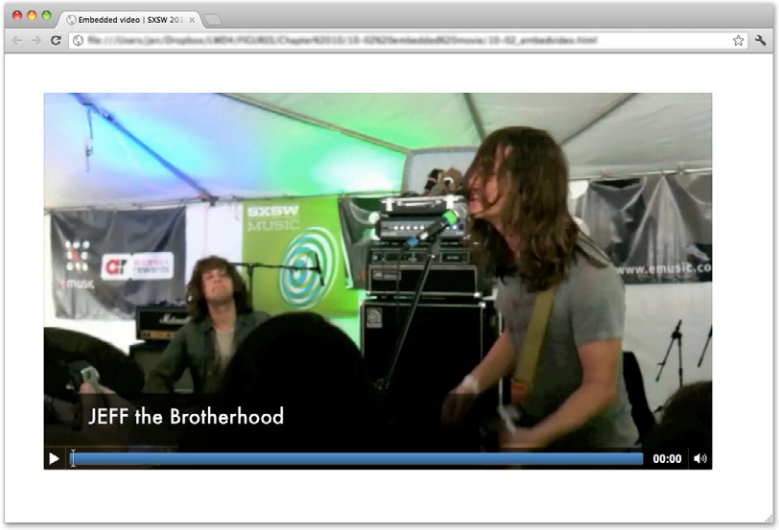


Figure 10-2. An embedded movie using the video element (shown in Chrome on Mac).

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

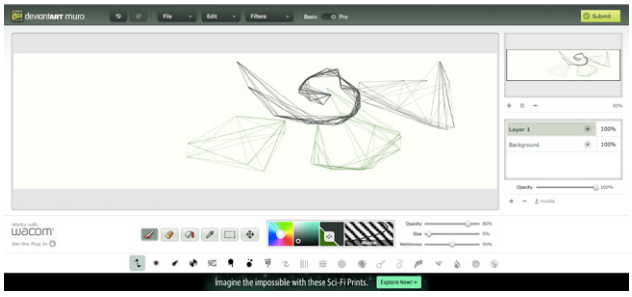
display an illustration, but what gives the **canvas** element so much potential (and has all the web development world so delighted) is that it's all generated with scripting. That means it is dynamic and can draw things on the fly and respond to user input. This makes it a nifty platform for creating 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.

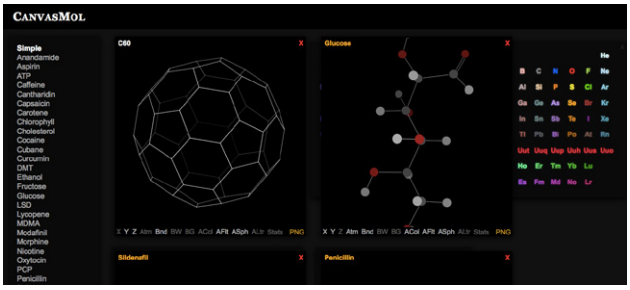
Figure 10-3 shows a few examples of the **canvas** element used to create games, drawing programs, an interactive molecule structure tool, and an asteroid animation. You can find more examples at *Canvasdemos.com*.

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"?