II

HTML FOR STRUCTURE

CHAPTER

4

CREATING A SIMPLE PAGE

(HTML OVERVIEW)

IN THIS CHAPTER

An introduction to elements and attributes

Marking up a simple web page

The elements that provide document structure

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 how HTML documents are structured.
- 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 set up areas for content and metadata.

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 how the text looks with a style sheet. This exercise gives you a taste of formatting content with Cascading Style Sheets.

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

FIGURE 4-1. In this chapter, we'll write the HTML document for this page in five steps.

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's helpful to see the cause and effect of each small change to the source file along the way.

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 visual 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"**).

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, and 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 visual or drag-and-drop 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.

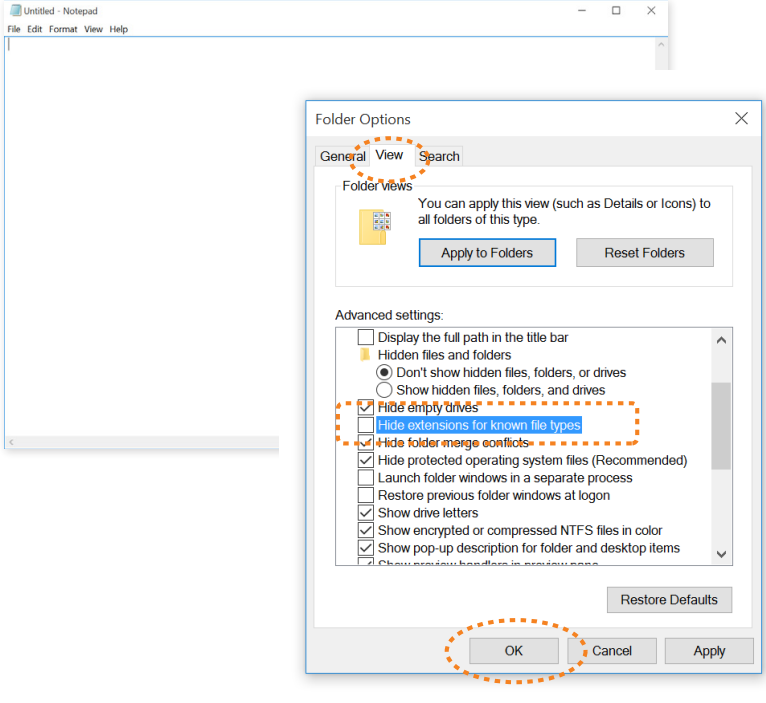
And for what it's worth, professional web developers tend to mark up content manually for better control over the code and the ability to make deliberate decisions about what elements to use.

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.

Creating a New Document in Notepad (Windows)

These are the steps to creating a new document in Notepad on Windows 10 (FIGURE 4-2):

1. Search for "Notepad" to access it quickly. Click on Notepad to open a new document window, and you're ready to start typing. ❶
2. Next, 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. Open the File Explorer, select the View tab, and then select the Options button on the right. In the Folder Options panel, select the View tab again. ❷
3. Find "Hide extensions for known file types" and uncheck that option. ❸
4. Click OK to save the preference ❹, and the file extensions will now be visible.



❶ Click on Notepad to open a new document.

❷ Open the File Explorer, select the View tab, and then select the Options button on the right (not shown). Select the View tab.

❸ Uncheck "Hide extensions for known file types."

❹ Click OK to save the preference, and the file extensions will now be visible.

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

Creating a New Document in TextEdit (macOS)

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. In the initial TextEdit dialog box, click the New Document button in the bottom-left corner. If you see the text formatting menu and tab ruler at the top of the Untitled document, you are in rich-text mode ❶. If you don't, you are in plain-text mode ❷. Either way, there are some preferences you need to set.
3. Close that document, and open the Preferences dialog box from the TextEdit menu.
4. Change these preferences:

   On the New Document tab, select Plain text ❸. Under Options, deselect all of the automatic formatting options ❹.

   On the Open and Save tab, select Display HTML files as HTML Code ❺ and deselect "Add '.txt' extensions to plain text files" ❻. The rest of the defaults should be fine.
5. When you are done, click the red button in the top-left corner.
6. Now create a new document by selecting File → New. 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.

❶ Formatting menu indicates rich text.

❷ Plain text documents have no menu.

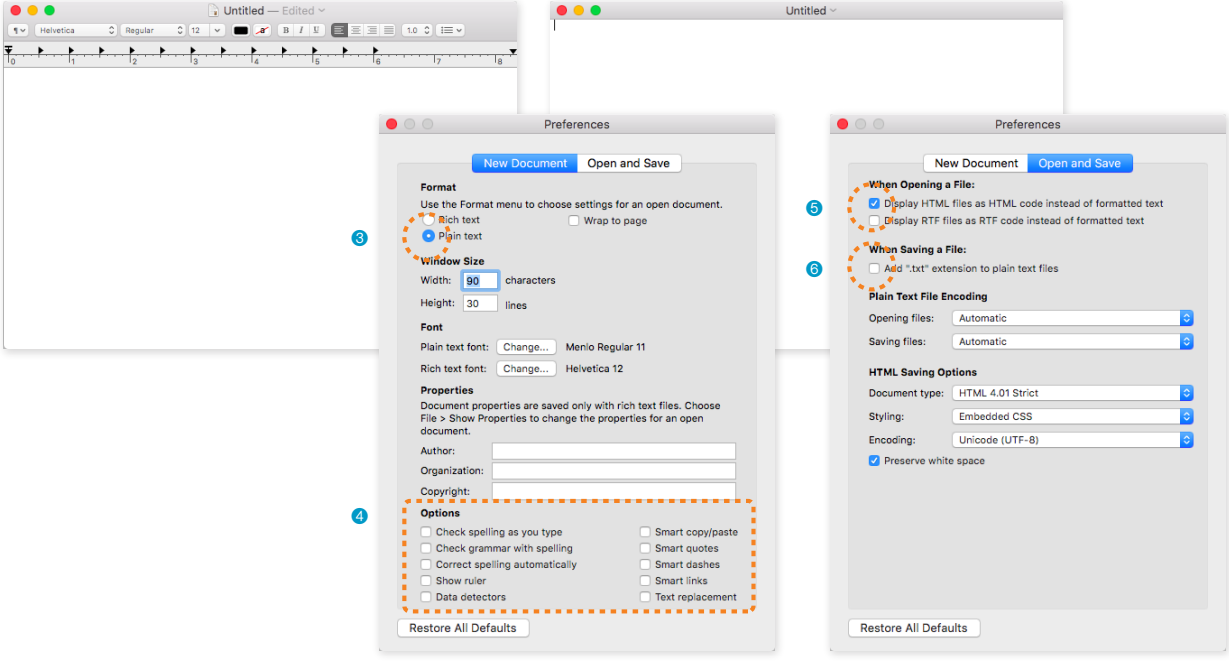


FIGURE 4-3. Launching TextEdit and choosing "Plain text" settings in the Preferences.

STEP 1: START WITH CONTENT

Now that we have our new document, it's time to get typing. A web page is all about 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 home page content below 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 also available online at *learningwebdesign.com/5e/materials/*.

```
Black Goose Bistro

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

Catering
You have fun. We'll handle the cooking. Black Goose Catering can handle
events from snacks for a meetup 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 (click the New Folder button on both Windows and Mac) that will contain all of the files for the site. The technical name for the folder that contains everything is the local root directory.

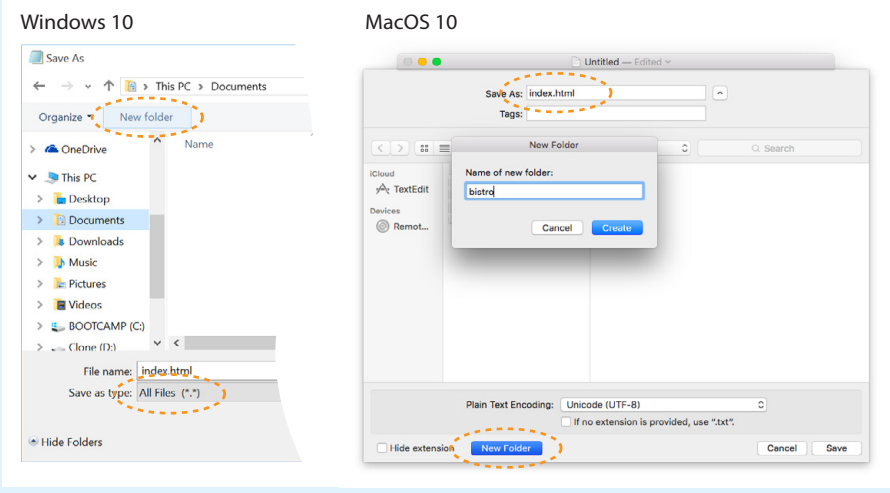


FIGURE 4-4. Saving *index.html* in a new folder called *bistro*.

Name the new folder *bistro*, and save the text file as *index.html* in it. 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.

   Windows users: Double-click the filename in the File Explorer to launch your default browser, or right-click the file for the option to open it in the browser of your choice.

   Mac users: 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.

4. You should see something like the page shown in FIGURE 4-5. We'll talk about the results in the following section.

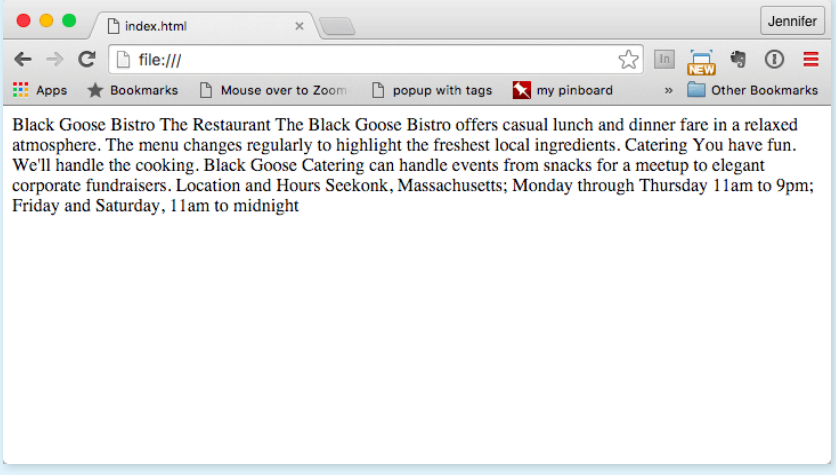


FIGURE 4-5. A first look at the content in a browser.

Naming Conventions

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

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

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. It is also best to avoid international characters, such as the Swedish å.

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

Keep filenames short. Long names are more likely to be misspelled, and short names shave a few extra bytes off the file size. 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.

Learning from Step 1

Our page isn't looking so good (FIGURE 4-5). The text is all run together into one block—that's not how it looked when we typed it into the original document. There are a couple of lessons 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 types of information in the source document that are 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.

STEP 2: GIVE THE HTML DOCUMENT STRUCTURE

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

The Anatomy of an HTML Element

Back in **Chapter 2** you saw examples of elements with an opening tag (`<p>` for a paragraph, for example) and a closing tag (`</p>`). 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.

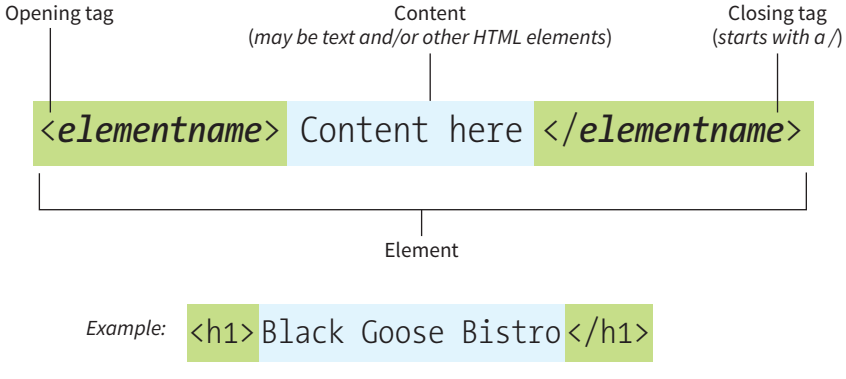


FIGURE 4-6. The parts of an HTML container element.

What Browsers Ignore

The following information in the source document will be ignored when it is viewed in a browser:

Multiple-character (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.

Tabs

Tabs are also converted to character spaces, so guess what? They're useless for indenting text on the web page (although they may make your code more readable).

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 do not display text between the special `<!--` and `-->` tags used to denote a comment. See the upcoming **"Adding Hidden Comments"** sidebar.

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 Versus 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 (it is not case-sensitive). So **<img>**, **<Img>**, and **<IMG>** are all the same as far as the browser is concerned. However, most developers prefer the consistency of writing element names in all lowercase (see **Note**), as I will be doing throughout this book.

Basic Document Structure

FIGURE 4-8 shows the recommended minimal skeleton of an HTML 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 into metadata (**head**) and content (**body**) areas. Let's take a look at what's going on in this minimal markup example.

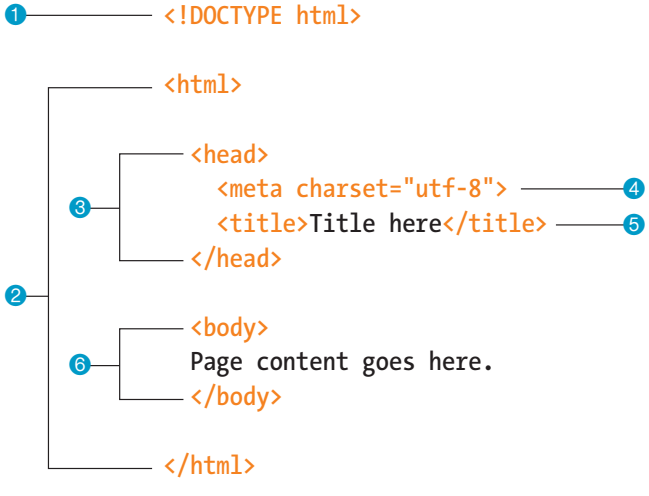


FIGURE 4-8. The minimal structure of an HTML document includes **head** and **body** contained within the **html** root element.

■ MARKUP TIP

Slash Versus Backslash

HTML tags and URLs use the slash character (/). The slash character is found under the question mark (?) on the English QWERTY keyboard (key placement on keyboards in other countries may vary).

It is easy to confuse the slash with the backslash character (\), which is found under the bar character (|); see FIGURE 4-7. The backslash key will not work in tags or URLs, so be careful not to use it.

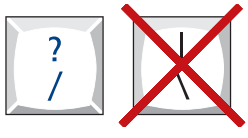


FIGURE 4-7. Slash versus backslash keys.

NOTE

There is a stricter version of HTML called XHTML that requires all element and attribute names to appear in lowercase. HTML5 has made XHTML all but obsolete except for certain use cases when it is combined with other XML languages, but the preference for all lowercase element names has persisted.

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 lets modern browsers know which HTML specification to use to interpret the document. This DOCTYPE identifies the document as written in HTML5.

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.

❸ Within the **`html`** element, the document is divided into a head and a body. The **`head`** element contains elements that pertain to the document that are not rendered as part of the content, such as its title, style sheets, scripts, and metadata.

❹ **`meta`** elements provide document metadata, information about the document. In this case, it specifies the character encoding (a standardized collection of letters, numbers, and symbols) used in the document as Unicode version UTF-8 (see the sidebar **"Introducing Unicode"**). 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 markup. Other types of metadata provided by the **`meta`** element are the author, keywords, publishing status, and a description that can be used by search engines.

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 start marking up the Black Goose Bistro home page? Open the *index.html* document in your text editor and move on to EXERCISE 4-2.

Introducing Unicode

All the characters that make up languages are stored in computers as numbers. A standardized collection of characters with their reference numbers (code points) is called a coded character set, and the way in which those characters are converted to bytes for use by computers is the character encoding. In the early days of computing, computers used limited character sets such as ASCII that contained 128 characters (letters from Latin languages, numbers, and common symbols). The early web used the Latin-1 (ISO 8859-1) character encoding that included 256 Latin characters from most Western languages. But given the web was "worldwide," it was clearly not sufficient.

Enter Unicode. Unicode (also called the Universal Character Set) is a super-character set that contains over 136,000 characters (letters, numbers, symbols, ideograms, logograms, etc.) from all active modern languages. You can read all about it at *unicode.org*. Unicode has three standard encodings—UTF-8, UTF-16, and UTF-32—that differ in the number of bytes used to represent the characters (1, 2, or 3, respectively).

HTML5 uses the UTF-8 encoding by default, which allows wide-ranging languages to be mixed within a single document. It is always a good idea to declare the character encoding for a document with the **`meta`** element, as shown in the previous example. Your server also needs to be configured to identify HTML documents as UTF-8 in the HTTP header (information about the document that the server sends to the user agent). You can ask your server administrator to confirm the encoding of the HTML documents.

EXERCISE 4-2. Adding minimal structure

1. Open the new *index.html* document if it isn't open already and add the DOCTYPE declaration:

```
<!DOCTYPE html>
```

2. Put the entire document in an HTML root element by adding an **<html>** start tag after the DOCTYPE and an **</html>** end tag at the very end of the text.
3. Next, create the document head that contains the title for the page. Insert **<head>** and **</head>** tags before the content. Within the **head** element, add information about the character encoding **<meta charset="utf-8">**, and the title, "Black Goose Bistro", surrounded by opening and closing **<title>** tags.
4. Finally, define the body of the document by wrapping the text 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 relaxed atmosphere. The menu
changes regularly to highlight the freshest local
ingredients.

Catering
You have fun. We'll handle the cooking. Black
Goose Catering can handle events from snacks for a
meetup to elegant corporate fundraisers.

Location and Hours
Seekonk, Massachusetts;
Monday through Thursday 11am to 9pm; Friday and
Saturday, 11am to midnight
</body>
</html>
```

5. 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 4-9 shows how it should look now.

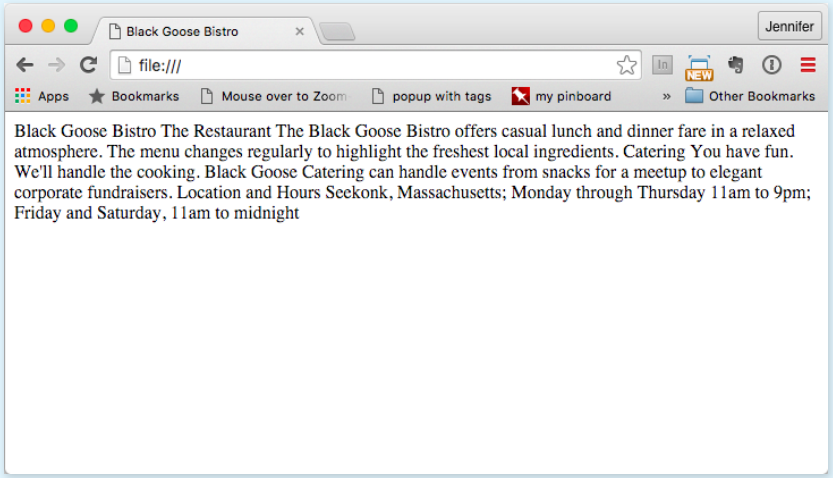


FIGURE 4-9. The page in a browser after the document structure elements have been defined.

Not much has changed in the bistro page after setting up the document, except that the browser now displays the title of the document in the top bar or tab (FIGURE 4-9). If someone were to bookmark this page, that title would be added to their Bookmarks or Favorites list as well (see the sidebar **"Don't Forget a Good Title"**). But the content still runs together because we haven't given the browser any indication of how it should be structured. We'll take care of that next.

STEP 3: IDENTIFY TEXT ELEMENTS

With a little markup experience under your belt, it should be a no-brainer to add the markup for 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.

Mark It Up Semantically

The purpose of HTML is to add meaning and structure to the content. It is *not* intended to describe 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 it looks like... 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 them. You can think of this structure as an outline (its technical name is the DOM, for Document Object Model). The underlying document hierarchy 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.

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, I'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.

Don't Forget a Good Title

A `title` element is not only required for every document, but it is also quite useful. 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 things a person hears when using a screen reader (an assistive device that reads the content of a page aloud for users with impaired sight). 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. Knowing that users typically have a number of tabs open or a long list of Bookmarks, put your most uniquely identifying information in the first 20 or so characters.

The purpose of HTML is to add meaning and structure to the content.

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 relaxed atmosphere. The menu changes regularly to highlight the
freshest local 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 a meetup 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 reload) the page in the browser. You should see a page that looks much like the one in FIGURE 4-10. 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.

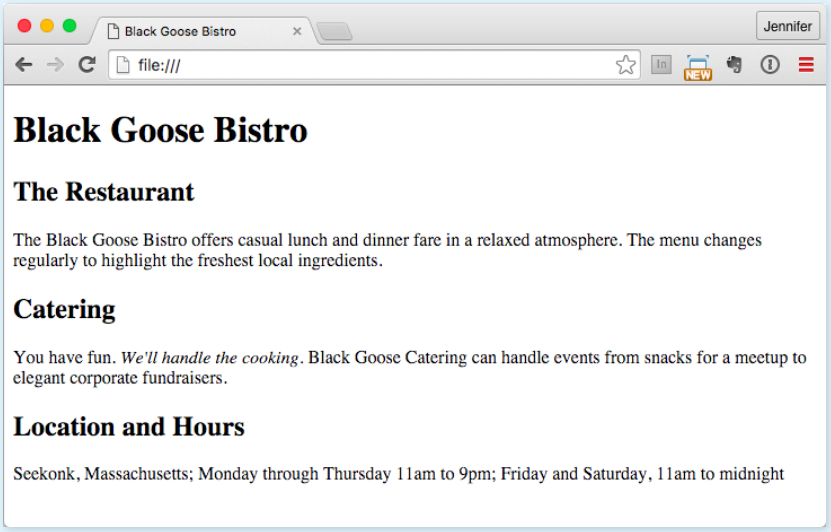


FIGURE 4-10. The home page after the content has been marked up with HTML elements.

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 4-10.

Block and Inline Elements

Although it may seem like stating the obvious, it's 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 4-11, the edges of the block elements are outlined in red.

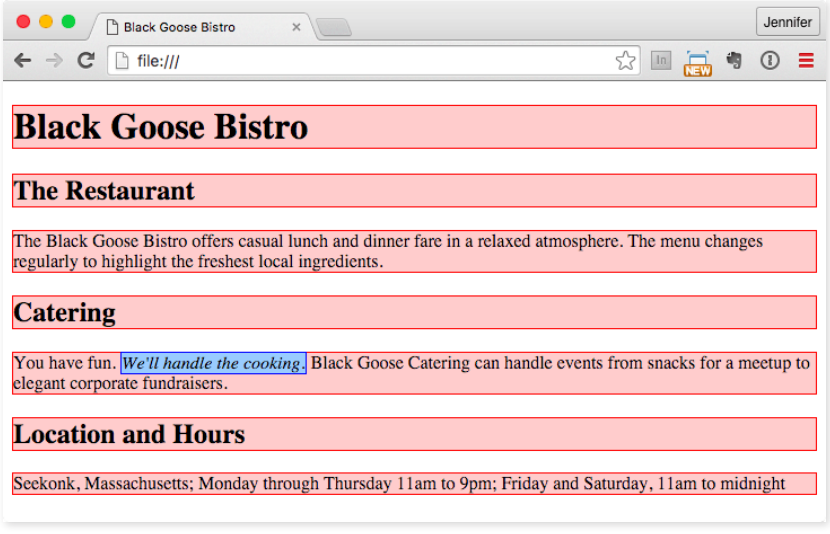


FIGURE 4-11. The outlines show the structure of the elements in the home page.

By contrast, look at the text we marked up as emphasized (`em`, outlined in blue in FIGURE 4-11). 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 (also called a text-level semantic element or phrasing element). Inline elements do not start new lines; they just go with the flow.

Default Styles

The other thing that you will notice about the marked-up page in FIGURES 4-10 and 4-11 is that the browser makes an attempt to give the page some

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.

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 (the `blockquote` element for 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 your own 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. You should always choose elements based on how 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 images? In EXERCISE 4-4, we'll add an image to the page with the `img` element. Images will be discussed in more detail in **Chapter 7, Adding Images**, but for now, they give 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 4-6: a bit of text content surrounded by start and end tags.

A handful of elements, however, do not have 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 an empty 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`, a.k.a. "horizontal rules"), 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 4-12 shows the very simple syntax of an empty element (compare it to FIGURE 4-6).

```
<element-name>
```

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

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

FIGURE 4-12. Empty element structure.

Attributes

Let's get back to adding an image with the empty **img** element. Obviously, an **<img>** tag is not very useful by itself—it doesn't indicate 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.

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">
```

FIGURE 4-13 shows an **img** element with its required attributes labeled.

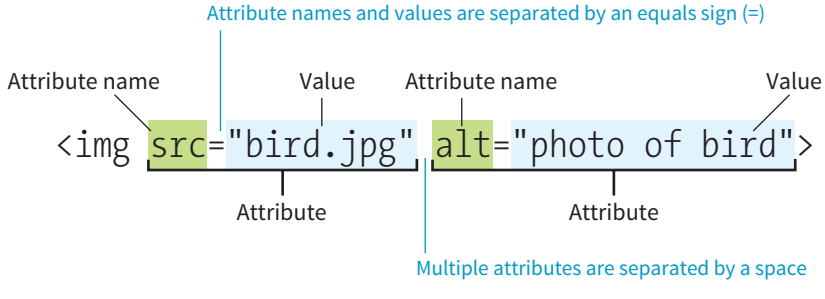


FIGURE 4-13. An **img** element with two attributes.

What Is That Extra Slash?

If you poke around in source documents for existing web pages, you may see empty elements with extra slashes at the end, like so: **<img />**, **
, **<meta />, and **<hr />**. That indicates the document was written according to the stricter rules of XHTML. In XHTML, all elements, including empty elements, must be closed (or terminated, to use the proper term). You terminate empty elements by adding a trailing slash before the closing bracket. The preceding character space is not required but was used for backward compatibility with browsers that did not have XHTML parsers, so **<img/>**, **
**, and so on are valid.

Attributes are instructions that clarify or modify an element.

Here's what you need to know about attributes:

- Attributes go after the element name in the opening tag only, never in the closing 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 are single descriptive words. For example, the **`checked`** attribute, which makes a form checkbox checked when the form loads, is equivalent to **`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.
- Wrapping attribute values in double quotation marks is a strong convention, but note that quotation marks are not required and may be omitted. In addition, either single or double quotation marks are acceptable as long as the opening and closing marks match. Note that quotation marks in HTML files need to be straight ("), not curly (").
- The attribute names and values available for each element are defined in the HTML specifications; in other words, you can't make up an attribute for an element.
- Some attributes are required, such as the **`src`** and **`alt`** attributes in the **`img`** element. The HTML specification also defines which attributes are required in order for the document to be valid.

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 EXERCISE 4-4. 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 (*learningwebdesign.com/5e/materials*). You can also get the image file by saving it right from the sample web page online at *learningwebdesign.com/5e/materials/ch04/bistro*. Right-click (or Control-click on a Mac) the goose image and select "Save to disk" (or similar) from the pop-up menu, as shown in FIGURE 4-14. 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="logo">Black Goose Bistro</h1>
```

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

Mac: Control-click on the image to access the pop-up menu. The options may vary by browser.

FIGURE 4-14. Saving an image file from a page on the web.

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.

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

```
<h1><img src="blackgoose.png" alt="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 4-15.
5. Now save *index.html* and open or refresh it in the browser window. The page should look like the one shown in FIGURE 4-15. 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.

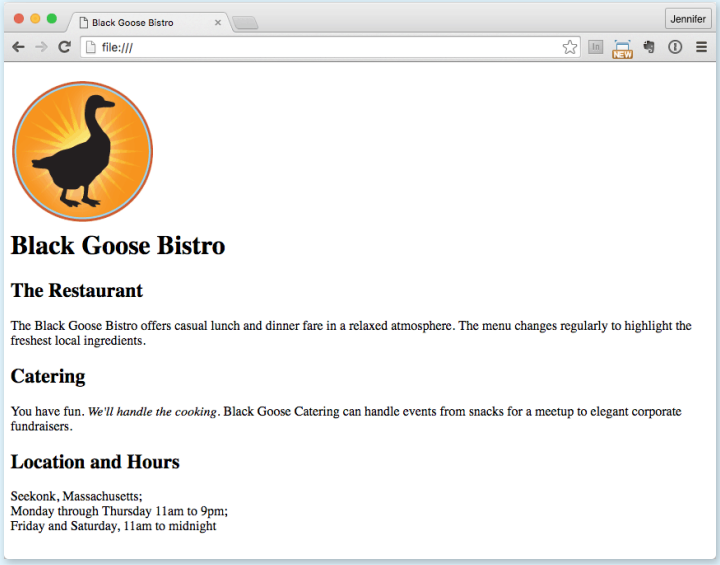


FIGURE 4-15. The Black Goose Bistro page with the logo image.

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 by 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. 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, Introducing Cascading Style Sheets**.
2. The **style** element is placed inside the document **head**. 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. Next, 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's going on (although it's fairly intuitive). You'll learn all about style rules in **Part III**.

```
<style>
body {
  background-color: #faf2e4;
  margin: 0 10%;
  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 4-16. If it doesn't, go over the style sheet to make sure you didn't miss a semicolon or a curly bracket. Look at the way the page looks with our styles compared to the browser's default styles (FIGURE 4-15).

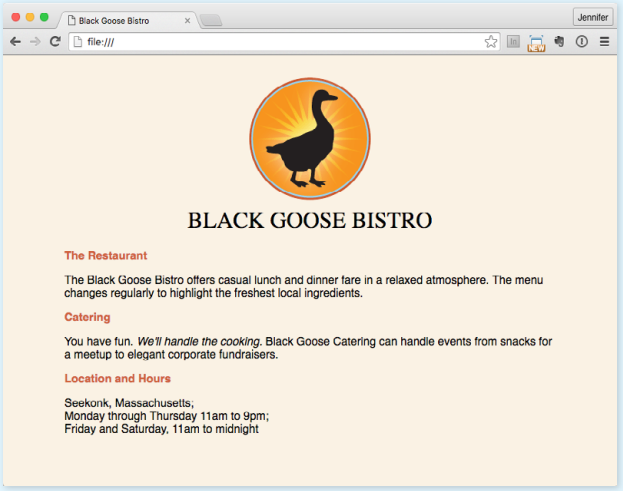


FIGURE 4-16. The Black Goose Bistro page after CSS style rules have been applied.

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 also 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 you're 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.

What if I had neglected to type the slash in the closing emphasis tag (****)? With just one character out of place (FIGURE 4-17), 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 (see **Note**).

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

g.

Catering

You have fun. *We'll handle the cooking. Black Goose Catering can handle events from snacks for a meetup to elegant corporate fundraisers.*

Location and Hours

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

FIGURE 4-17. When a slash is omitted, the browser doesn't know when the element ends, as is the case in this example.

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 4-18).

See how the headline is missing? That's because without the closing tag bracket, the browser assumes that all the following text—all the way up to the next closing bracket (**>**) it finds—is part of the **<h2>** opening tag. 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.

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.

Having Problems?

The following are some typical problems that crop up when you are 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 you're writing HTML by hand.

I put in a graphic by 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 (PNG, JPEG, GIF, or SVG) and that it is named with the proper suffix (*.png*, *.jpeg* or *.jpg*, *.gif*, or *.svg*, respectively).

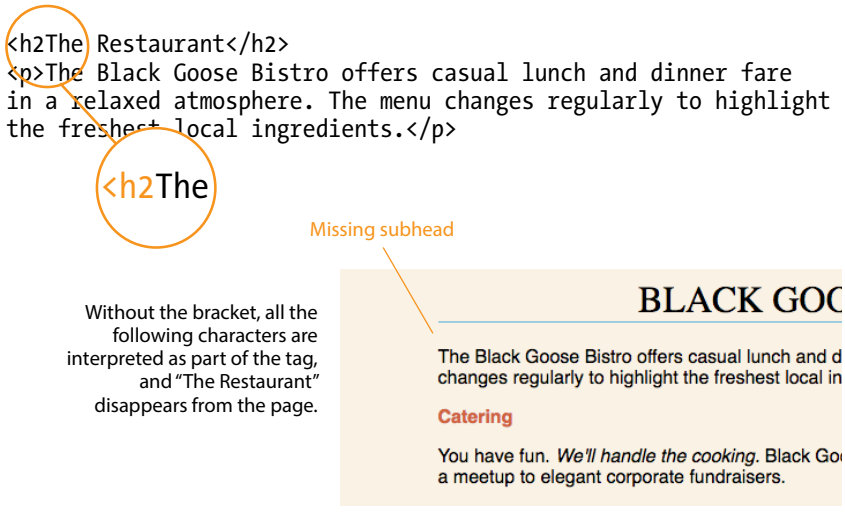


FIGURE 4-18. A missing end bracket makes the browser think the following characters are part of the tag, and therefore the headline text doesn't display.

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 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.

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. 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 anytime you stray from the standard, you introduce unpredictability in the way the page is handled by browsers or 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 expected to memorize every minute rule in the specifications. Instead, 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 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 (incorrectly putting elements inside other elements).
- Typos and other minor errors.

Developers use a number of helpful tools for checking and correcting errors in HTML documents. The best web-based validator is at *html5.validator.nu*. There you can upload a file or provide a link to a page that is already online. FIGURE 4-19 shows the report the validator generates when I upload the version of the Bistro *index.html* file that doesn't have any markup. For this document, there are a number of missing elements that keep this document from being valid. It also shows the problem source code and provides an explanation of how the code should appear. Pretty darned handy!

Built-in browser developer tools for Safari and Chrome also have validators so you can check your work on the fly. Some code editors have validators built in as well.

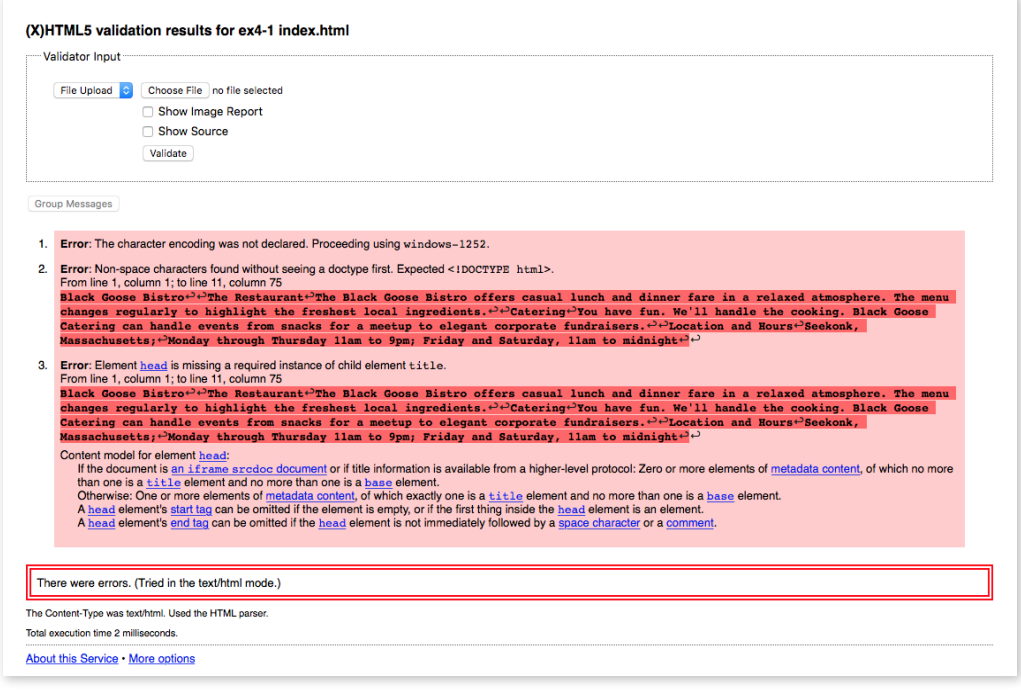


FIGURE 4-19. The (X)HTML5 Validator (Living Validator) for checking errors in HTML documents (*html5.validator.nu*).

ELEMENT REVIEW: HTML DOCUMENT SETUP

This chapter introduced the elements that establish metadata and content portions of an HTML 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 itself
`html`	Is the root element that contains all the other elements
`meta`	Provides information about the document
`title`	Gives the page a title

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 markup for an HTML5 document.

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

   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. `<em>Congratulations!<em>`

   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
   ```

CHAPTER 5

MARKING UP TEXT

Once your content is ready to go (you've proofread it, right?) and you've added the markup to structure the document (`<!DOCTYPE>`, `html`, `head`, `title`, `meta charset`, 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. 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.

IN THIS CHAPTER

PARAGRAPHS

`<p>…</p>`

Paragraph element

Paragraphs are the most rudimentary elements of a text document. Indicate a paragraph with the `p` element by inserting an opening `<p>` tag at the beginning of the paragraph and a closing `</p>` 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>
```

NOTE

I will be teaching markup according to the HTML5 standard maintained by the W3C (www.w3.org/TR/html5/). As of this writing, the latest version is the HTML 5.2 Proposed Recommendation (www.w3.org/TR/html52/).

```
<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>
```

No Naked Text!

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.

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), but they may *not* contain headings, lists, sectioning elements, or any elements that typically display as blocks by default.

Technically, it is OK to omit the closing **</p>** tag because it is not required in order for the document to be valid. A browser just assumes it is closed when it encounters the next block element. Many web developers, including myself, prefer to close paragraphs and all elements for the sake of consistency and clarity. I recommend folks who are just learning markup 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, creating a logical document structure and outline.

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 display 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.

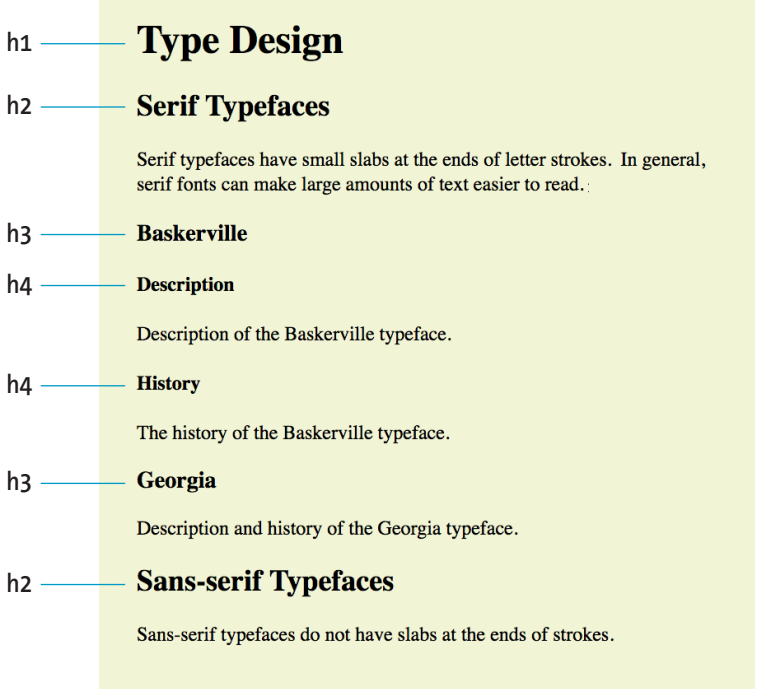


FIGURE 5-1. The default rendering of four heading levels.

THEMATIC BREAKS (HORIZONTAL RULE)

`<hr>`
A horizontal rule

If you want to indicate that one topic has completed and another one is beginning, you can insert what the spec calls a "paragraph-level thematic break" with the **hr** element. The **hr** element adds a logical divider between sections of a page or paragraphs without introducing a new heading level.

In older HTML versions, **hr** was defined as a "horizontal rule" because it inserts a horizontal line on the page. Browsers still render **hr** as a 3-D shaded rule and put it on a line by itself with some space above and below by default; but in the HTML5 spec, it has a new semantic name and definition. 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:

```
<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>
```

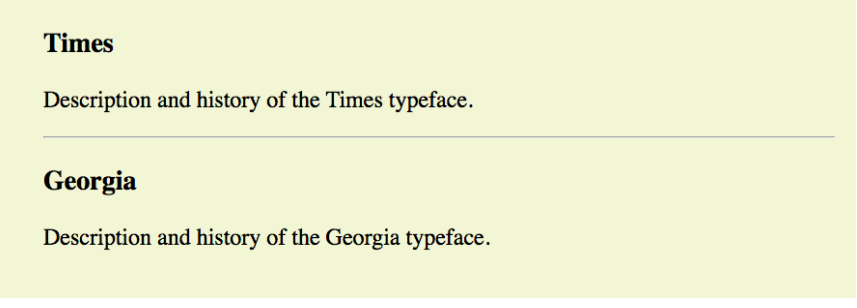


FIGURE 5-2. The default rendering of a thematic break (horizontal rule).

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 qualifies as an unordered list. 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, to mark up each item in the list as a list item (**li**), enclose it in opening and closing **li** tags, as shown in this example. Notice that there are no bullets in the source document. The browser adds them automatically (FIGURE 5-3).

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 content element within a list item (**li**):

```
<ul>
  <li>Serif</li>
  <li>Sans-serif</li>
  <li>Script</li>
  <li>Display</li>
  <li>Dingbats</li>
</ul>
```

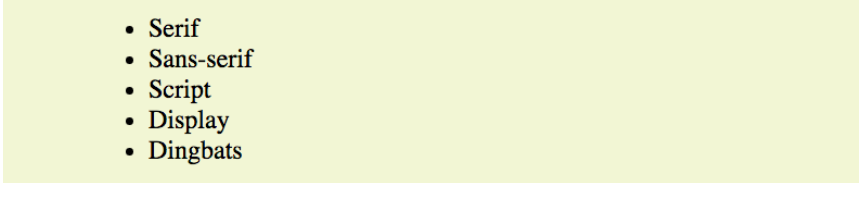


FIGURE 5-3. The default rendering of the sample unordered list. The browser adds the bullets automatically.

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, and even made them look like graphical buttons. The markup stays exactly the same.

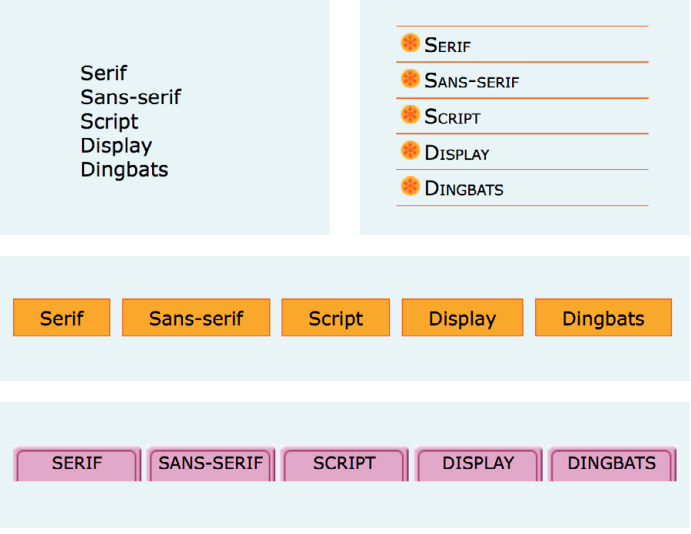


FIGURE 5-4. With style sheets, you can give the same unordered list many looks.

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, but they are defined with the **ol** element (for "ordered list," of course). Instead of bullets, the browser automatically inserts numbers before ordered list items (see **Note**), so you don't need to number them in the source document. This makes it easy to rearrange list items without renumbering them.

NOTE

If something is logically an ordered list, but you don't want numbers to display, remember that you can always remove the numbering with style sheets. So go ahead and mark up the list semantically as an **ol** *and adjust how it displays with a style rule.*

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

```
<ol>
  <li>Gutenberg 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. Gutenberg 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 browser adds the numbers automatically.

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.

Description Lists

`<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

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.

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>
```

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 item of an ordered list:

```
<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 with CSS rules.

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 12, Formatting Text**.

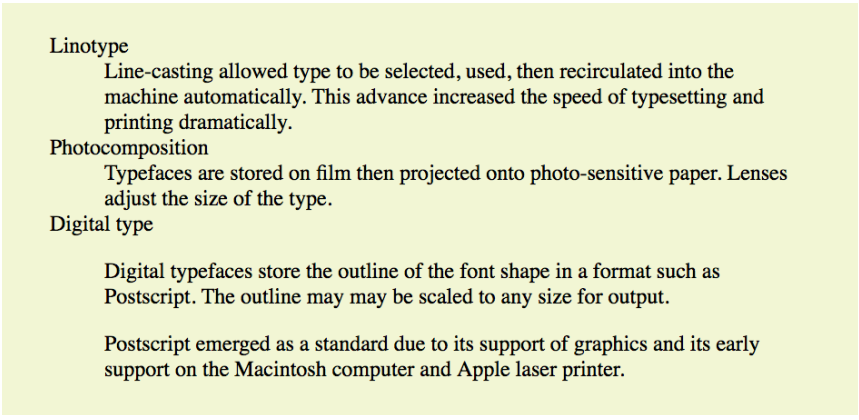


FIGURE 5-6. The default rendering of a definition list. Definitions are set off from the terms by an indent.

The **dl** element is allowed to contain only **dt** and **dd** elements. You cannot put headings or content-grouping elements (like paragraphs) in names (**dt**), but the value (**dd**) can contain any type of flow content. For example, the last **dd** element in the previous example contains two paragraph elements (the awkward default spacing could be cleaned up with a style sheet).

It is permitted to have multiple definitions with one term and vice versa. Here, each term-description group has one term and multiple definitions:

```
<dl>
  <dt>Serif examples</dt>
  <dd>Baskerville</dd>
  <dd>Goudy</dd>

  <dt>Sans-serif examples</dt>
  <dd>Helvetica</dd>
  <dd>Futura</dd>
  <dd>Avenir</dd>
</dl>
```

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, **main**, 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. The one exception is the newer **main** element, which is not recognized by any version of Internet Explorer (although it is supported in the Edge browser); see the sidebar **"HTML5 Support in Internet Explorer,"** later in this chapter, for a workaround.

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, 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:

```
<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>
```

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

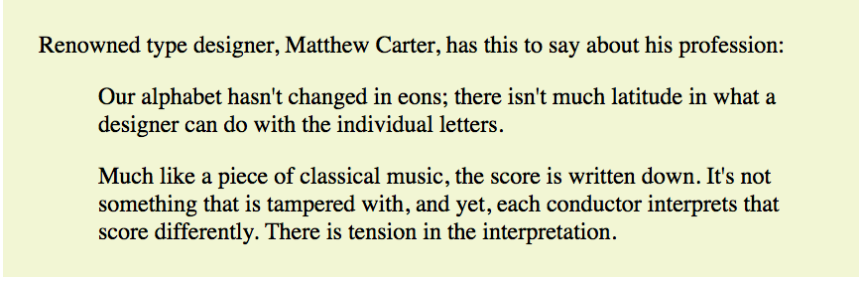


FIGURE 5-7. The default rendering of a **blockquote** element.

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.*

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 certain poems, the whitespace is important for conveying meaning. For content in which whitespace is semantically significant, use 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; however, you can easily change the font with a style sheet rule.

NOTE

The **white-space:pre** *CSS property can also be used to preserve spaces and returns in the source.*

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>
```

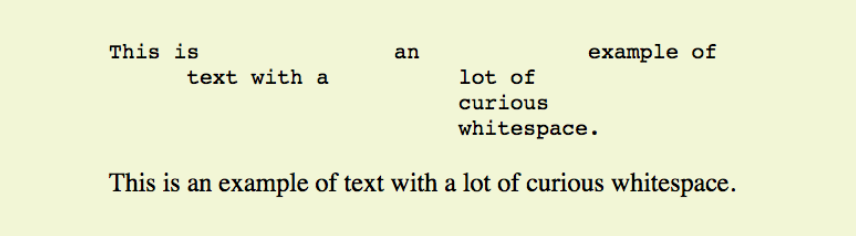


FIGURE 5-8. Preformatted text is unique in that the browser displays the whitespace exactly as it is typed into the source document. Compare it to the paragraph element, in which multiple line returns and character spaces are reduced to a single space.

Figures

<figure>…</figure>
Related image or resource

<figcaption>…</figcaption>
Text description of a figure

The **figure** element identifies 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. Content in a **figure** element 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 you can simply add an image to a page, wrapping it in **figure** tags makes its purpose explicitly clear semantically. It also works as a hook for applying 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>
```

If you want to provide a text caption for the figure, use the **figcaption** element above or below the content inside the **figure** element. It is a more semantically rich way to mark up the caption than using a simple **p** element.

```
<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.

BROWSER SUPPORT NOTE

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).*

EXERCISE 5-1. Marking up a recipe

The owners of the Black Goose Bistro have decided to share recipes and news on their site. In the exercises in this chapter, we'll assist them with content markup.

In this exercise, 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, as well as the final version with markup, is available at *learningwebdesign.com/5e/materials*.

```
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 (see **Note**), including main content (**main**), headers (**header**), footers (**footer**), sections (**section**), articles (**article**), navigation (**nav**), and tangentially related or complementary content (**aside**). Curiously, the spec lists the old **address** element as a section as well, so we'll look at that one here too.

NOTE

The new element names are based on a Google study that looked at the top 20 names that developers assigned to generic division elements (code.google.com/webstats/2005-12/classes.html).

HTML5 Support in Internet Explorer

Nearly all browsers today support the 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 you need to make them behave correctly:

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

Unfortunately, that fix won't work for the small fraction of users who are still using Internet Explorer versions 8 and earlier (less than 1.5% of browser traffic as of 2017). IE8 has been hanging around well past its prime because it is tied to the popular Windows Vista operating system. If you work on a large site for which 1% of users represents thousands of people, you may want to be familiar with workarounds and fallbacks for IE8. Most likely, you won't need to support it. Still, at the risk of looking outdated, I will provide notes about IE8 support throughout this book.

For example, the following is a workaround that applies only to IE8 and earlier. Not only do those browsers not recognize the HTML5 elements, but 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:

```
documencreateElement("section");
```

Fortunately, Remy Sharp wrote a script that creates all of the HTML5 elements for IE8 and earlier in one fell swoop. It is called "HTML5 Shiv" (or Shim) and it is available on a server that you can point to in your documents. Just 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="//cdnjs.cloudflare.com/ajax/libs/html5shiv/3.7.3/ →
html5shiv.min.js">
</script >
<![endif]-->
```

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 Modernizr online at *modernizr.com*. It is also covered in **Chapter 20, Modern Web Development Tools**.

Main Content

`<main>…</main>`
Primary content area of page or app

Web pages these days are loaded with different types of content: mastheads, sidebars, ads, footers, more ads, even more ads, and so on. It is helpful to cut to the chase and explicitly point out the main content on the page. Use the **`main`** element to identify the primary content of a page or application. It helps screen readers and other assistive technologies know where the main content of the page begins and replaces the "Skip to main content" links that have been utilized in the past. The content of a **`main`** element should be unique to that page. In other words, headers, sidebars, and other elements that appear across multiple pages in a site should not be included in the **`main`** section:

```
<body>
<header>…</header>
<main>
  <h1>Humanist Sans Serif</h1>
  <!-- code continues -->
</main>
</body>
```

The W3C HTML5 specification states that pages should have only one **`main`** section and that it should not be nested within an **`article`**, **`aside`**, **`header`**, **`footer`**, or **`nav`**. Doing so will cause the document to be invalid.

The **`main`** element is the most recent addition to the roster of HTML5 grouping elements. You can use it and style it in most browsers, but for Internet Explorer (including version 11, the most current as of this writing), you'll need to create the element with JavaScript and set its display to **`block`** with a style sheet, as discussed in the **"HTML5 Support in Internet Explorer"** sidebar. Note that **`main`** is supported in MS Edge.

Headers and Footers

`<header>…</header>`
Introductory material for page, section, or article

`<footer>…</footer>`
Footer for page, section, or article

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.

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 (we'll get to those elements next). 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:

```
<body>
<header>
  <img src="/images/logo.png" alt="logo">
  <h1>Nuts about Web Fonts</h1>
```

```
    <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-->
  </body>
```

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="2017-11-11">November 11,
2017</time></p>
  </header>
  <!-- article content here -->
</article>
```

NOTE

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 appear last in the document or sectioning element. It could also appear at or near the beginning if that makes sense.

In this simple example, we see the typical information listed at the bottom of an article marked up as a **footer**:

```
<article>
  <header>
    <h1>More about WOFF</h1>
    <p>by Jennifer Robbins, <time datetime="2017-11-11">November 11,
2017</time></p>
  </header>
  <!-- article content here -->
  <footer>
    <p><small>Copyright &copy;2017 Jennifer Robbins.</small></p>
```

NOTE

The **a href="/">** *code in the examples is the markup for adding links to other web pages. We'll take on links in* **Chapter 6, Adding Links**. *Normally the value would be the URL to the page, but I've used a simple slash as a space-saving measure.*

NOTE

The **time** *element will be discussed in the section* **"Dates and times"** *later in this chapter.*

```
      <nav>
        <ul>
          <li><a href="/">Previous</a></li>
          <li><a href="/">Next</a></li>
        </ul>
      </nav>
    </footer>
  </article>
```

Sections and Articles

`<section>…</section>`
Thematic group of content

`<article>…</article>`
Self-contained, reusable composition

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 include a heading (inside the **`section`** element) plus 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 "yes" to the question "Could this appear on another site and make sense?"

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>
```

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.*

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

Conversely, a **section** in a web document might be composed 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**.

Aside (Sidebars)

`<aside>…</aside>`
Tangentially related material

The **aside** element identifies content that is separate from, but tangentially related to, the surrounding content. In print, its equivalent is a sidebar, but it couldn't be called "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://fonts.google.com">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

The **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 **nav** 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.

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:

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

Document Outlines

Behind the scenes, browsers look at the markup in a document and generate a hierarchical outline based on the headings in the content. A new section gets added to the outline whenever the browser encounters a new heading level.

In past versions of HTML, that was the only way the outline was created. HTML5 introduced a new outline algorithm that enables authors to explicitly add a new section to the outline by inserting a sectioning element: **article**, **section**, **aside**, and **nav**. In addition to the four sectioning elements, the spec defines some elements (**blockquote**, **fieldset**, **figure**, **dialog**, **details**, and **td**) as sectioning roots, which means headings in those elements do not become part of the overall document outline.

It's a nice idea because it allows content to be repurposed and merged without breaking the outline, but unfortunately, no browsers to date have implemented it and they are unlikely to do so. The W3C has kept the sectioning elements and their intended behavior in the spec (which is why I mention this at all), but now precede it with a banner recommending sticking with the old hierarchical heading method.

THE INLINE ELEMENT ROUNDUP

Now that we've identified the larger chunks of content, we can provide semantic meaning to phrases within the chunks by using what the HTML5 specification 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. TABLE 5-1 lists all of them.

Although it may be handy seeing all of the text-level elements listed together in a table, they certainly deserve more detailed explanations.

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

The first sentence indicates *who* is very smart. The second example is about *how* smart he is. Notice that the **em** element has an effect on the meaning of the sentence.

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 is important, serious, or urgent. In the following example, the **strong** element identifies the portion of instructions that requires extra attention. The **strong** element does not change the meaning of the sentence; it merely draws attention to the important parts:

```
<p>When returning the car, <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

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	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	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) or alternate language
ins	Inserted text; indicates an insertion in a document
kbd	Keyboard; text entered by a user (for technical documents)
mark	Contextually relevant text
q	Short, inline quotation
ruby, rt, rp	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	Machine-readable time data
u	Indicates a formal name, misspelled word, or text that would be 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 because of 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").

Many inline elements have the expected style rendering (bold for the **b** element, for example). Other inline elements are purely semantic (such as **abbr** or **time**) and don't have default renderings. For any inline elements, you can use CSS rules if you want to change the way they display.

Obsolete HTML 4.01 Text Elements

Here are some old text elements that were made obsolete in HTML5: **acronym**, **applet**, **basefont**, **big**, **center**, **dir** (directory), **font**, **isindex** (search box), **menu**, **strike**, **tt** (teletype). I mention them here in case you run across them in an old document when viewing its source or if you are using an older web authoring tool. There is no reason to use them today.

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.

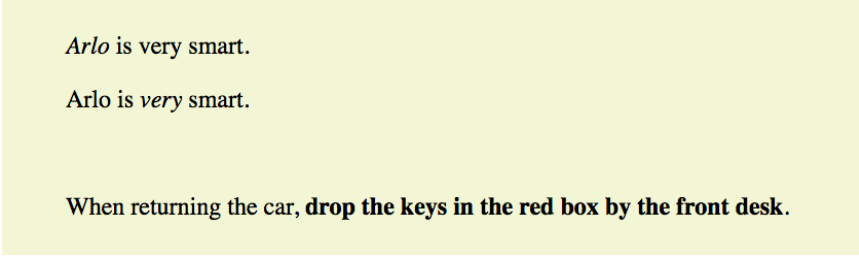


FIGURE 5-9. The default rendering of emphasized and strong text.

Elements originally named for their presentational properties

`<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)

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, strike-through, 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

Keywords, product names, and other phrases that need to stand out from the surrounding text without conveying added importance or emphasis (see **Note**). [*Old definition:* Bold]

CSS Property: 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

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 a thought. [*Old definition:* Italic]

CSS Property: 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

Indicates text that is incorrect. [*Old definition:* Strike-through text]

CSS Property: To draw a line through a selection of text, use **`text-decoration`**. Example: **`text-decoration: line-through`**

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

u

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, such as the misspelled "Helvitica" in the following example. Note that underlined text is easily confused with a link and should generally be avoided except for a few niche cases. [*Old definition:* Underline]

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

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

small

Indicates an addendum or side note to the main text, such as the legal "small print" at the bottom of a document. [*Old definition:* Renders in font smaller than the surrounding text]

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

Example: `<p><small>(This font is free for personal and commercial use.)</small></p>`

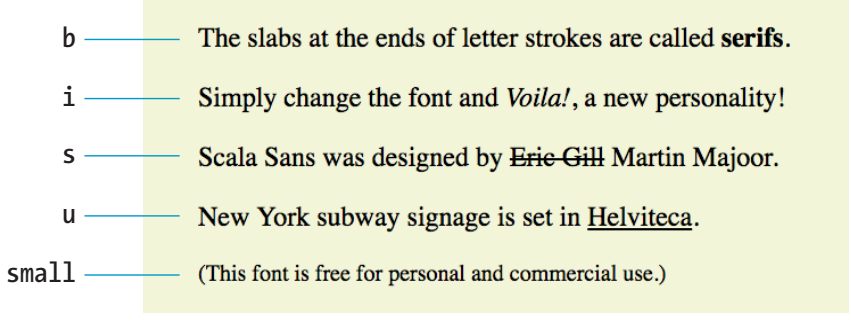


FIGURE 5-10. The default rendering of **`b`**, **`i`**, **`s`**, **`u`**, and **`small`** elements.

Short quotations

`<q>…</q>`
Short inline quotation

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>
```

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. Some browsers, like Firefox, render curly quotes, which is preferable. Others (Safari and Chrome, which I used for my examples) render them as straight quotes as shown in the figure.

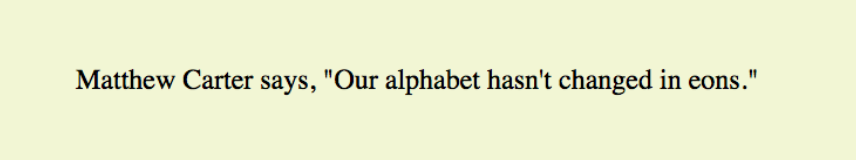


FIGURE 5-11. Browsers add quotation marks automatically around **q** elements.

Abbreviations and acronyms

`<abbr>…</abbr>`
Abbreviation or acronym

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 NASA 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>
```

NOTE

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

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>
```

Here is an example of elements that are nested incorrectly. Notice that the inner **i** element is not closed within the containing **q** element:

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

It is easy to spot the nesting error in an example that is this short, but when you're nesting long passages or nesting multiple levels deep, it is easy to end up with overlaps. One advantage to using an HTML code editor is that it can automatically close elements for you correctly or point out when you've made a mistake.

Citations

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>
```

`<cite>…</cite>`
Citation

Defining terms

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>
```

`<dfn>…</dfn>`
Defining term

Program code elements

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.

`<code>…</code>`
Code

`<var>…</var>`
Variable

`<samp>…</samp>`
Program sample

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

Subscript and 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>
```

`<sub>…</sub>`
Subscript

`<sup>…</sup>`
Superscript

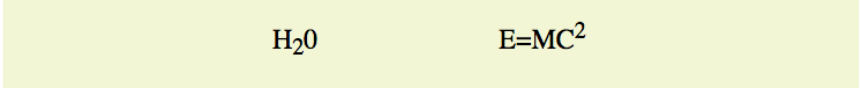


FIGURE 5-12. Subscript and superscript

Highlighted text

`<mark>…</mark>`
Contextually relevant text

The **mark** element indicates a word that may be considered especially relevant to the reader. One might use it to dynamically highlight a search term in a page of results, to manually call attention to a passage of text, or to indicate the current page in a series. Some designers (and browsers) give marked text a light colored background as though it were 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. In this example, search terms are identified with **mark** elements and given a yellow background with a style sheet so they are easier for the reader to find.

Dates and times

`<time>…</time>`
Time data

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. The full time format begins with the date (year–month–day). The time section begins with a letter "T" and lists hours (on the 24-hour clock), 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. When identifying dates and times alone, you can omit the other sections.

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."*

■ FURTHER READING

For more information on the intricate ins and outs of specifying dates and times, with examples, check out the **time** element entry in the HTML5 specification: *www.w3.org/TR/2014/REC-html5-20141028/text-level-semantics.html#the-time-element*.

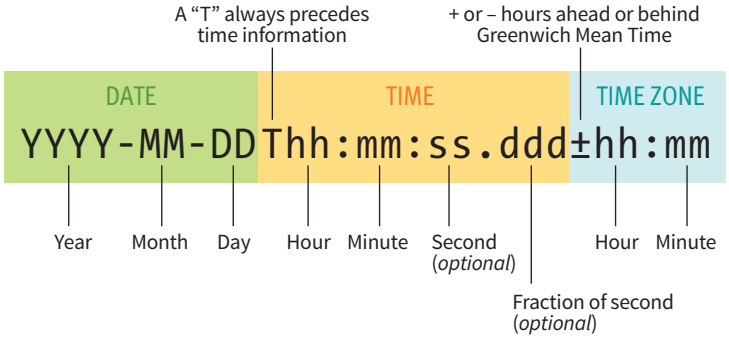


FIGURE 5-14. *Standardized date and time syntax.*

Here are a few examples of valid values for **datetime**:

- **Time only:** 9:30 p.m.

```
<time datetime="21:30">9:30p.m.</time>
```

- **Date only:** June 19, 2016

```
<time datetime="2016-06-19">June 19, 2016</time>
```

- **Date and time:** Sept. 5, 1970, 1:11a.m.

```
<time datetime="1970-09-05T01:11:00">Sept. 5, 1970, 1:11a.m.</time>
```

- **Date and time, with time zone information:** 8:00am on July 19, 2015, in Providence, RI

```
<time datetime="2015-07-19T08:00:00-05:00">July 19, 2015, 8am,
Providence RI</time>
```

NOTE

You can also use the **time** *element without the* **datetime** *attribute, but its content must be a valid date/time string:*

```
<time>2016-06-19</time>
```

Machine-readable information

The **data** element is another tool for helping computers make sense of content. It can be used for all sorts of data, including dates, times, measurements, weights, microdata, and so on. The required **value** attribute provides the machine-readable information. Here are a couple of examples:

```
<data value="12">Twelve</data>
<data value="978-1-449-39319-9">CSS: The Definitive Guide</data>
```

`<data>…</data>`
Machine-readable data

I'm not going to go into more detail on the **data** element, because as a beginner, you are unlikely to be dealing with machine-readable data quite yet. But 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
Longstocking</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 the flow of text where you want a break to occur, as shown here and in FIGURE 5-15:

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

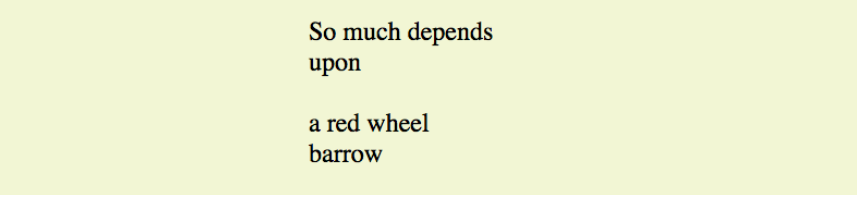


FIGURE 5-15. Line breaks are inserted at each **br** element. (Example extracted from "The Red Wheelbarrow" by William Carlos Williams.)

Unfortunately, the **br** element is easily abused. 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>
```

Word breaks

`<wbr>`
Word break

The word break (**wbr**) element lets you mark the place where a word should break (a "line break opportunity" according to the spec) should there not be enough room for the whole word (FIGURE 5-16). It takes some of the guesswork away from the browser and allows authors to control the best spot for the word to be split over two lines. If there is enough room, the word stays in one piece. Without word breaks, the word stays together, and if there is not enough room, the whole word wraps to the next line. Note that the browser does not add a hyphen when the word breaks over two lines. The **wbr** behaves as though it were a character space in the middle of the word:

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

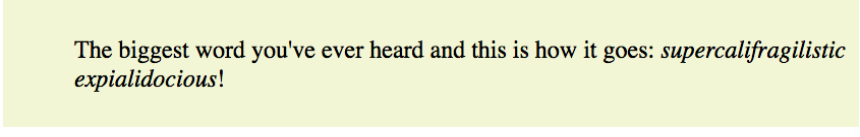


FIGURE 5-16. When there is not enough room for a word to fit on a line, it will break at the location of the **wbr** element.

BROWSER SUPPORT NOTE

*The **wbr** element is not supported by any version of Internet Explorer as of this writing. It is supported in MS Edge.*

You've been introduced to 32 new elements since your last exercise. I'd say it's time to give some of the inline elements a try in EXERCISE 5-2.

Accommodating Non-Western Languages

If the web is to reach a truly worldwide audience, it needs to be able to support the display of all the languages of the world, with all their unique alphabets, symbols, directionality, and specialized punctuation. The W3C's efforts for internationalization (often referred to as "i18n" —an i, then 18 letters, then an n) ensure that the formats and protocols defined in web technologies are usable worldwide.

Internationalization efforts include the following:

- Using the Unicode character encoding that contains the characters, glyph, symbols, ideographs, and the like from all active, modern languages. Unicode is discussed in **Chapter 4, Creating a Simple Page**.
- Declaring the primary language of a document by using a two-letter language code from the ISO 639-1 standard (available at *www.loc.gov/standards/iso639-2/php/code_list.php*). For example, English is "EN," Czech is "CS, "and German is "DE." Use the **lang** attribute in the **html** element to declare the language for the whole document, or in individual elements that require clarification.
- Accommodating the various writing directions of languages. In HTML, the **dir** attribute explicitly sets the direction for the document or an element to **ltr** (left-to-right) or **rtl** (right-to-left). On phrase-level elements, it also creates a bidirectional isolation, preventing text within the element from influencing the ordering of text outside it. (This can be an important consideration when you are embedding user-generated text.)

  For example, to include a passage of Hebrew in an English document, use the **dir** attribute to indicate that the phrase should be displayed right-to-left:

  ```
  <p>This is how you write Shalom:
  <span dir="rtl">שלום</span></p>
  ```

- Providing a system that allows for ruby annotation, notes that typically appear above ideographs from East Asian languages to give pronunciation clues or translations (**ruby**, **rt**, and **rp** elements). See the spec for details if this is something you need to do.

The W3C Internationalization Activity site provides a thorough collection of HTML and CSS authoring techniques and resources to help with your internationalization efforts: *www.w3.org/International/techniques/authoring-html*.

EXERCISE 5-2. Identifying inline elements

This little post for the Black Goose Bistro News page 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 my final markup, but there is an opportunity to use all of the preceding elements in the article. For extra credit, there is a phrase that could have two elements applied to it. (Hint: look for a term in another language.) Remember to nest them properly by closing the inner element before you close the outer one. Also, be sure that all text-level elements are contained *within* block elements.

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 *learningwebdesign.com/5e/materials* along with the resulting code.

```
<article>

<header>
<p>posted by BGB, November 15, 2016</p>
</header>

<h2>Low and Slow</h2>
<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)

** … **
Generic block-level element

…
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** elements, as well as the **id** and **class** attributes, to learn how authors use them to structure content.

Divide It Up with a div

Use the **div** element to create a logical grouping of content or elements on the page. It indicates that they belong together in a 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** 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.

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="images/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 also allows me to style **p** elements within listings differently than other **p** elements in the document.

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 I have a custom element that I've given the name "news." 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 often now that we have better semantic sectioning elements in HTML5.

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.

Define a Phrase 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 **span**s are inline elements, they may contain only 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 **span**s 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 also 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 upcoming sidebar **"Structured Data in a Nutshell"**).

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' ISBNs (International Standard Book Numbers) to uniquely identify each listing. No two book listings may share the same **id**.

```
<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 "links" or "news" in the document:

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

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

id and class Values

In HTML5, the values for **id** and **class** attributes must contain one character (that is, they may not be empty) and may not contain any character spaces. You can use pretty much any character in the value.

Earlier versions of HTML had restrictions on **id** values (for example, they needed to start with a letter), but those restrictions were removed in HTML5.

Classification with class

The **class** attribute classifies elements into conceptual groups; therefore, unlike the **id** attribute, a **class** name may be shared by multiple elements. 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 classify 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>
```

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>
```

Identify and Classify All Elements

The **id** and **class** attributes are not limited to just **div** and **span**—they are two of the global attributes (see the **"Global Attributes"** sidebar) in HTML,

MARKUP TIP

Use the **id** attribute to *identify*.

Use the **class** attribute to *classify*.

Global Attributes

HTML5 defines a set of attributes that can be used with every HTML element. They are called the global attributes:

- accesskey
- class
- contenteditable
- dir
- draggable
- hidden
- id
- lang
- spellcheck
- style
- tabindex
- title
- translate

Appendix B lists all of the global attributes, their values, and definitions.

which means you may use them with all HTML elements. 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>
```

This should have given you a good introduction to how to use the **class** and **id** attributes to add meaning and organization to documents. We'll work with them even more in the style sheet chapters in **Part III**. The sidebar **"Structured Data in a Nutshell"** discusses more advanced ways of adding meaning and machine-readable data to documents.

IMPROVING ACCESSIBILITY WITH ARIA

As web designers, we must always consider the experience of users with assistive technologies for navigating pages and interacting with web applications. Your users may be listening to the content on the page read aloud by a screen reader and using keyboards, joysticks, voice commands, or other non-mouse input devices to navigate through the page.

Many HTML elements are plainly understood when you look at (or read) only the HTML source. Elements like the title, headings, lists, images, and tables have implicit meanings in the context of a page, but generic elements like **div** and **span** lack the semantics necessary to be interpreted by an assistive device. In rich web applications, especially those that rely heavily on JavaScript and AJAX (see **Note**), the markup alone does not provide enough clues as to how elements are being used or whether a form control is currently selected, required, or in some other state.

NOTE

AJAX (Asynchronous JavaScript and XML) is explained in a sidebar in **Chapter 22, Using JavaScript**.

Fortunately, we have ARIA (Accessible Rich Internet Applications), a standardized set of attributes for making pages easier to navigate and interactive features easier to use. The specification was created and is maintained by a Working Group of the Web Accessibility Initiative (WAI), which is why you also hear it referred to as WAI-ARIA. ARIA defines roles, states, and properties that developers can add to markup and scripts to provide richer semantic information.

Roles

Roles describe an element's function or purpose in the context of the document. Some roles include **alert**, **button**, **dialog**, **slider**, and **menubar**, to name only a few. For example, as we saw earlier, you can turn an unordered list into a tabbed menu of options using style sheets, but what if you can't *see* that it is styled that way? Adding **role="toolbar"** to the list makes its purpose clear:

Structured Data in a Nutshell

It is pretty easy for us humans to tell the difference between a recipe and a movie review. For search engines and other computer programs, however, it's not so obvious. When we use HTML alone, all browsers see is paragraphs, headings, and other semantic elements of a document. Enter structured data! Structured data allows content to be machine-readable as well, which helps search engines provide smarter, user-friendly results and can provide a better user experience—for example, by extracting event information from a page and adding it to the user's calendar app.

There are several standards for structured data, but they share a similar approach. First, they identify and name the "thing" being presented. Then they point out the properties of that thing. The "thing" might be a person, an event, a product, a movie…pretty much anything you can imagine seeing on a web page. Properties consist of name/value pairs. For example, "actor," "director," and "duration" are properties of a movie. The values of those properties appear as the content of an HTML element. A collection of the standardized terms assigned to "things," as well as their respective properties, form what is called a vocabulary.

The most popular standards for adding structured data are Microformats, Microdata, RDFa (and RDFa Lite), and JSON-LD. They differ in the syntax they use to add information about objects and their properties.

Microformats

microformats.org

This early effort to make web content more useful created standardized values for the existing **`id`**, **`class`**, and **`rel`** HTML attributes. It is not a documented standard, but it is a convention that is in widespread use because it is very simple to implement. There are about a dozen stable Microformat vocabularies for defining people, organizations, events, products, and more. Here is a short example of how a person might be marked up using Microformats:

```
<div class="h-card">
  <p class="p-name">Cindy Sherman</p>
  <p class="p-tel">555.999-2456</p>
</div>
```

Microdata

html.spec.whatwg.org/multipage/microdata.html

Microdata is a WHATWG (Web Hypertext Application Technology Working Group) HTML standard that uses microdata-specific attributes (**`itemscope`**, **`itemtype`**, **`itemprop`**, **`itemid`**, and **`itemref`**) to define objects and their properties. Here is an example of a person defined using Microdata.

```
<div itemscope itemtype="http://schema.org/Person">
  <p itemprop="name">Cindy Sherman</p>
  <p itemprop="telephone">555.999-2456</p>
</div>
```

For more information on the WHATWG, see **Appendix D, From HTML+ to HTML5**.

RDFa and RDFa Lite

www.w3.org/TR/xhtml-rdfa-primer/

The W3C dropped Microdata from the HTML5 spec in 2013, putting all of its structured data efforts behind RDFa (Resource Description Framework in Attributes) and its simplified subset, RDFa Lite. It uses specified attributes (**`vocab`**, **`typeof`**, **`property`**, **`resource`**, and **`prefix`**) to enhance HTML content. Here is that same person marked up with RDFa:

```
<div vocab="http://schema.org" typeof="Person">
  <p property="name">Cindy Sherman</p>
  <p property="telephone">555.999-2456</p>
</div>
```

JSON-LD

json-ld.org

JSON-LD (JavaScript Object Notation to serialize Linked Data) is a different animal in that it puts the object types and their properties in a script removed from the HTML markup. Here is the JSON-LD version of the same person:

```
<script type="application/ld+json">
{
  "@context": "http://schema.org/",
  "@type": "Person",
  "name": "Cindy Sherman"
  "telephone": "555.999-2456"
}
</script>
```

It is possible to make up your own vocabulary for use on your sites, but it is more powerful to use a standardized vocabulary. The big search engines have created Schema.org, a mega-vocabulary that includes standardized properties for hundreds of "things" like blog posts, movies, books, products, reviews, people, organizations, and so on. Schema.org vocabularies may be used with Microdata, RDFa, and JSON-LD (Microformats maintain their own separate vocabularies). You can see pointers to the Schema.org "Person" vocabulary in the preceding examples. For more information, the Schema.org "Getting Started" page provides an easy-to-read introduction: *schema.org/docs/gs.html*.

There is a lot more to say about structured data than I can fit in this book, but once you get the basic semantics of HTML down, it is definitely a topic worthy of further exploration.

```
<ul id="tabs" role="toolbar">
  <li>A-G</li>
  <li>H-O</li>
  <li>P-T</li>
  <li>U-Z</li>
</ul>
```

Here's another example that reveals that the "status" **div** is used as an alert message:

```
<div id="status" role="alert">You are no longer connected to the
server.</div>
```

Some roles describe "landmarks" that help readers find their way through the document, such as **navigation**, **banner**, **contentinfo**, **complementary**, and **main**. You may notice that some of these sound similar to the page-structuring elements that were added in HTML5, and that's no coincidence. One of the benefits of having improved semantic section elements is that they can be used as landmarks, replacing ** ** with **main**.

Most current browsers already recognize the implicit roles of the new elements, but some developers explicitly add ARIA roles until all browsers comply. The sectioning elements pair with the ARIA landmark roles in the following way:

```
<nav role="navigation">
<header role="banner"> (see Note)
<main role="main">
<aside role="complementary">
<footer role="contentinfo">
```

NOTE

The banner role is used when the **header** *applies to only the whole page, not just a section or article.*

States and Properties

ARIA also defines a long list of states and properties that apply to interactive elements such as form widgets and dynamic content. States and properties are indicated with attributes prefixed with **aria-**, such as **aria-disabled**, **aria-describedby**, and many more.

The difference between a state and property is subtle. For properties, the value of the attribute is more likely to be stable, such as **aria-labelledby**, which associates labels with their respective form controls, or **aria-haspopup**, which indicates the element has a related pop-up menu. States have values that are more likely to be changed as the user interacts with the element, such as **aria-selected**.

For Further Reading

Obviously, this is not enough ARIA coaching to allow you to start confidently using it today, but it should give you a good feel for how it works and

its potential value. When you are ready to dig in and take your skills to a professional level, here is some recommended reading:

The WAI-ARIA Working Draft *(www.w3.org/TR/wai-aria-1.1/)*

This is the current Working Draft of the specification as of this writing.

ARIA in HTML *(www.w3.org/TR/html-aria/)*

This W3C Working Draft helps developers use ARIA attributes with HTML correctly. It features a great list of every HTML element, whether it has an implicit role (in which ARIA should *not* be used), and what roles, states, and properties apply.

ARIA Resources at MDN Web Docs *(developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA)*

This site features lots of links to ARIA-related and up-to-date resources. It is a good starting point for exploration.

HTML5 Accessibility *(www.html5accessibility.com)*

This site tests which new HTML5 features are accessibly supported by major browsers.

SPEC TIP

The W3C HTML specification now lists which ARIA roles and properties apply in the descriptions of every HTML element (*www.w3.org/TR/html52/*).

CHARACTER ESCAPES

There's just one more text-related topic before we close out this chapter. The section title makes it sound like someone left the gate open and all the characters got out. The real meaning is more mundane, albeit useful to know.

You already know that as a browser parses an HTML document, when it runs into a **<** symbol, it interprets it as the beginning of a tag. But what if you just need a less-than symbol in your text? Characters that might be misinterpreted as code need to 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 entity 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 (escaping) a specific character:

- Using a predefined abbreviated name for the character (called a named entity; see **Note**).
- Using an assigned numeric value that corresponds to its position in a coded character set (numeric entity). Numeric values may be in decimal or hexadecimal format.

All character references begin with an **&** (ampersand) and end with a **;** (semicolon).

NOTE

HTML defines hundreds of named entities as part of the markup language, which is to say you can't make up your own entity.

An example should make this clear. I'd like to use a less-than symbol in my text, so I must use the named entity (**<**) or its numeric equivalent (**<**) where I want the symbol to appear (FIGURE 5-17):

```
<p>3 tsp. &lt; 3 Tsp.</p>
```

or:

```
<p>3 tsp. &#060; 3 Tsp.</p>
```

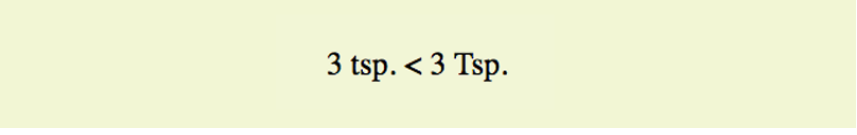


FIGURE 5-17. The special character is substituted for the character reference when the document is displayed in the browser.

When to Escape Characters

There are a few instances in which you may need or want to use a character reference.

HTML syntax characters

■ MARKUP TIP

These character entities are useful when you need to show an example of HTML markup on a web page.

The <, >, &, ", and ' characters have special syntax meaning in HTML, and may be misinterpreted as code. Therefore, the W3C recommends that you escape <, >, and & characters in content. If attribute values contain single or double quotes, escaping the quote characters in the values is advised. Quote marks are fine in the content and do not need to be escaped. (See TABLE 5-2.)

TABLE 5-2. *Syntax characters and their character references*

Character	Description	Entity name	Decimal no.	Hexadecimal no.
<	Less-than symbol	<	<	<
>	Greater-than symbol	>	>	>
"	Quotation mark	"	"	"
'	Apostrophe	'	'	'
&	Ampersand	&	&	&

Invisible or ambiguous characters

Some characters have no graphic display and are difficult to see in the markup (TABLE 5-3). These include the non-breaking space (** **), which is used 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. Another use for non-breaking spaces is to separate digits in a long number, such as 32 000 000.

Zero-width space can be placed in languages that do not use spaces between words to indicate where the line should break. A zero-width joiner is a non-printing space that causes neighboring characters to display in their connected forms (common in Arabic and Indic languages). Zero-width non-joiners prevent neighboring characters from joining to form ligatures or other connected forms.

TABLE 5-3. *Invisible characters and their character references*

Character	Description	Entity name	Decimal no.	Hexadecimal no.
(non-printing)	Non-breaking space	` `	` `	` `
(non-printing)	En space	` `	` `	` `
(non-printing)	Em space	` `	` `	` `
(non-printing)	Zero-width space	(none)	`&#8203;`	`&#x200B;`
(non-printing)	Zero-width non-joiner	`&zwnj;`	`&#8204;`	`&#x200C;`
(non-printing)	Zero-width joiner	`&zwj;`	`&#8205;`	`&#x200D;`

Input limitations

If your keyboard or editing software does not include the character you need (or if you simply can't find it), you can use a character entity to make sure you get the character you want. The W3C doesn't endorse this practice, so use the proper character in your source if you are able. TABLE 5-4 lists some special characters that may be less straightforward to type into the source.

TABLE 5-4. *Special characters and their character references*

Character	Description	Entity name	Decimal no.	Hexadecimal no.
‘	Left curly single quote	`&lsquo;`	`&#8216;`	`&#x2018;`
’	Right curly single quote	`&rsquo;`	`&#8217;`	`&#x2019;`
“	Left curly double quote	`&ldquo;`	`&#8220;`	`&#x201C;`
”	Right curly double quote	`&rdquo;`	`&#8221;`	`&#x201D;`
…	Horizontal ellipsis	`&hellip;`	`&#8230;`	`&#x2026;`
©	Copyright	`&copy;`	`&#169;`	`&#xA9;`
®	Registered trademark	`&reg;`	`&#174;`	`&#xAE;`
™	Trademark	`&trade;`	`&#8482;`	`&#x2026;`
£	Pound	`&pound;`	`&#163;`	`&#xA3;`
¥	Yen	`&yen;`	`&#165;`	`&#xA5;`
€	Euro	`&euro;`	`&#8364;`	`&#x20AC;`
–	En dash	`&ndash;`	`&#8211;`	`&#x2013;`
—	Em dash	`&mdash;`	`&#8212;`	`&#x2014;`

A complete list of HTML named entities and their Unicode code-points can be found as part of the HTML5 specification at *www.w3.org/TR/html5/syntax.html#named-character-references*. For a more user-friendly listing of named and numerical entities, I recommend this archived page at the Web Standards Project: *www.webstandards.org/learn/reference/charts/entities*.

PUTTING IT ALL TOGETHER

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, grouping (block) elements, phrasing (inline) elements, sectioning elements, and character entities. Have fun!

EXERCISE 5-3. The Black Goose Bistro News page

Now that you've been introduced to all of the text elements, you can put them to work by marking up the News page for the Black Goose Bistro site. Get the starter text and finished markup files at *learningwebdesign.com/5e/materials*. Once you have the text, follow the instructions listed after it. The resulting page is shown in FIGURE 5-18.

```
The Black Goose Bistro News

Home
Menu
News
Contact

Summer Menu Items
posted by BGB, June 18, 2017
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, 2016
<p>This week I am <em>extremely</em> excited about a new cooking
technique called <dfn><i>sous vide</i></dfn>. In <i>sous vide</i>
cooking, you submerge the food (usually vacuum-sealed in plastic) into a
water bath that is precisely set to the target temperature you want the
food to be cooked to. In his book, <cite>Cooking for Geeks</cite>, Jeff
Potter describes it as <q>ultra-low-temperature poaching.</q></p>

<p>Next month, we will be serving <b><i>Sous Vide</i> Salmon with Dill
Hollandaise</b>. To reserve a seat at the chef table, contact us before
<time datetime="20161130">November 30</time>.</p>

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

All content copyright 2017, Black Goose Bistro and Jennifer Robbins
```

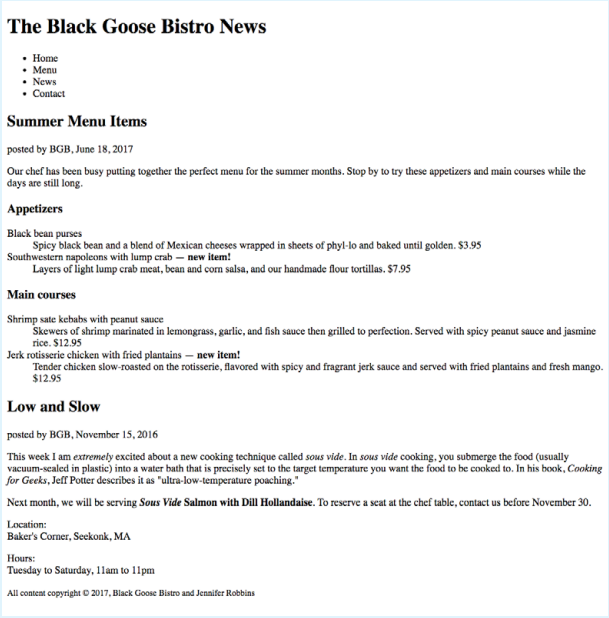
MARKUP TIP

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

NOTE

The "Low and Slow" paragraph is already marked up with the inline elements from EXERCISE 5-2).

EXERCISE 5-3. Continued



The Black Goose Bistro News

- Home
- Menu
- News
- Contact

Summer Menu Items

posted by BGB, June 18, 2017

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 phyl-lo 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, 2016

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."

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.

Location:
Baker's Corner, Seekonk, MA

Hours:
Tuesday to Saturday, 11am to 11pm

FIGURE 5-18. The finished menu page.

1. Start by adding the DOCTYPE declaration to tell browsers this is an HTML5 document.
2. Add all the document structure elements first (**`html`**, **`head`**, **`meta`**, **`title`**, and **`body`**). Give the document the title "The Black Goose Bistro News."
3. 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`**).
4. The News page has two posts titled "Summer Menu Items" and "Low and Slow." Mark up each one as an article.
5. 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 in EXERCISE 5-2.
6. The content after the header is 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 final 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."
7. 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.
8. Now we can add a few fine details. *Classify* each price as "price" using **`span`** elements.
9. Two of the dishes are new items. Change the double hyphens to an em dash character and mark up "new item!" as "strongly important." Classify the title of each new dish as "newitem" (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.
10. 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 date information.
11. 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.
12. 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. Mark up the hours with the **`time`** element (you don't need the date or time zone portions).
13. 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" using the keyboard or the **`&copy;`** character entity.

Save the as *bistro_news.html*, and check your page in a modern browser. You can also upload it to *validator.nu* and make sure it is valid (it's a great way to spot mistakes). 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. Find the answers in **Appendix A**.

1. Add the markup to insert 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** element?

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

6. What element would you use to mark up "W3C" and provide its full name (World Wide Web Consortium)? Can you write out the complete markup?

7. What is the difference between **dl** and **dt**?

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

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

Want More Practice?

Try marking up your own résumé. Start with the raw text and add document structure elements, content grouping elements, and 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 ELEMENTS

The global attributes apply to all text elements. Additional attributes are listed under their respective elements.

Page sections

`address`	Author contact information
`article`	Self-contained content
`aside`	Tangential content (sidebar)
`footer`	Related content
`header`	Introductory content
`nav`	Primary navigation
`section`	Conceptually related group of content

Heading content

`h1...h6`	Headings, levels 1 through 6

Grouping content elements and attributes

`blockquote`	Blockquote
`cite`	The URL of the cited content
`div`	Generic division
`figure`	Related image or resource
`figcaption`	Text description of a figure
`hr`	Paragraph-level thematic break (horizontal rule)
`main`	Primary content area of page or app
`p`	Paragraph
`pre`	Preformatted text

List elements and attributes

`dd`	Definition
`dl`	Definition list
`dt`	Term
`li`	List item (for **`ul`** and **`ol`**)
`value`	Provides a number for an **`li`** in an **`ol`**
`ol`	Ordered list
`reversed`	Numbers the list in reverse order
`start`	Provides the starting number for the list
`ul`	Unordered list

Breaks

`br`	Line break
`wbr`	Word break

Phrasing elements and attributes

`abbr`	Abbreviation
`b`	Added visual attention (bold)
`bdi`	Bidirectional isolation
`bdo`	Bidirectional override
`cite`	Citation
`code`	Code sample
`data`	Machine-readable equivalent
`del`	Deleted text
`cite`	The URL of cited content.
`datetime`	Specifies the date and time of a change
`dfn`	Defining term
`em`	Stress emphasis
`i`	Alternate voice (italic)
`ins`	Inserted text
`cite`	The URL of cited content
`datetime`	Specifies the date and time of a change
`kbd`	Keyboard input
`mark`	Highlighted text
`q`	Short inline quotation
`cite`	The URL of the cited content
`ruby`	Section containing ruby text
`rp`	Parentheses in ruby text
`rt`	Ruby annotation
`s`	Strike-through; incorrect text
`samp`	Sample output
`small`	Annotation; "small print"
`span`	Generic phrase of text
`strong`	Strong importance
`sub`	Subscript
`sup`	Superscript
`time`	Machine-readable time data
`datetime`	Provides machine readable date/time
`pubdate`	Indicates the time refers to publication
`u`	Added attention (underline)

CHAPTER 6

ADDING LINKS

If you're creating a page for the web, chances are you'll want to link 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—links to other sites, to your own site, and within a page. There is one element that makes linking possible: the anchor (**a**).

IN THIS CHAPTER

`<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 site:

```
<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="tarsierlogo.gif"
alt="O'Reilly Media site"></a>
```

By the way, you can put any HTML content element in an anchor to make it a link, not just images.

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**.

When a user clicks or taps 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.

USABILITY TIP

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.

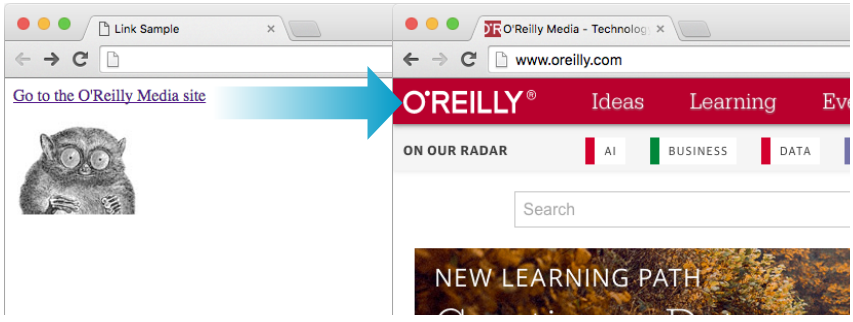


FIGURE 6-1. When a user clicks or taps the linked text or image, the page specified in the anchor element loads in the browser window.

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

Absolute URLs provide the full URL for the document, including the protocol (**`http://`** or **`https://`**), 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):

```
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 long pathname intimidate you.

Relative URLs

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:

```
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.

■ AT A GLANCE

Anchor Structure

The simplified structure of the anchor element is as follows:

```
<a href="url">linked content</a>
```

■ MARKUP TIP

URL Wrangling

If you're linking to a page with a long URL, it's 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.

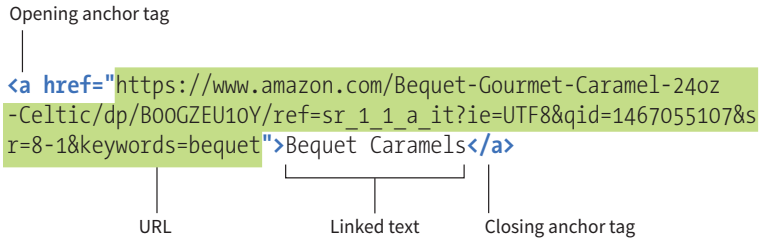


FIGURE 6-2. An example of a long URL. Although it may make the anchor tag look confusing, the structure is the same.

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, 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 *www.foodnetwork.com* site. 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** element 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 *voilà*! Now "The Food Network" appears as a link and takes my visitors to that site when they click or tap it. Give it a try in EXERCISE 6-1.

■ TRY IT

Working Along with Jen's Kitchen

FIGURE 6-3. The Jen's Kitchen page.

All the files for the Jen's Kitchen website are available online at *learningwebdesign.com/5e/materials*. Download the entire directory, making sure not to change the way its contents are organized. The pages aren't much to look at, but they will give you a chance to develop your linking skills.

The resulting markup for all of the exercises is also provided.

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 Food Network link example:

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

When you are done, save *index.html* and open it in a browser. If you have an internet connection, you can click 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

A large portion of the linking you do is 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, (see **Note**) 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. I recommend you do EXERCISES 6-2 through 6-8 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.

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.

Important Pathname Don'ts

When writing relative pathnames, 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.

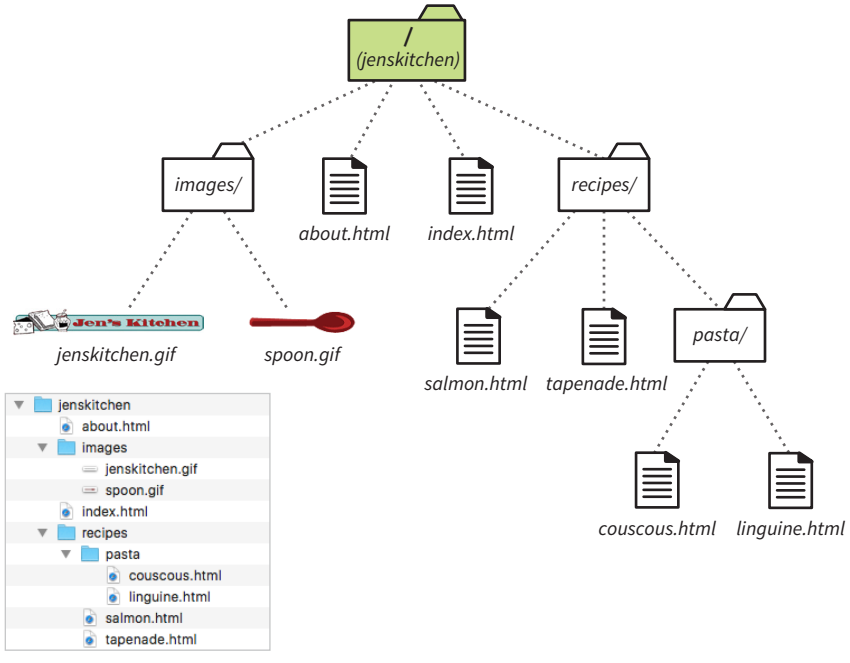


FIGURE 6-4. A diagram of the *jenskitchen* site structure.

Linking Within a Directory

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

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

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

EXERCISE 6-2 gives you a chance to mark up a simple link yourself.

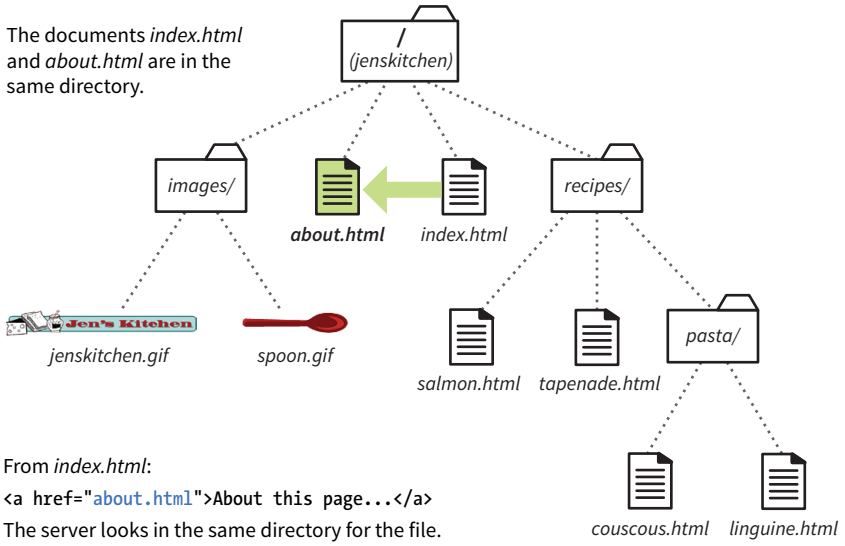


FIGURE 6-5. Writing a relative URL to another document in the same directory.

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, 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 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>
```

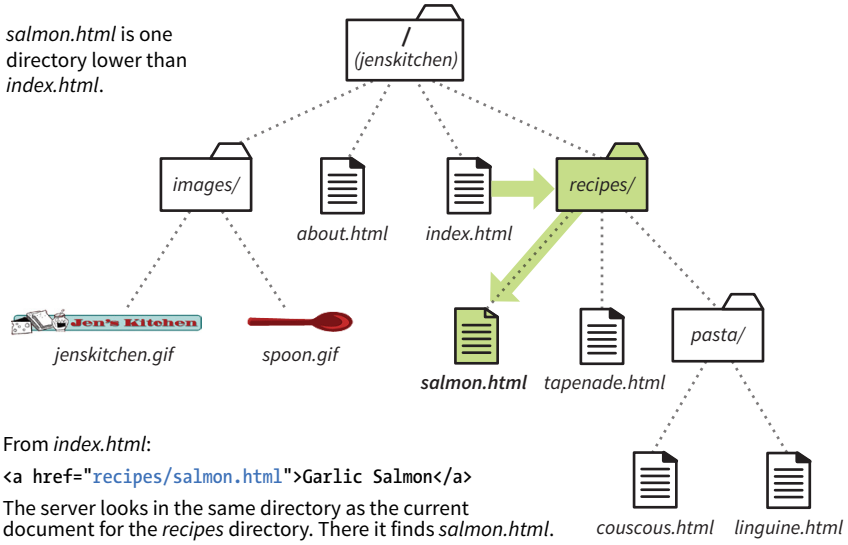


FIGURE 6-6. Writing a relative URL to a document that is one directory level lower than the current document.

Have a try at linking to a file in a directory in EXERCISE 6-3.

EXERCISE 6-3. Link to a file in a directory

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, 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...</a></li>
```

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 a directory called *pasta*, and in there is the file *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 the same location as the current file, and follow each directory name with a slash until you get to the linked filename.

When you link to a file in a lower directory, the pathname contains the names of each subdirectory you go through to get to the file.

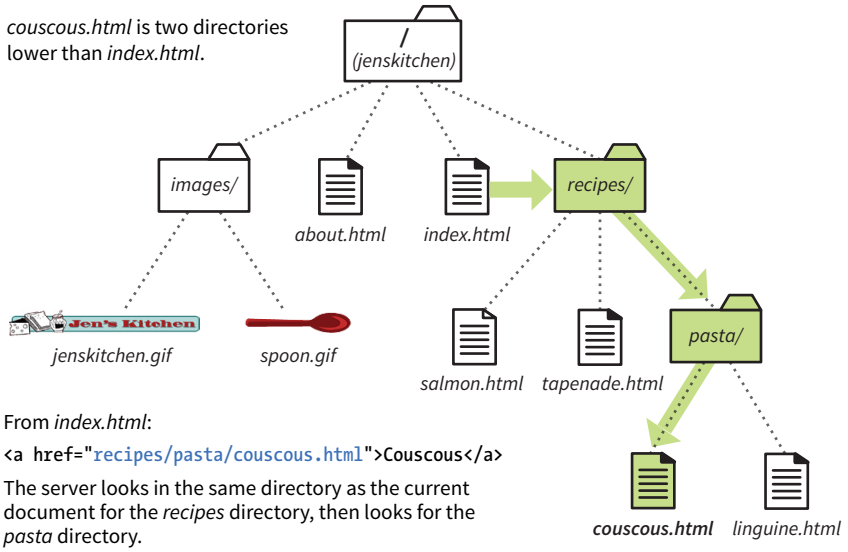


FIGURE 6-7. Writing a relative URL to a document that is two directory levels lower than the current document. You can try it yourself in EXERCISE 6-4.

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, save *index.html* and open it in a browser. Click 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? Now it gets more interesting. 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 macOS.

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 go back up to the *jenskitchen* directory to find *index.html*. This pathname tells the browser to "back 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 the **../** stands in for the name of the higher directory, and we don't need to write out *jenskitchen* in the pathname.

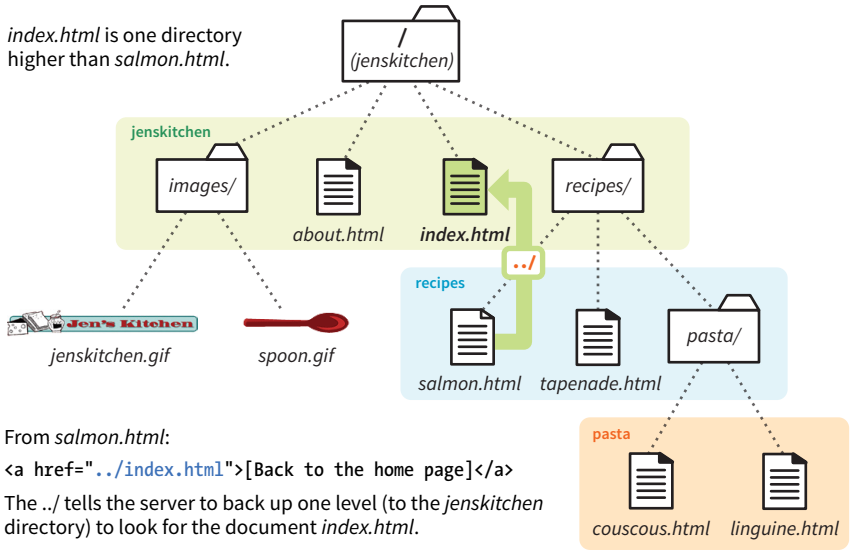


FIGURE 6-8. Writing a relative URL to a document that is one directory level higher than the current document.

Try adding a dot-dot-slash pathname to a higher directory in EXERCISE 6-5.

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 (*jenskitchen*), where *index.html* can be found. Again, there is no need to write out the directory names; the **../** does it all.

Now you try (EXERCISE 6-6).

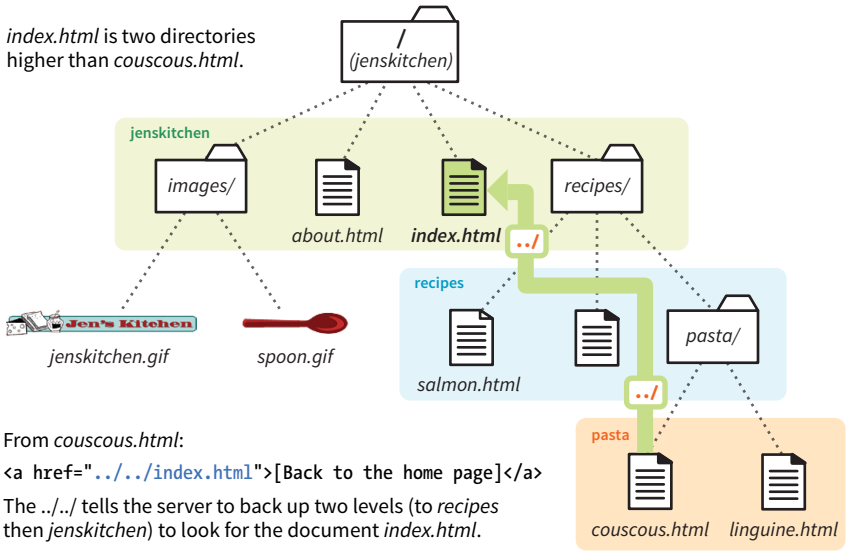


FIGURE 6-9. Writing a relative URL to a document that is two directory levels higher than the current document.

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 6-5. 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.

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 back to the home page by 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.

Linking with Site Root Relative Pathnames

Site root relative links are generally preferred because of their flexibility.

All sites have a root directory, 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 to the file you want to link to. This type of pathname is known as site root relative.

In the Unix pathname convention, a forward slash (/) at the start of the pathname indicates that 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, and then find the *salmon.html* file" (FIGURE 6-10):

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

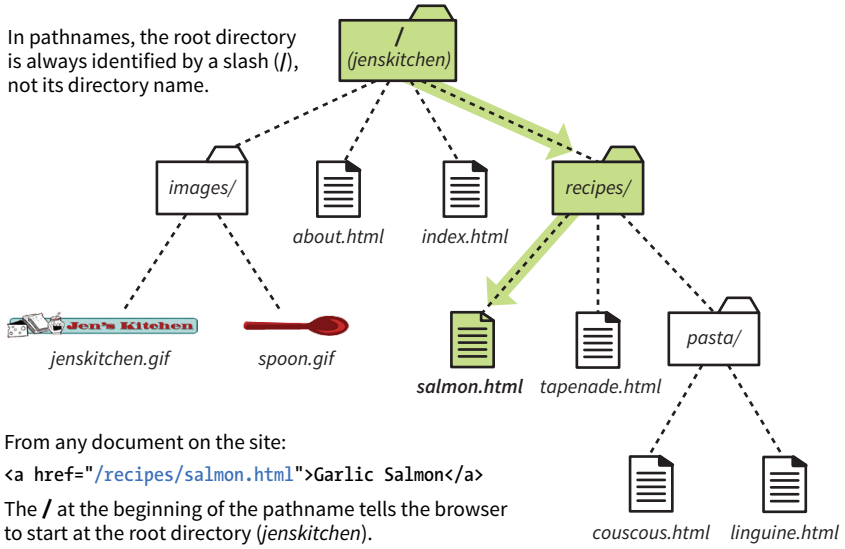


FIGURE 6-10. Writing a relative URL starting at the root directory.

WARNING

Site root relative pathnames won't work on your local computer unless it is set up as a server.

Note that you don't need to (and you shouldn't) write the name of the root directory (*jenskitchen*) in the path—the forward slash (/) at the beginning represents the top-level directory in the pathname. From there, just specify the directories the browser should look in.

Because this type of link starts at the root to describe the pathname, it works 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.

Writing Pathnames to Images

The **src** attribute in the **img** element works the same as the **href** attribute in anchors. Because 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, you'd use the following markup:

```
<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 *pièce 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.

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 write your answers here in the book, or if you want to test your markup to see whether it works, make changes in the actual files. Note that the text shown here isn't included on the exercise pages—you'll need to add it before you can create the link (for example, type in "Go to the Tapenade recipe" for the first question). The final code is in the finished exercise files in the *materials* folder for this chapter. I also included them 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

Most 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 known as linking to a document fragment.

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.

Linking to a particular spot within a page is a two-part process. First, identify the destination, and then 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 the letter H, they'll jump to the "H" heading lower on the page.

Step 1: Identifying the destination

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.

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):

```
<h2 id="startH">H</h2>
```

Step 2: Linking to the destination

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

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

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, pound, or number symbol, before the fragment name:

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

And that's it. Now when someone clicks 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 by using the **id** attribute.

```
<h2 id="startH">H</h2>
<dl>
<dt>hexadecimal</dt>
...
```

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.

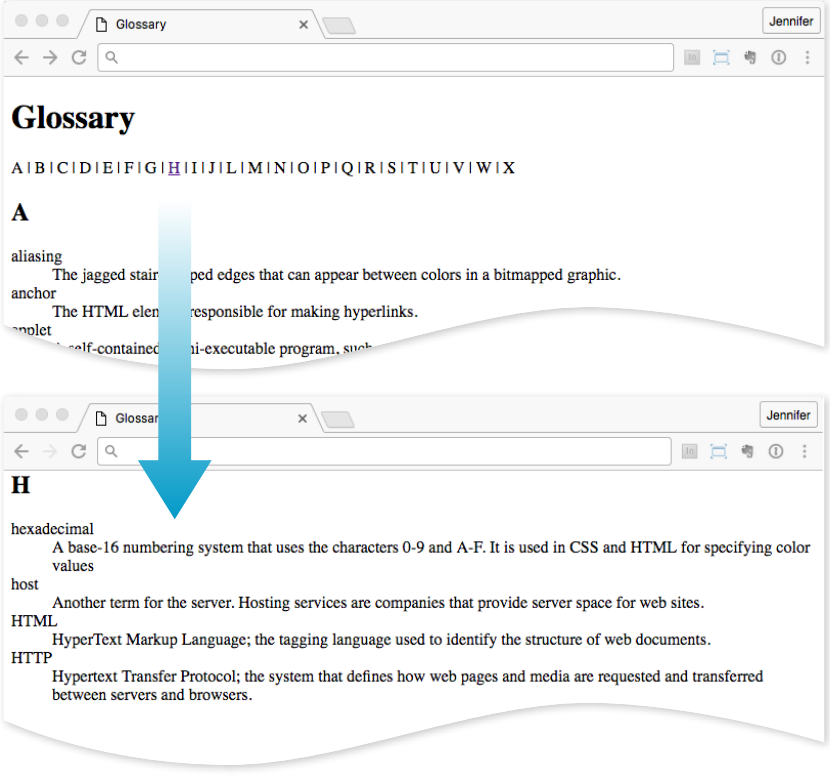


FIGURE 6-11. Linking to a specific destination (a fragment) within a single web page.

USABILITY 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 for users to scroll back after every link.

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>
```

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 their content.*

EXERCISE 6-8. Linking to a fragment

Want some practice linking to specific destinations? Open *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're 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.

Of course, you don't have any control over the named fragments in other people's web pages. 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.

EXERCISE 6-8 gives you an opportunity to add links to fragments in the example glossary page.

TARGETING A NEW BROWSER WINDOW

One problem with putting links on your page is that when people click them, they may never come back to your content. 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.

Be aware that opening new browser windows can cause hiccups in the user experience of your site. Opening new windows is problematic for accessibility, and may be confusing to some users. They might not be able to tell that a new window has opened or they may never find their way back to the original page. At the very least, new windows may be perceived as an annoyance rather than a convenience. So consider carefully whether you need a new window and whether the benefits outweigh the potential drawbacks.

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 (see the "**Pop-up Windows**" sidebar).

Pop-up Windows

It is possible to open a browser window to specific dimensions and with parts of the browser chrome (toolbars, scrollbars, etc.) turned on or off, but you know what…I'm not going to go into that here. First of all, it requires JavaScript. Second, in the era of mobile devices, opening a new browser window at a particular pixel size is an antiquated technique. People often turn off pop-up windows anyway.

For what it's worth, the little interstitial panels you see popping up on every web page asking you to sign up for a mailing list or showing you an ad are done with HTML elements and JavaScript, not a whole new browser window, so that is an entirely different beast.

That said, if you have a legitimate reason for opening a browser window to a specific size, I will refer you to this tutorial by Peter-Paul Koch at Quirksmode: *www.quirksmode.org/js/popup.html*.

To open a new window with 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. The new window may or may not be brought to the front depending on the browser and device used.

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 include **target="_blank"** for every link, every link will launch a new window, potentially leaving your user with a mess of open windows. There's nothing wrong with it, per se, as long as it is not overused.

Another 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.

You can decide which method (a new window for every link or reusing named windows) is most appropriate for your content and interface.

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 a **mailto** link, the browser opens a new mail message preaddressed to that address in a designated mail program (see the **"Spam-Bots"** sidebar).

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 address 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 putting an email address in your document source makes 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 address to display on the page so 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, "you [-at-] example [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 by 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. Using a contact form is a good alternative (web forms are coming up in **Chapter 9, Forms**).

TELEPHONE LINKS

Keep in mind that the smartphones people are using to access your site 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:** protocol and is very simple:

```
<a href="tel:+01-800-555-1212">Call us free at (800) 555-1212</a>
```

When mobile users tap the link, what happens depends on the device: Android launches the phone app; BlackBerry and IE11 Mobile initiate the call immediately; and iOS launches a dialog box giving the option to call, message, or add the number to Contacts. Desktop browsers may launch a dialog box to switch apps (for example, to FaceTime on Safari) or they may ignore the link.

If you don't want any interruption on desktop browsers, 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. This will also prevent them from overriding any styles you've applied to telephone links.

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

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.

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

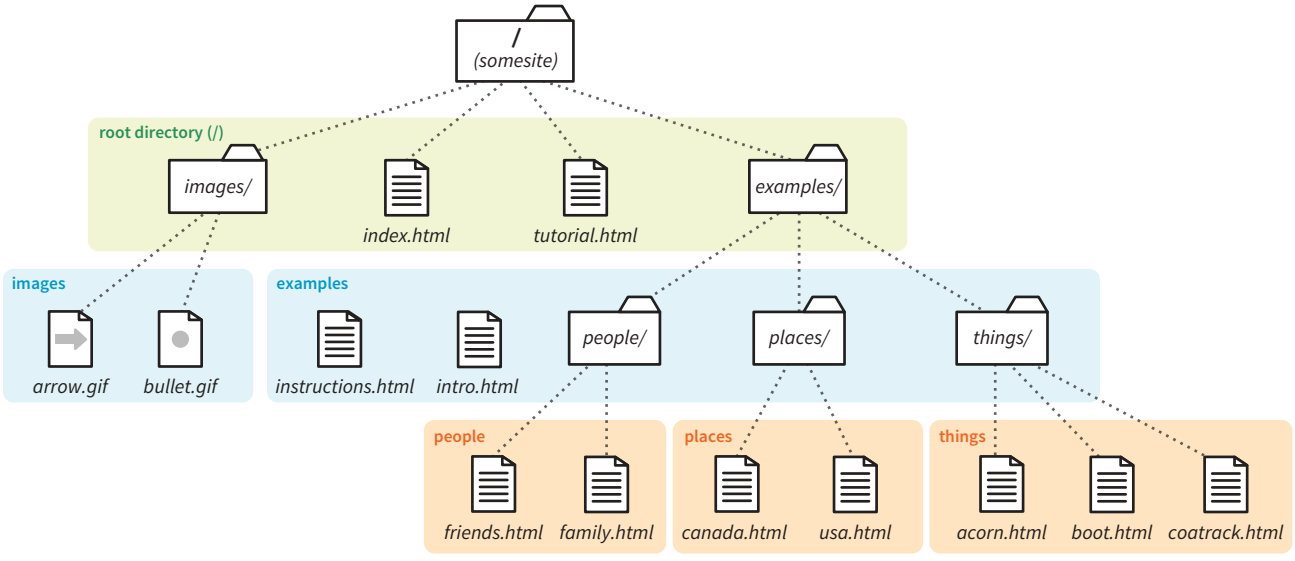


FIGURE 6-12. The directory structure for the "Test Yourself" questions.

documents are just dummy files and contain no content. I filled in the first one for you as an example. The answers are located in **Appendix A**.

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

```
<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 *boot.html* from the *family.html* page, but this time, start with the root directory.

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

MARKUP 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.

6. Create a link to the website for this book (*learningwebdesign.com*) in the file *intro.html*.

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

8. Create a link back to the home page (*index.html*) from *acorn.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*, use this URL:

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

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

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

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

```
<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
`target="`*`text string`*`"`	Targets a browser window by name

CHAPTER 7

ADDING IMAGES

IN THIS CHAPTER

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. If the image is part of the editorial content, such as product shots, gallery images, ads, illustrations, and so on, then it should be placed in the flow of the HTML document. If the image is purely decorative, such as a stunning image in the background of the header or a patterned border around an element, then it should be added through Cascading Style Sheets. Not only does it make sense to put images that affect presentation in a style sheet, but it makes the document cleaner and more accessible and makes the design much easier to update later. I will talk about CSS background images at length in **Chapter 13, Colors and Backgrounds**.

This chapter focuses on embedding image content into the flow of the document, and it is divided into three parts. First, we'll look at the tried-and-true `img` element for adding basic images to a page the way we've been doing it since 1992. It has worked just fine for over 25 years, and as a beginner, you'll find it meets most of your needs as well.

The second part of this chapter introduces some of the methods available for embedding SVG images (Scalable Vector Graphics) in HTML documents. SVGs are a special case and demand special attention.

Finally, we'll look at the way image markup has had to adapt to the wide variety of mobile devices with an introduction to new responsive image elements (`picture` and `source`) and attributes (`srcset` and `sizes`). As the number of types of devices used to view the web began to skyrocket, we realized that a single image may not meet the needs of all viewing environments, from palm-sized screens on slow cellular networks to high-density cinema displays. We needed a way to make images "responsive"—that is, to serve images

appropriate for their browsing environments. After a few years of back and forth between the W3C and the development community, responsive image features were added to the HTML 5.1 specification and are beginning to see widespread browser support.

I want to point out up front that responsive image markup is not as straightforward as the examples we've seen so far in this book. It's based on more advanced web development concepts, and the syntax may be tricky for someone just getting started writing HTML (heck, it's a challenge for seasoned professionals!). I've included it in this chapter because it is relevant to adding inline images, but frankly, I wouldn't blame you if you'd like to skip the "**Responsive Image Markup"** section and come back to it after we've done more work with Responsive Web Design and you have more HTML and CSS experience under your belt.

FIRST, A WORD ON IMAGE FORMATS

We'll get to the **`img`** element and other markup examples in a moment, but first it's important to know that you can't put just any image on a web page; it needs to be in one of the web-supported formats.

In general, images that are made up of a grid of colored pixels (called bitmapped or raster images, as shown in FIGURE 7-1, top) must be saved in the PNG, JPEG, or GIF file formats in order to be placed inline in the content. Newer, more optimized WebP and JPEG-XR bitmapped image formats are slowly gaining in popularity, particularly now that we have markup to make them available to browsers that support them.

For vector images (FIGURE 7-1, bottom), such as the kind of icons and illustrations you create with drawing tools such as Adobe Illustrator, we have the SVG format. There is so much to say about SVGs and their features that I've given them their own chapter (**Chapter 25, SVG**), but we'll look at how to add them to HTML documents later in this chapter.

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), 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>
```

You should name your image files with the proper suffixes—*.png*, *.jpg* (or *.jpeg*), *.gif*, *.webp*, and *.jxr*, respectively. In addition, your server must be configured to recognize and serve these various image types properly. All web server software today is configured to handle PNG, JPEG, and GIF out of the box, but if you are using SVG or one of the newer formats, you may need to deliberately add that media type to the server's official list.

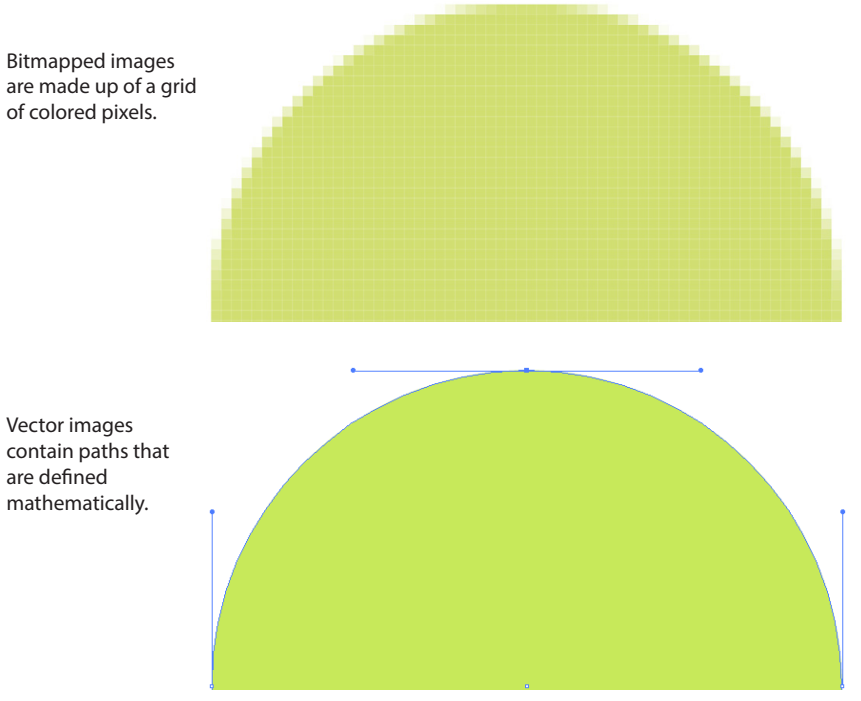


FIGURE 7-1. A comparison of circles saved in bitmapped and vector formats.

A little background information may be useful here. Image files, and indeed any media files that may reside on a server, have an official media type (also called a MIME type) and suffixes. For example, SVG has the MIME type *image/svg+xml* and the suffixes *.svg* and *.svgz*.

Server packages have different ways of handling MIME information. The popular Apache server software uses a file in the root directory called *htaccess* that contains a list of all the file types and their acceptable suffixes. Be sure to add (or ask your server administrator to add) the MIME types of new image formats so they may be served correctly. The server looks up the suffix (*.webp*, for example) of requested files in the list and matches it with the Content-Type (*image/webp*) that it includes in its HTTP response to the browser. That tells the browser what kind of data is coming and how to parse it.

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 browser 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.

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 **Chapter 4, Creating a Simple Page**. 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, aligned with the baseline of the text, and do not cause any line breaks (HTML5 calls this a phrasing element), as shown in FIGURE 7-2:

```
<p>This summer, try making pizza <img src="pizza.png" alt="">
 on your grill.</p>
```

FIGURE 7-2. By default, images are aligned with the baseline of the surrounding text and 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 or device, 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 using dial-up internet connections or other slow networks, like the expensive WiFi at luxury hotels.

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

The **src** and **alt** attributes shown in the sample are required. The **src** (source) attribute provides the location of the image file (its URL). 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.

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.
- It is an inline element, so it behaves like any other inline element in the text flow. FIGURE 7-3 demonstrates the inline nature of image elements. When the browser window is resized, a line of images reflows to fill the new width.

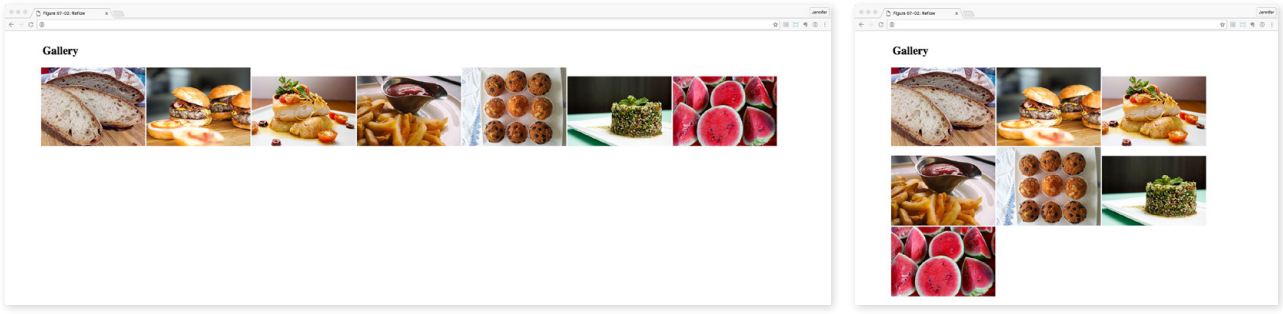


FIGURE 7-3. Inline images are part of the normal document flow. They reflow when the browser window is resized.

- 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 FIGURE 7-2. Using CSS, you can float the image to the right or left margin and allow text to flow around it, crop it to a shape, control the space and borders around the image, and change its vertical alignment. We'll talk about those styles in **Part III**.

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 own 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. In short, if the image is in the same directory as the HTML document, you can 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* or *img* (in fact, it helps search engines when you do it that way). 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 could place images from other websites by using a full URL, like this, but it is not recommended (see **Warning**):

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

WARNING

Before you use any image on your web page, be sure that you own the image, that you have explicit written permission by the copyright holder, or that it is in the public domain. Linking to an image on another server (called hotlinking) is considered seriously uncool, so don't do it unless there is a specific use case in which you have permission. Even then, be aware that you cannot control the image and risk having it moved or renamed, which would break your link.

■ PERFORMANCE TIP

Take Advantage of Caching

When a browser downloads an image, it stores the file 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 image without making a new server request.

If you use the same image repeatedly, be sure that the **src** attribute for each **img** element points to the same URL on the server. The image downloads once, then gets called from cache for subsequent uses. That means less traffic for the server and faster display for the user.

Providing Alternative Text with alt

`alt="text"`

Alternative text

Every **img** element must also contain an **alt** attribute that provides a text alternative to the image for those who are not able to see it. Alternative text (also called alt text) should serve as a substitute for the image content—conveying the same information and function. Alternative text is used by screen readers, search engines, and graphical browsers when the image doesn't load (FIGURE 7-4).

FIGURE 7-4. Most browsers display alternative text in place of the image if the image is not available. Safari for macOS is a notable exception. Firefox's substitution is the most seamless.

In this example, a PDF icon indicates that the linked text downloads a file in PDF format. In this case, the image is conveying valuable content that would be missing if the image cannot be seen. Providing the alt text "PDF file" replicates the purpose of the image:

```
<a href="application.pdf">High school application</a> <img src="images/→
pdflogo.png alt="PDF file">
```

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

"High school application. Image: PDF file"

Sometimes images function as links, in which case providing alternative text is critical because the screen reader needs something to read for the link. In the next example, an image of a book cover is used as a link to the book's website. Its alt text does not describe the cover itself, but rather performs the same function as the cover image on the page (indicating a link to the site):

```
<a href="http://learningwebdesign.com"><img src="/images/LWD_cover.png"
alt="Learning Web Design site"></a>
```

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 (null). In the following example, a decorative floral accent is not contributing to the content of the page, so its **alt** value is null. (You may also consider whether it is more appropriately handled as a background image in CSS, but I digress.) Note that there is no character space between the quotation marks:

```
<img src="/images/floralembellishment.png" alt="">
```

For each inline image on your page, consider what the alternative text would sound like when read aloud and whether that enhances the experience or might be obtrusive to a user with assistive technology.

Alternative text may benefit users with graphical browsers as well. If the 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 7-4.

■ ACCESSIBILITY TIP

Avoid using "image of" or "graphic of" in alt text values. It will be clear that it is an image. If the medium of the image, for example painting, photograph, or illustration, is relevant to the content, then it is fine to include the descriptive term.

Providing the Dimensions with width and height

The **width** and **height** attributes indicate the dimensions of the image in number of pixels. 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, resulting in faster page display. If only one dimension is set, the image will scale proportionally.

`width="number"`
Image width in pixels

`height="number"`
Image height in pixels

These attributes have become less useful in the age of modern web development. They should never be used to resize an image (use your image-editing program or CSS for that), and they should be omitted entirely when you're using one of the responsive image techniques introduced later in this chapter. They may be used with images that will appear at the same fixed size across all devices, such as a logo or an icon, to give the browser a layout hint.

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 7-5). If you are using **width** and **height** attributes and your image looks distorted or even slightly blurry, check to make sure that the values are in sync.

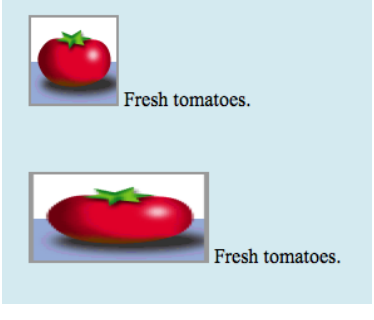


FIGURE 7-5. Browsers resize images to match the provided **width** and **height** values, but you should not resize images this way.

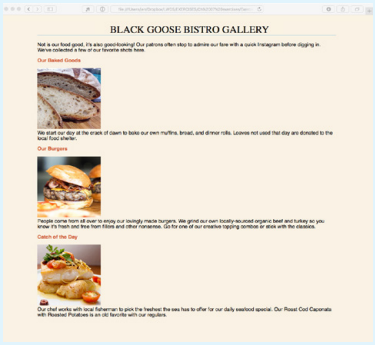
Now that you know the basics of the **img** element, you should be ready to add a few photos to the Black Goose Bistro Gallery site in EXERCISE 7-1.

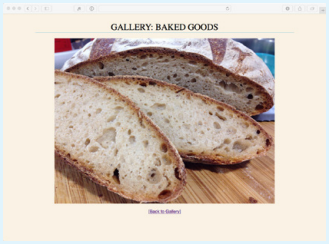
■ FURTHER READING

Image Accessibility

Some types of images, such as data charts and diagrams, require long descriptions that aren't practical as **alt** values. These cases require alternate accessibility strategies, which you will find in these resources:

- "Accessible Images" at WebAIM (*webaim.org/techniques/images/*)
- "Alternative Text" at WebAIM (*webaim.org/techniques/alttext/*)
- The Web Content Accessibility Guidelines (WCAG 2.0) at the W3C (*www.w3.org/TR/WCAG20-TECHS/*) include techniques for improving accessibility across all web content. Warning: this one is pretty dense.





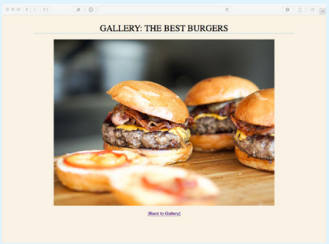


FIGURE 7-6. Photo gallery pages.

Like more practice?

If you'd like more practice, you'll find three additional images (*chicken-800.jpg*, *fries-800.jpg*, and *tabouleh-800.jpg*) with their thumbnail versions (*chicken-200.jpg*, *fries-200.jpg*, and *tabouleh-200.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.

EXERCISE 7-1. Adding and linking images

In this exercise, you'll add images to pages and use them as links. All of the full-size photos and thumbnails (small versions of the images) you need have been created for you, and I've given you a head start on the HTML files with basic styles as well. The starter files and the resulting code are available at *learningwebdesign.com/5e/materials.* Put a copy of the *gallery* folder on your hard drive, making sure to keep it organized as you find it.

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 7-6). 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:

```
<p><img src="thumbnails/bread-200.jpg" alt="close-up of sliced rustic
bread" width="200" height="200"><br>We start our day at the…
```

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 added a description of the image with the **alt** attribute, and because I know these thumbnails will appear at exactly 200 pixels wide and high on all devices, I've included the **width** and **height** attributes as well to tell the browser how much space to leave in the layout. Now it's your turn.

1. Add the thumbnail images *burgers-200.jpg* and *fish-200.jpg* at the beginning of the paragraphs in their respective sections, following my example. Be sure to include the pathnames and thoughtful alternative text descriptions. Finally, add a line break (**
) after the **img element.

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

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

   ```
   <h1>Gallery: Baked Goods</h1>
   <p><img src="photos/bread-800.jpg" alt="close-up of sliced rustic
   bread" width="800" height="600"></p>
   ```

   Notice that the full-size images are in a directory called *photos*, so that needs to be reflected in the pathnames. Notice also that because this page is not designed to be responsive, and the images will be a fixed size across devices, I went ahead and included the **width** and **height** attributes here as well.

   Add images to *burgers.html* and *fish.html*, following my example. Hint: all of the images are 800 pixels wide and 600 pixels high.

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

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

   ```
   <p><a href="bread.html"><img src="thumbnails/bread-200.jpg" alt="close-
   up of sliced rustic bread" width="200" height="200"></a><br>We start
   our day at the crack of dawn…
   ```

   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 link 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!

That takes care of the basics of adding images to a page. Next we'll take on adding SVG images, which are a special case, both in terms of the underlying format and the ways they can be added to HTML.

ADDING SVG IMAGES

No lesson on adding images to web pages would be complete without an introduction to adding SVGs (Scalable Vector Graphics). After all, the popularity of SVG images has been gaining momentum thanks to nearly ubiquitous browser support and the need for images that can resize without loss of quality. For illustration-style images, they are a responsive dream come true. I'm saving my deep-dive into all things SVG for **Chapter 25**, but for now I'll give you a quick peek at what they're made of so that the embedding markup makes sense.

As I mentioned at the beginning of this chapter, SVGs are an appropriate format for storing vector images (FIGURE 7-1). Instead of a grid of pixels, vectors are made up of shapes and paths that are defined mathematically. And even more interesting, in SVGs those shapes and paths are specified by instructions written out in a text file. Let that sink in: they are *images* that are written out in *text*! All of the shapes and paths as well as their properties are written out in the standardized SVG markup language (see **Note**). As HTML has elements for paragraphs (**p**) and tables (**table**), SVG has elements that define shapes like rectangle (**rect**), circle (**circle**), and paths (**path**).

A simple example will give you the general idea. Here is the SVG code that describes a rectangle (**rect**) with rounded corners (**rx** and **ry**, for x-radius and y-radius) and the word "hello" set as **text** with attributes for the font and color (FIGURE 7-7). Browsers that support SVG read the instructions and draw the image exactly as I designed it:

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 300 180">
  <rect width="300" height="180" fill="purple" rx="20" ry="20"/>
  <text x="40" y="114" fill="yellow" font-family="'Verdana-Bold'"
font-size="72">
    hello!
  </text>
</svg>
```

FIGURE 7-7. A simple SVG made up of a rectangle and text.

NOTE

SVG is an example, or application, of XML (Extensible Markup Language), which provides the rules and standards for how markup languages should be written and work together. As a result, SVG plays well alongside HTML content.

SVGs offer some significant advantages over their bitmapped counterparts for certain image types:

- Because they save only instructions for what to draw, they generally require less data than an image saved in a bitmapped format. That means faster downloads and better performance.
- Because they are vectors, they can resize as needed in a responsive layout without loss of quality. An SVG is always nice and crisp. No fuzzy edges.
- Because they are text, they integrate well with HTML/XML and can be compressed with tools like Gzip and Brotli, just like HTML files.
- They can be animated.
- You can change how they look with Cascading Style Sheets.
- You can add interactivity with JavaScript so things happen when users hover their mouse over or click the image.

Again, all of the ins and outs of creating SVGs, as well as their many features, are discussed in detail in **Chapter 25**. For now, I'd like to focus on the HTML required to place them in the flow of a web page. You have a few options: embedded with the **`img`** element, written out in code as an inline **`svg`** element, embedded with **`object`**, and used as a background image with CSS.

Embedded with the img Element

SVG text files saved with the *.svg* suffix (sometimes referred to as a standalone SVG) can be treated as any other image, including placing it in the document by using the **`img`** element. You're an expert on the **`img`** element by now, so the following example should be clear:

```
<img src="/images/circle.svg" alt="">
```

Pros and cons

The advantage to embedding an SVG with **`img`** is that it is universally supported in browsers that support SVG.

This approach works fine when you are using a standalone SVG as a simple substitute for a GIF or a PNG, but there are a few disadvantages to embedding SVGs with **`img`**:

- You cannot apply styles to the items within the SVG by using an external style sheet, such as a *.css* file applied to the whole page. The *.svg* file may include its own internal style sheet using the **`style`** element, however, for styling the elements within it. You can also apply styles to the **`img`** element itself.
- You cannot manipulate the elements within the SVG with JavaScript, so you lose the option for interactivity. Scripts in your web document can't

see the content of the SVG, and scripts in the SVG file do not run at all. Other interactive effects, like links or **:hover** styles, are never triggered inside an SVG embedded with **img** as well.

- You can't use *any* external files, such as embedded images or web fonts, within the SVG.

In other words, standalone SVGs behave as though they are in their own little, self-contained bubble. But for static illustrations, that is just fine.

Browser support for SVG with img

The good news is that all modern browsers support SVGs embedded with the **img** element. The two notable exceptions are Internet Explorer versions 8 and earlier, and the Android browser prior to version 3. As of this writing, users with those browsers may still show up in small but significant numbers in your user logs. If you see a reason for your site to support these older browsers, there are workarounds, which I address briefly in the upcoming **"SVG Fallbacks"** section.

SVG Server Configuration

If you are using SVGs and they are not showing up correctly when your site is uploaded, you may need to configure the server to recognize the SVG image type, as discussed at the beginning of this chapter. Here's how to do it on the Apache server, but similar configurations can be done in other server languages:

```
AddType image/svg+xml .svg
```

Inline in the HTML Source

Another option for putting an SVG on a web page is to copy the content of the SVG file and paste it directly into the HTML document. This is called using the SVG inline. Here is an example that looks a lot like the inline **img** example that we saw way back in FIGURE 7-2, only this time our pizza is a vector image drawn with circles and inserted with the **svg** element (FIGURE 7-8). Each **circle** element has attributes that describe the fill color, the position of its center point (**cx** and **cy**), and the length of its radius (**r**):

<svg>
An inline SVG image

```
<p>This summer, try making pizza

<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 72 72" width="100"
height="100">
  <circle fill="#D4AB00" cx="36" cy="36" r="36"/>
  <circle opacity=".7" fill="#FFF" stroke="#8A291C" cx="36.1" cy="35.9"
r="31.2"/>
  <circle fill="#A52C1B" cx="38.8" cy="13.5" r="4.8"/>
  <circle fill="#A52C1B" cx="22.4" cy="20.9" r="4.8"/>
  <circle fill="#A52C1B" cx="32" cy="37.2" r="4.8"/>
  <circle fill="#A52C1B" cx="16.6" cy="39.9" r="4.8"/>
  <circle fill="#A52C1B" cx="26.2" cy="53.3" r="4.8"/>
  <circle fill="#A52C1B" cx="42.5" cy="27.3" r="4.8"/>
  <circle fill="#A52C1B" cx="44.3" cy="55.2" r="4.8"/>
  <circle fill="#A52C1B" cx="54.7" cy="42.9" r="4.8"/>
  <circle fill="#A52C1B" cx="56" cy="28.3" r="4.8"/>
</svg>

 on your grill.</p>
```

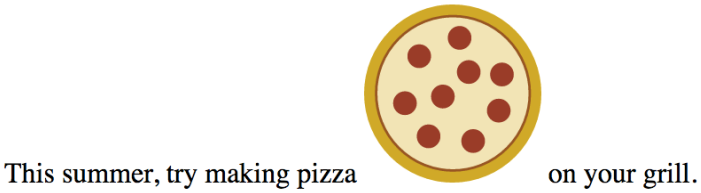


FIGURE 7-8. This pizza image is an SVG made up of 11 `circle` elements. Instead of an `img` element, the SVG source code is placed right in the HTML document with an `svg` element.

This code was generated by Adobe Illustrator, where I created the illustration and saved it in SVG format. I also optimized it to strip out a lot of cruft that Illustrator adds in there. We'll discuss SVG optimization in **Chapter 25**.

Pros and cons

Inline SVGs allow developers to take full advantage of SVG features. When the SVG markup is alongside the HTML markup, all of its elements are part of the main DOM tree. That means you can access and manipulate SVG objects with JavaScript, making them respond to user interaction or input. There are similar benefits for style sheets because the elements in the SVG can inherit styles from HTML elements. That makes it easy to apply the same styles to elements on the page and within the SVG graphic.

On the downside, the code for SVG illustrations can get extremely long and unwieldy, resulting in bloated HTML documents that are difficult to read. Even that little pepperoni pizza requires a serious block of code. It also makes the images for a site more difficult to maintain, since they are tucked away in the HTML documents. Another disadvantage is that inline SVGs are not cached by the browser separate from the HTML file, so avoid this method for large images that are reused across many HTML pages.

Browser support

The good news is that all modern browsers support SVG images placed inline with the `svg` element. The following older browser versions lack support: Internet Explorer versions 8 and earlier, Safari versions 5 and earlier, Android mobile browser prior to version 3, and iOS prior to version 5.

Embedded with the object Element

HTML has an all-purpose media embedding element called `object`. We'll talk about it more in **Chapter 10, Embedded Media**, but for now, know that `object` is another option for embedding an SVG in a web page. It is a good

compromise between **img** and inline SVG, allowing a fully functional SVG that is still encapsulated in a separate, cacheable file.

The opening **object** tag specifies the media type (an **svg+xml** image) and points to the file to be used with the **data** attribute. The **object** element comes with its own fallback mechanism—any content within the **object** gets rendered if the media specified with **data** can't be displayed. In this case, a PNG version of the image will be placed with an **img** if the *.svg* is not supported or fails to load:

```
<object type="image/svg+xml" data="pizza.svg">
  <img src="pizza.png" alt="pizza">
</object>
```

There is one catch, however. Some browsers download the fallback image even if they support SVG and don't need it. Useless downloads are not ideal. The workaround is to make the fallback image a CSS background image in an empty **div** container. Unfortunately, it is not as flexible for scaling and sizing, but it does solve the extra download issue.

```
<object type="image/svg+xml" data="pizza.svg">
  <div style="background-image: url(pizza.png); width 100px; height:
100px;" role="img" aria-label="pizza">
</object>
```

Pros and cons

The main advantage to embedding SVGs with the **object** element is that they can be scripted and load external files. They can also use scripts to access the parent HTML document (with some security restrictions). However, because they are separate files and not part of the DOM for the page, you can't use a style sheet in the HTML document to style elements within the SVG. Embedded SVGs may also have some buggy behaviors in browsers, so be sure to test thoroughly.

Other Embedding Options

Older techniques for adding SVGs involve using two other HTML elements for embedding media: **embed** and **iframe** (we'll talk about them in **Chapter 10**). You may still see these in use with SVGs out there, and they work fine for browsers that support SVG, but most developers consider them to be outdated methods. Stick with **img**, inline **svg**, **object**, and CSS **background-image**.

Used as a Background Image with CSS

I know that this is an HTML chapter, but I'd be remiss if I didn't at least mention that SVGs can be used as background images with CSS. This style rule example puts a decorative image in the background of a **header**:

```
header {
  background-image: url(/images/decorative.svg);
}
```

SVG Fallbacks

As mentioned earlier, all modern browsers support SVGs either embedded as an **img**, embedded as an **object**, or included inline, which is very good news. However, if your server logs show significant traffic from Internet Explorer 8 and earlier, Android version 3 and earlier, or Safari 5 and earlier, or if your

client just requires support for those browsers, you may need to use a fallback technique. One option is to use the **object** element to embed the SVG on the page and take advantage of its fallback content feature shown earlier.

If you are using SVG as an image with the **img** element, another option is to use the **picture** element (it's discussed as part of the **"Responsive Image Markup"** section later in this chapter). The **picture** element can be used to provide several versions of an image in different formats. Each version is suggested with the **source** element, which in the following example points to the *pizza.svg* image and defines its media type. The **picture** element also has a built-in fallback mechanism. If the browser doesn't support the suggested **source** files, or if it does not support the **picture** element, users will see the PNG image provided with the good old **img** element instead:

```
<picture>
  <source type="image/svg+xml" srcset="pizza.svg">
  <img srcset="pizza.png" alt="No SVG support">
</picture>
```

If you Google for "SVG fallbacks," you'll likely get quite a few hits, many of which use JavaScript to detect support. For more detailed information on SVG fallbacks, I recommend reading Amelia Bellamy-Royd's article, "A Complete Guide to SVG Fallbacks" (*css-tricks.com/a-complete-guide-to-svg-fallbacks/*) or Chris Coyier's book, *Practical SVG* (A Book Apart) when you are ready. Ideally, you will be reading this in a world where old Internet Explorer and Android versions are no longer an issue.

Are you ready to give SVGs a spin? Try out some of the embedding techniques we discussed in EXERCISE 7-2.

EXERCISE 7-2. Adding an SVG to a page

In this exercise, we'll add some SVG images to the Black Goose Bistro page that we worked on in **Chapter 4**. The materials for this exercise are available online at *learningwebdesign.com/5e/materials*. You will find everything in a directory called *svg*. The resulting code is provided with the materials.

This exercise has two parts: first, we'll replace the logo with an SVG version, and second, we'll add a row of social media icons at the bottom of the page (FIGURE 7-9).

Part I: Replacing the logo

1. Open *blackgoosebistro.html* in a text editor. It should look just like we left it in **Chapter 4**.
2. Just for fun, let's see what happens when you make the current PNG logo really large. Add **width="500" height="500"** to the **img** tag. Save the file and open it in the browser to see how blurry bitmapped images get when you size them larger. Yuck.
3. Let's replace it with an SVG version of the same logo by using the inline SVG method. In the *svg* folder, you will find a file called *blackgoose-logo.svg*. Open it in your text editor and copy all of the text (from **<svg>** to **</svg>**).

FIGURE 7-9. The Black Goose Bistro page with SVG images.

4. Go back to the *blackgoosebistro.html* file and delete the entire **img** element (be careful not to delete the surrounding markup). Paste the SVG text in its place. If you look closely, you will see that the SVG contains two circles, a gradient definition, and two paths (one for the starburst shape and one for the goose).
5. Next, set the size the SVG should appear on the page. In the opening **svg** tag, add **width** and **height** attributes set to 200px each.

   ```
   <h1><svg width="200px" height="200px" …
   ```

   Save the file and open the page in the browser. You should see the SVG logo in place, looking a lot like the old one.
6. Try seeing what happens when you make the SVG logo really big! Change the width and height to 500 pixels, save the file, and reload the page in the browser. It should be big and *sharp*! No blurry edges like the PNG. OK, now put the size back to 200 × 200 or whatever looks good to you.

Part II: Adding icons

7. Next we're going to create a footer at the bottom of the page for social media icons. Below the Location & Hours section, add the following (the empty paragraph is where we'll add the logos):

   ```
   <footer>
     <p>Please visit our social media pages</p>
     <p> </p>
   </footer>
   ```

8. Use the **img** element to place three SVG icons: *twitter.svg*, *facebook.svg*, and *instagram.svg*. Note that they are located in the *icons* directory. There are also icons for Tumblr and GitHub if you'd like extra practice. Here's a head start on the first one:

   ```
   <p><img src="icons/twitter.svg" alt="twitter"></p>
   ```

9. Save the file and open it in the browser. The icons should be there, but they are *huge*. Let's write a couple of style rules to make the footer look nice. We haven't done much with style rules yet, so just copy exactly what you see here inside the **style** element in the **head** of the document:

   ```
   footer {
     border-top: 1px solid #57b1dc;
     text-align: center;
     padding-top: 1em;
   }
   footer img {
     width: 40px;
     height: 40px;
     margin-left: .5em;
     margin-right: .5em;
   }
   ```

10. Save the file again and open it in the browser (you should see a page that looks like FIGURE 7-9). Go ahead and play around with the style settings, or even the code in the inline SVG, if you'd like to get a feel for how they affect the appearance of the images. It's kinda fun.

RESPONSIVE IMAGE MARKUP

Pretty quickly after smartphones, tablets, "phablets," and other devices hit the scene, it became clear that large images that look great on a large screen were overkill on smaller screens. All that image data...downloaded and wasted. Forcing huge images onto small devices slows down page display and may cost real money too, depending on the user's data plan (and your server costs). Conversely, small images that download quickly may be blurry on large, high-resolution screens. Just as we need a way to make whole web pages respond and adapt to various screen sizes, we need a way to make images on those page "responsive" as well. Our trusty `img` element with its single `src` attribute just doesn't cut it in most cases.

You provide multiple images, sized or cropped for different screen sizes, and the browser picks the most appropriate one based on what it knows about the current viewing environment.

It took a couple of years of proposals, experimentation, and discussion between browser makers and the web development community, but we now have a way to suggest alternate images by using HTML markup alone. No complicated JavaScript or server-side hacks. The resulting responsive image features (`srcset` and `sizes` attributes as well as the `picture` element) have been incorporated into the HTML 5.1 specification, and browser support is growing steadily, led by the Chrome browser in September 2014.

Thanks to a foolproof fallback and scripts that add support to older browsers, you can start using these techniques right away. That said, none of this is set in stone. Responsive image solutions are likely to be tweaked and improved, or perhaps one day even made obsolete. If you are going to include them in your sites, a good starting place for getting up-to-speed is the Responsive Images Community Group (*responsiveimages.org*). RICG is a group of developers who worked together to hammer out the current spec with the browser creators. They are on top of this stuff. You should also look for recent articles and perhaps even crack open the spec.

How It Works

When we say "responsive images," we are talking about providing images that are tailored to the user's viewing environment. First and foremost, responsive image techniques prevent browsers on small screens from downloading more image data than they need. They also include a mechanism to give high-resolution displays on fast networks images large enough to look extra-gorgeous. In addition, they provide a way for developers to take advantage of new, more efficient image formats.

In short, responsive images work this way: you provide multiple images, sized or cropped for different screen sizes, and the browser picks the most appropriate one based on what it knows about the current viewing environment. Screen dimensions are one factor, but resolution, network speed, what's already in its cache, user preferences, and other considerations may also be involved.

The responsive image attributes and elements address the following four basic scenarios:

- Providing extra-large images that look crisp on **high-resolution screens**
- Providing a set of images of various dimensions for use on **different screen sizes**
- Providing versions of the image with varying amount of detail based on the device size and orientation (known as the **art direction** use case)
- Providing **alternative image formats** that store the same image at much smaller file sizes

Let's take a look at each of these common use cases.

High-Density Displays (x-descriptor)

Everything that you see on a screen display is made up of little squares of colored light called pixels. We call the pixels that make up the screen itself device pixels (you'll also sometimes see them referred to as hardware pixels or physical pixels). Until recently, screens commonly fit 72 or 96 device pixels in an inch (now 109 to 160 is the norm). The number of pixels per inch (ppi) is the resolution of the screen.

Bitmapped images, like JPEG, PNG, and GIF, are made up of a grid of pixels too. It used to be that the pixels in images as well as pixel dimensions specified in our style sheets mapped one-to-one with the device pixels. An image or box element that was 100 pixels wide would be laid out across 100 device pixels. Nice and straightforward.

Device-pixel-ratios

It should come as no surprise that it's not so straightforward today. Manufacturers have been pushing screen resolutions higher and higher in an effort to improve image quality. The result is that device pixels have been getting smaller and smaller, so small that our images and text would be illegibly tiny if they were mapped one-to-one.

Devices use a measurement called a reference pixel for layout purposes.

To compensate, devices use a measurement called a reference pixel for layout purposes. Reference pixels are also known as points (PT) in iOS, Device Independent Pixels (DP or DiP) in Android, or CSS pixels because they are the unit of measurement we use in style sheets. The iPhone 8 has a screen that is made up of 750 × 1334 device pixels, but it uses a layout grid of 375 × 667 points or CSS pixels (a ratio of 2 device pixels to 1 layout pixel—2:1 or 2x). A box sized to 100 pixels wide in CSS would be laid out across 200 device pixels on the iPhone8. The iPhone X has a screen that is made up of 1125 × 2436 pixels, but it uses a layout grid of 375 × 812 points (a ratio of 3 device pixels to one point—or 3x). A box sized to 100 pixels is laid out across 300 device pixels on the iPhone X.

The ratio of the number of device pixels to CSS pixels is called the device-pixel-ratio (FIGURE 7-10). Common device-pixel-ratios on handheld devices are 1.325x, 1.5x, 1.7x, 2x, 2.4x, 3x, and even 4x (the "x" is the convention for indicating a device-pixel-ratio). Even large desktop displays are featuring ratios of 2x, 3x, and 4x.

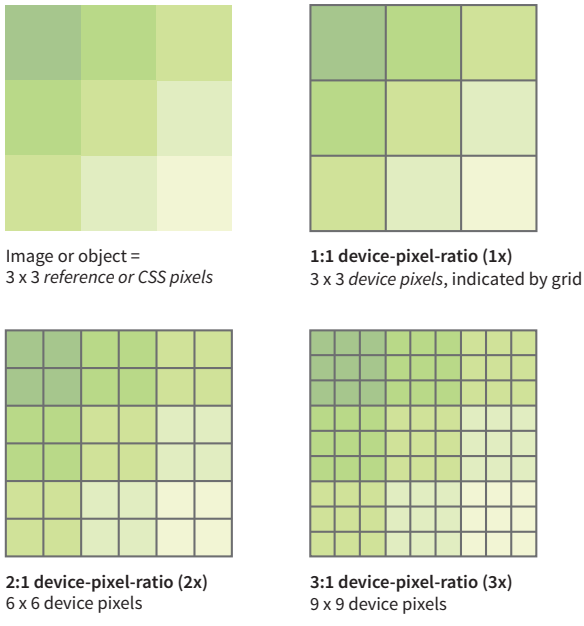


FIGURE 7-10. Device pixels compared to CSS/reference pixels.

Let's say you have an image that you want to appear 200 pixels wide on all displays. You can make the image exactly 200px wide (px is short for pixels), and it will look fine on standard-resolution displays, but it might be a little blurry on high-resolution displays. To get that image to look sharp on a display with a device-pixel-ratio of 2x, you'd need to make that same image 400 pixels wide. It would need to be 600 pixels wide to look sharp on a 3x display. Unfortunately, the larger images may have file sizes that are four or more times the size of the original. Who wants to send all that extra data to a 1x device that really only needs the smaller image?

Introducing srcset

We now have a way to serve larger images just to the browsers on displays that benefit from them. We do it using the new **`srcset`** attribute with our old friend the **`img`** element. **`srcset`** allows developers to specify a list of image source options for the browser to choose from.

The value of **`srcset`** is a comma-separated list of options. Each item in that list has two parts: the URL of an image and an x-descriptor that specifies the target device-pixel-ratio. Note that the whole list is the value of **`srcset`** and

goes inside a single set of quotation marks. This sample shows the structure of a **srcset** value:

```
srcset="image-URL #x, image-URL #x"
```

The **src** attribute is still required, and is generally used to specify the default 1x image for browsers that don't support **srcset**. Make sure there is an **alt** attribute as well:

```
<img src="image-URL" alt="" srcset="image-URL #x, image-URL #x">
```

Let's look at an example. I have an image of a turkey that I'd like to appear 200 pixels wide. For standard resolution, I created the image at 200 pixels wide and named it *turkey-200px.jpg*. I'd also like it to look crisp in high-resolution displays, so I have two more versions: *turkey-400px.jpg* (for 2x) and *turkey-600px.jpg* (for 3x). Here is the markup for adding the image and indicating its high-density equivalents with x-descriptors:

```
<img src="/images/turkey-200px.jpg" alt=""
srcset="/images/turkey-400px.jpg 2x, /images/turkey-600px.jpg 3x" >
```

Because browsers ignore line returns and spaces in the source document, I can also write that same element stacked in this way to make it a little easier to read, as I will be doing throughout this chapter:

```
<img
  src="/images/turkey-200px.jpg" alt=""
  srcset="/images/turkey-400px.jpg 2x,
          /images/turkey-600px.jpg 3x" >
```

That makes the options and structure more clear at a glance, don't you think?

Browsers that recognize the **srcset** attribute check the screen resolution and download what they believe to be the most appropriate image. If the browser is on a Mac with a 2x Retina display, it may download *image-400px.jpg*. If the device-pixel-ratio is 1.5x, 2.4x, or something else, it checks the overall viewing environment and makes the best selection. It is important to know that when we use **srcset** with the **img** element, we are handing the keys to the browser to make the final image selection.

The srcset attribute specifies a list of image options for the browser to choose from.

When we use srcset with the img element, we are allowing the browser to make the best image selection.

When to use x-descriptors

X-descriptors tell the browser to make a selection based on screen resolution only, with no regard for the dimensions of the screen or viewport. For this reason, x-selectors are best used for images that stay the same pixel dimensions regardless of the screen size, such as logos, social media badges, or other fixed-width images.

It is much more likely that you'll want images to resize based on the size of the screen and to be able to serve small images to small handheld devices, and large images to desktops (that's kind of the crux of this responsive image thing, after all). Now that you are familiar with using the **srcset** attribute, let's see how it can be used to deliver images targeted to various screen sizes. Here's where **srcset** really shines.

Variable-Width Images (w-descriptor)

When you're designing a responsive web page, chances are you'll want image sizes to change based on the size of the browser viewport (see **Note**). This is known as a viewport-based selection. And because you are the type of web developer who cares about how fast pages display, you'll want to limit unnecessary data downloads by providing appropriately sized images.

NOTE

On a mobile device, the viewport fills the whole screen. On a desktop browser, the viewport is the area where the page displays, not including the scrollbars and other browser "chrome."

To achieve this goal, use the **srcset** and **sizes** attributes with the **img** element. As we saw in previous examples, the **srcset** gives the browser a set of image file options, but this time, it uses a w-descriptor (width descriptor) that provides the *actual pixel width* of each image. Using **srcset** with a w-descriptor is appropriate when the images are identical except for their dimensions (in other words, they differ only in scale). Here's an example of a **srcset** attribute that provides four image options and specifies their respective pixel widths via w-descriptors. Note again that the whole list is in a single set of quotation marks:

```
srcset="strawberries-480.jpg 480w,
        strawberries-960.jpg 960w,
        strawberries-1280.jpg 1280w,
        strawberries-2400.jpg 2400w"
```

Using the sizes attribute

The sizes attribute is required when you use width descriptors.

That's a good start, but whenever you use w-descriptors, you also need to use the **sizes** attribute to tell the browser the approximate size that the image will appear in the page's layout. There is a very good reason (in addition to being required in the spec), and it is worth understanding.

When a browser downloads the HTML document for a web page, the first thing it does is look through the whole document and establish its outline structure (its Document Object Model, or DOM). Then, almost immediately, a preloader goes out to get all the images from the server so they are ready to go. Finally, the CSS and the JavaScript are downloaded. It is likely that the style sheet has instructions for layout and image sizes, but by the time the browser sees the styles, the images are already downloaded. For that reason, we have to give the browser a good hint with the **sizes** attribute whether the image will fill the whole viewport width or only a portion of it. That allows the preloader to pick the correct image file from the **srcset** list.

We'll start with the simplest scenario in which the image is a banner and always appears at 100% of the viewport width, regardless of the device (FIGURE 7-11). Here's the complete **img** element:

```
<img src="strawberries-640.jpg"
     alt="baskets of ripe strawberries"
     srcset="strawberries-480.jpg 480w,
             strawberries-960.jpg 960w,
             strawberries-1280.jpg 1280w,
             strawberries-2400.jpg 2400w"
     sizes="100vw">
```

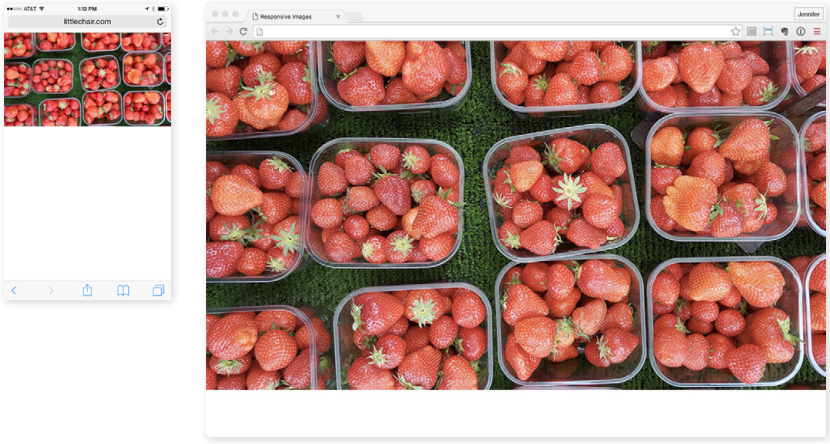


FIGURE 7-11. The image fills 100% of the viewport width, regardless of its size.

In this example, the **sizes** attribute tells the browser that the image fills the full viewport by using viewport width units (**vw**), the most common unit for the **sizes** attribute, so the browser can pick the best image for the job. For example, **100vw** translates to 100% of the viewport width, **50vw** would be 50%, and so on. You can also use **em**, **px**, and a few other CSS units, but you cannot use percentages. Browsers that do not support **srcset** and **sizes** simply use the image specified in the **src** attribute.

Browsers that do not support srcset and sizes use the image specified in the src attribute.

Sizing an image to fill the whole width of the browser is a pretty specific case. More likely, your images will be one component in a responsive page layout that resizes and rearranges to make best use of the available screen width. FIGURE 7-12 shows a sidebar of food photos that take up the full width of the screen on small devices, take up a portion of the width on larger devices, and appear three across in a layout for large browser windows.

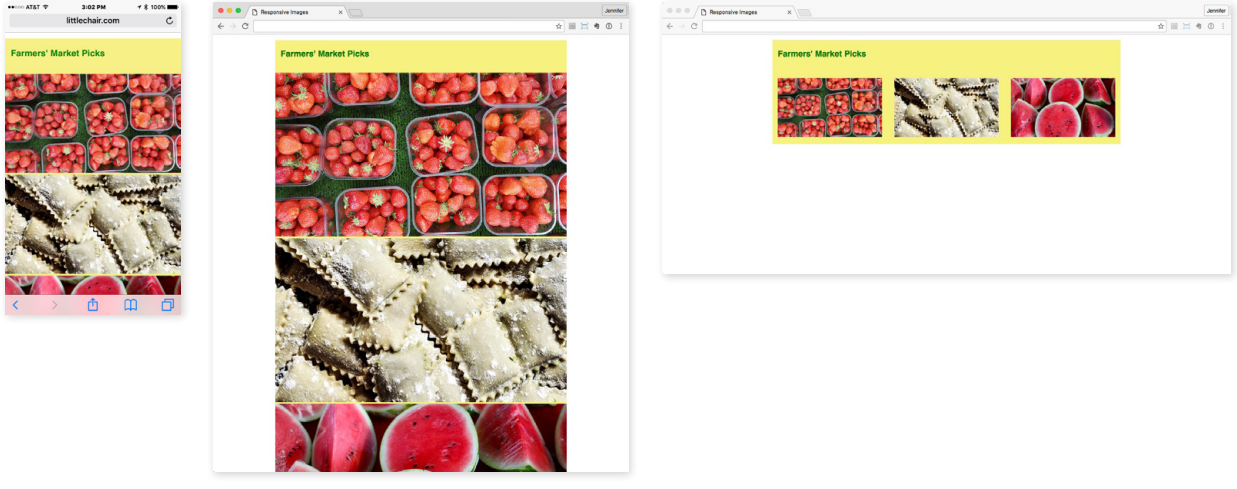


FIGURE 7-12. The width of the images changes based on the width of the viewport.

For cases like these, use the **sizes** attribute to tell the browser something about how the image will be sized for each layout. The **sizes** value is a comma-separated list in which each item has two parts. The first part in parentheses is a media condition that describes a parameter such as the width of the viewport. The second part is a length that indicates the width that image will occupy in the layout if the media condition is met. Here's how that syntax looks:

```
sizes="(media-feature: condition) length,
       (media-feature: condition) length,
       (media-feature: condition) length"
```

I've added some media conditions to the previous example, and now we have a complete valid **img** element for one of the photo images in FIGURE 7-12:

```
<img src="strawberries-640.jpg" alt="baskets of ripe strawberries"
    srcset="strawberries-240.jpg 240w,
            strawberries-480.jpg 480w,
            strawberries-672.jpg 672w"
     sizes="(max-width: 480px) 100vw,
            (max-width: 960px) 70vw,
            240px">
```

The **sizes** attribute tells the browser the following:

- If the viewport is 480 pixels wide or smaller (maximum width is 480 pixels), the image fills 100% of the viewport width.
- If the viewport is wider than 480 pixels but no larger than 960 pixels (**max-width: 960px**), then the image will appear at 70% of the viewport. (This layout has 15% margins on the left and the right of the images, or 30% total.)
- If the viewport is larger than 960 pixels and doesn't meet any of the prior media conditions, the image gets sized to exactly 240 pixels.

Now that the browser knows the width of the viewport and how big the image will appear within it, it can select the most appropriate image from the **srcset** list to download.

There's a bit more to using **sizes** than shown here—other media conditions, additional length units, even the ability to ask the browser to calculate widths for you. If you plan on using viewport-width-based images in your designs, I recommend reading the spec to take full advantage of the possibilities.

WARNING

*The **sizes** attribute will resize an image even if there is no CSS applied to it. If there is a CSS rule specifying image size that conflicts with the value of the **sizes** attribute, the style rule wins (i.e., it overrides the **sizes** value).*

NOTE

Strategies and tools for producing the image sets for responsive layouts are introduced in **Chapter 24, Image Asset Production**.

Art Direction (picture Element)

`<picture>…</picture>`
Specifies a number of image options

`<source>…</source>`
Specifies alternate image sources

So far, we've looked at image selection based on the resolution of the screen and the size of the viewport. In both of these scenarios, the content of the image does not change but merely resizes.

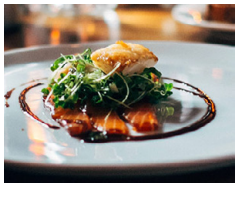
But sometimes, resizing isn't enough. You might want to crop into important details of an image when it is displayed on a small screen. You may want to change or remove text from the image if it gets too small to be legible. Or you might want to provide both landscape (wide) and portrait (tall) versions of the same image for different layouts.

For example, in FIGURE 7-13, the whole image of the table as well as the dish reads fine on larger screens, but at smartphone size, it gets difficult to see the delicious detail. It would be nice to provide alternate versions of the image that make sense for the browsing conditions.

Use the picture element when simply resizing the image is not enough.

That dinner looks delicious on desktop browsers.
(1280px wide)

Detail is lost when the full image is shrunk down on small devices.
(300px wide)



Cropping to the most important detail may make better sense.
(300px wide)

FIGURE 7-13. Some images are illegible when resized smaller for mobile devices.

This scenario is known as an art-direction-based selection and it is accomplished with the **`picture`** element. The **`picture`** element has no attributes; it is just a wrapper for some number of **`source`** elements and an **`img`** element. The **`img`** element is required and must be the last element in the list. If the **`img`** is left out, no image will display at all because it is the piece that is actually

placing the image on the page. Let's look at a sample **picture** element and then pick it apart:

```
<picture>
  <source media="(min-width: 1024px)" srcset="icecream-large.jpg">
  <source media="(min-width: 760px)" srcset="icecream-medium.jpg">
  <img src="icecream-small.jpg" alt="hand holding ice cream cone and
text that reads Savor the Summer">
</picture>
```

This example tells the browser that if the viewport is 1024 pixels wide or larger, use the large version of the ice cream cone image. If it is wider than 760 pixels (but smaller than 1024, such as on a tablet), use the medium version. Finally, for viewports that are smaller than 760 pixels and therefore don't match any of the media queries in the previous **source** elements, the small version should be used (FIGURE 7-14). The small version, as specified in the **img** element, will be used for browsers that do not recognize **picture** and **source**.

Each **source** element includes a **media** attribute and a **srcset** attribute. It can also use the **sizes** attribute, although that is not shown in the previous example. The **media** attribute supplies a media query for checking the current browsing conditions. It is similar to the media conditions we saw in the earlier **srcset** example, but the **media** attribute specifies a full-featured CSS media query (we'll talk more about media queries in **Chapter 17, Responsive Web Design**). The **srcset** attribute supplies the URL for the image to use if the media query is a match. In the previous example, there is just one image specified, but it could also be a comma-separated list if you wanted to provide several options using x- or w-descriptors.

Browsers download the image from the first **source** that matches the current conditions, so the order of the **source** elements is important. The URL provided in the **srcset** attribute gets passed to the **src** attribute in the **img**

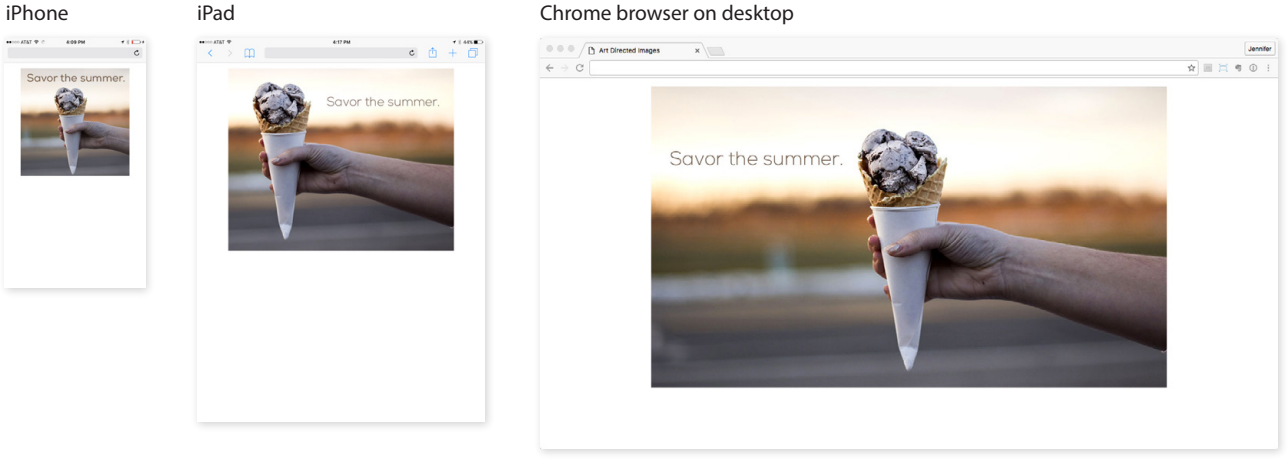


FIGURE 7-14. The **picture** element provides different image versions to be sourced at various screen sizes.

element. Again, it's the **img** that places the image on the page, so don't omit it. The **alt** attribute for the **img** element is required, but **alt** is not permitted in the **source** element.

Art direction is the primary use of the **picture** element, but let's look at one more thing it can do to round out our discussion on responsive images.

Alternative Image Formats (type Attribute)

Once upon a time, in the early 1990s, the only image type you could put on a web page was a GIF. JPEGs came along not long after, and we waited nearly a *decade* for reliable browser support for the more feature-rich PNG format. It takes a notoriously long time for new image formats to become universally supported. In the past, that meant simply avoiding newer formats.

In an effort to reduce image file sizes, more efficient image formats have been developed—such as WebP, JPEG 2000, and JPEG XR—that can compress images significantly smaller than their JPEG and PNG counterparts (see **Note**). And once again, some browsers support them and some don't. The difference is that today we can use the **picture** element to serve the newer image formats to browsers that can handle them, and a standard image format to browsers that can't. We no longer have to wait for universal browser support.

In the following example, the **picture** element specifies two image alternatives before the fallback JPEG listed in the **img** element:

```
<picture>
  <source type="image/webp" srcset="pizza.webp">
  <source type="image/jxr" srcset="pizza.jxr">
  <img src="pizza.jpg" alt="">
</picture>
```

For image-format-based selections, each **source** element has two attributes: the **srcset** attribute that we've seen before, and the **type** attribute for specifying the type of file (also known as its MIME type, see the **"File (MIME) Types"** sidebar). In this example, the first **source** points to an image that is in the WebP format, and the second specifies a JPEG XR. Again, the browser uses the image from the first source that matches the browser's image support, so it makes sense to put them in order from smallest to largest file size.

Browser Support

As I write this section, it seems like a new browser is adding support for **picture**, **srcset**, and **sizes** every day, but of course, old browser versions have a bad habit of sticking around for years. This is not a reason to avoid using responsive images, however. First of all, all of these features are designed to include the **img** element as a built-in fallback for browsers that don't recognize the newer markup. In the worst case, the browser grabs the image specified in the **img** element.

NOTE

The bitmapped image formats, including WebP, JPEG 2000, and JPEG XR, are discussed in more detail in **Chapter 23, Web Image Basics**.

File (MIME) Types

The web uses a standardized system to communicate the type of media files being transferred between the server and browser. It is based on MIME (Multipurpose Internet Mail Extension), which was originally developed for sending attachments via email. Every file format has a standardized type (such as **image**, **application**, **audio**, or **video**), subtype that identifies the specific format, and one or more file extensions. In our example, the **type** attribute specifies the WebP option with its type/subtype (**image/webp**) and uses the proper file extension (*.webp*). Other examples of media MIME types are **image/jpeg** (extensions *.jpg*, *.jpeg*), **video/mpeg** (extensions *.mpg*, *.mpe*, *.mpeg*, *.m1v*, *.mp2*, *.mp3*, and *.mpa*), and **application/pdf** (*.pdf*). The complete listing of registered MIME types is published by the IANA (Internet Assigned Numbers Authority) at *www.iana.org/assignments/media-types*.

If that isn't good enough, try including Picturefill with your web pages. Picturefill is an example of a polyfill, a script that makes older browsers behave as though they support a new technology—in this case, responsive images. It was created by Scott Jehl of Filament Group, creators of many fine responsive design and frontend development tools. Go to *scottjehl.github.io/picturefill/* to download the script and read the very thorough tutorial on how it works and how to use it.

Responsive Images Summary

This has been a long discussion about responsive images, and we've really only scratched the surface. We've looked at how to use the **`img`** element with **`srcset`** and **`sizes`** to make *pixel-ratio-based* and *viewport-size-based* selections (you can try them yourself in EXERCISE 7-3). We also saw how the **`picture`** element can be used for *art-direction-based* and *image-type-based* selections.

I've kept my examples short and sweet, but know that it is possible to combine techniques in different ways, often resulting in a tower of code for each image. To see some examples of how these responsive image techniques might be combined to target more than one condition, I recommend Andreas Bovens's article "Responsive Images: Use Cases and Documented Code Snippets to Get You Started" on the Dev.Opera site (*dev.opera.com/articles/responsive-images/*).

I also recommend the 10-part "Responsive Images 101" tutorial by Jason Grigsby at Cloud Four. He goes into a bit more detail than I was able to here and provides links to other good resources. Start with "Part 1: Definitions" (*cloudfour.com/thinks/responsive-images-101-definitions/*).

BROWSER SUPPORT TIP

The site *CanIUse.com* is a great tool for checking on the browser support for HTML, CSS, and other frontend web technologies. Type in **`picture`**, **`srcset`**, or **`sizes`** to see where browser support stands.

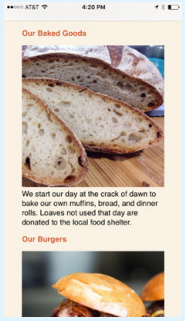
EXERCISE 7-3. Adding responsive images

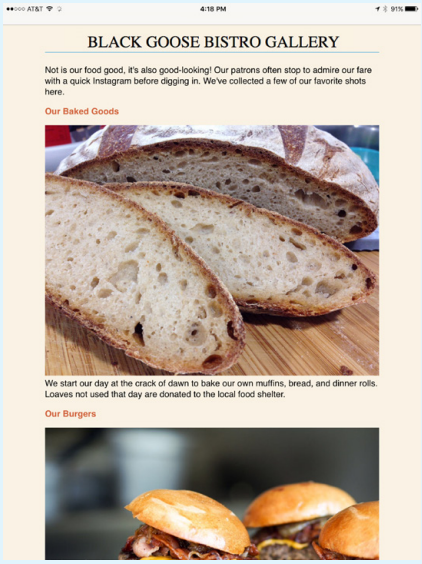
Ready to try out some of this responsive image stuff? I recommend downloading the latest version of Google Chrome (*google.com/chrome/*) or Firefox (*firefox.com*) so you are certain it supports the responsive image HTML features. The materials for this exercise are provided at *learningwebdesign.com/5e/materials*. Use the *responsivegallery* directory that contains a starter HTML file and *images* directory.

We're going to give the Black Goose Bistro Gallery page a makeover using responsive images. Now, instead of the user clicking a thumbnail and going to a separate page, the large images appear right on the page and resize to fill the available space. Small devices and browsers that don't support **`picture`** get a 400-pixel-square version of each image (FIGURE 7-15).

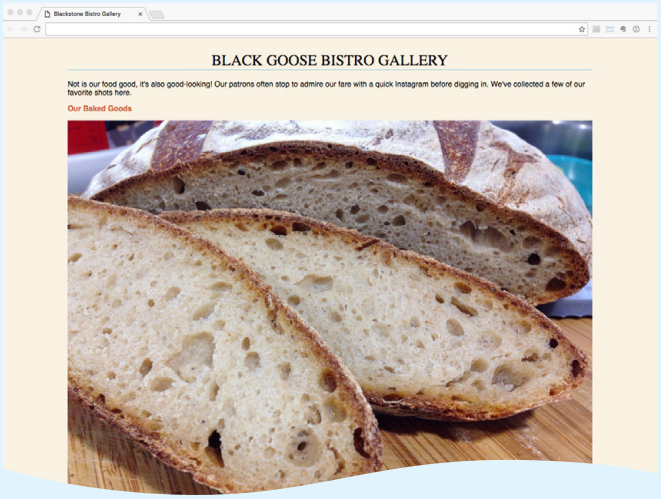
1. Open the file *index.html* located in the *responsivegallery* directory in a text or HTML editor. I've added a **`meta`** element that sets the viewport to the same size as the device width, which is required to make this page responsive. I also added a style for **`img`** elements that sets their maximum width to 100% of the available space. That is the bit that makes the images scale down for smaller screen widths. We'll talk a lot more about

Small devices like the iPhone show the cropped 400-pixel-square image.

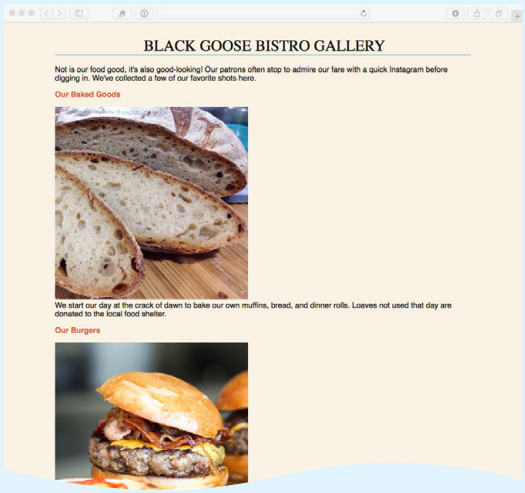




On viewports larger than 480 pixels, like the iPad shown here, the full version of the image is used. It resizes to fill the available width of the page between the margins.



On very large desktop displays, the full version of the image resizes to fill the available width.



Browsers that do not support **`picture`** display the 400-pixel-square image specified by the **`img`** element.

FIGURE 7-15. The Black Goose Bistro Gallery with responsive images in place. Smaller devices see a square cropped version of the image. Larger browsers get the full image that resizes to fill the content width.

responsive design in **Chapter 17**, so don't worry about it too much now. I just wanted to point out changes from our previous exercise.

2. Because we want to change between horizontal and square versions of the image on this page, we'll need to use the **`picture`** element. Start by adding the bare bones of a **`picture`** element in the first paragraph after "Our Baked Goods," including the **`picture`** wrapper and its required **`img`** element. The **`img`** element points to the default square version of the image (*bread-400.jpg*). Add a line break element after the **`picture`** element to start the text on the next line:

EXERCISE 7-3. Continued

```
<p>
<picture>
  <img src="images/bread-400.jpg" alt="close-up of sliced rustic
bread">
</picture>
<br>We start our day…
```

3. That takes care of small devices and the fallback for non-supporting devices. Now add a **source** element that tells browser to use a 1200-pixel-wide landscape version of the image when the viewport is larger than 480 pixels:

```
<p>
<picture>
  <source media="(min-width: 480px)"
          srcset="images/bread-1200.jpg">
  <img src="images/bread-400.jpg" alt="close-up of sliced rustic
bread">
</picture>
<br>We start our day…
```

   Note that because there is only one image specified in the **source**, we could have used a simple **src** attribute here, but we have more work to do, so the **srcset** gets us ready for the next step.

4. Because we don't want to force such a large image on everyone, let's give the browser an 800-pixel-wide version as well. (Even more versions would be useful, but for the sake of keeping this exercise manageable, we'll stop at two.) Remember that the **srcset** attribute specifies a comma-separated list of images and their respective pixel widths with w-descriptors. I've added the **1200w** descriptor to the original image and added the 800-pixel option to the **srcset**. Finally, use the **sizes** attribute to let the browser know that the image will occupy 80% of the viewport width (the style sheet adds a 10% margin on the left and right sides, leaving 80% for the content). Now the browser can choose the most appropriate size.

```
<p>
<picture>
  <source media="(min-width: 480px)"
          srcset="images/bread-1200.jpg 1200w,
                  images/bread-800.jpg 800w"
          sizes="80vw">
  <img src="images/bread-400.jpg" alt="close-up of sliced rustic
bread">
</picture>
<br>We start our day…
```

5. Save the file. Launch the Chrome or Firefox desktop browser and resize the window to as narrow as it will go. Open *index.html* and you should see the square cropped version of the bread photo. Slowly drag the corner of the browser window to make the window wider. When it gets wider than 480 pixels, it should switch to the full version of the photo. If you see a little "800" in the corner of the image, that means the browser has downloaded *bread-800.jpg* for this task. Keep expanding the window, and the image should keep getting larger. If you see "1200," it means it is using *bread-1200.jpg*. Once the larger image is in the browser's cache, you won't see the 800-pixel version again. Try making the window narrow and wide again and watch what changes. Congratulations! You are now an official responsive web designer! Making windows narrow and wide is how we spend a good portion of our workday.

6. Add the remaining two images to the page, following my example. Try experimenting with different min- and max-widths in the **media** attribute.

NOTE

If you don't see the images at all, it could be that your pathnames are incorrect or the *images* directory hasn't copied to your computer.

WHEW! WE'RE FINISHED

That wraps up our exploration of images. We've seen how to place images with the `img` element and its required `src` and `alt` attributes. We've talked about the importance of good alternative text for accessibility. We also looked at a few ways to embed SVG images into a web page. Finally, we took on the newly minted responsive image features, including `srcset` and `sizes` for the `img` element to target high-density displays or to provide a variety of image sizes for the browser to choose from, and the `picture` and `source` elements for art direction and alternative image formats. Now try answering a few questions to test your knowledge.

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. Name two reasons to include alternative text for an `img` element.

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

5. What is the difference between an x-descriptor and a w-descriptor?

6. What is the difference between a device pixel and a CSS (reference) pixel?

Alternatives to Responsive Images

Although it is terrific to have an HTML solution for getting the right images to the right browsers, the current system is cumbersome with stacks of code and the need to produce multiple images. If you work on an image-heavy site, it could prove to be unmanageable. Image processing is a task that begs to be automated. The solution: let the *server* do it!

Fortunately, there are many tools and services, both open source and for pay, that let the server do the work of creating appropriate image versions on the fly. You upload the largest available size of the image and let the server handle the rest—no need to create and store multiple versions of every image. In general, image-generation services address only resizing, and not art direction or alternative image types; however, at least one service (*Cloudinary.com*) uses face detection as a basis for image cropping.

Some content management systems (CMSs) have image resizing features built in. Another option is to install software on your own server. Bear in mind, however, that requiring JavaScript to be running is less than ideal. There are also many third-party solutions that provide image-resizing services (like Cloudinary.com and Kraken.io), usually for a fee. For large, image-heavy sites, they are worth looking into.

Jason Grigsby of Cloud Four has compiled a spreadsheet of image-resizing software and services that serves as a good jumping-off point. You can get to it from his article, "Image Resizing Services" (*cloudfour.com/thinks/image-resizing-services/*) or at *tinyurl.com/pmpbyzj*.

7. Match the responsive image scenarios with the HTML solutions:

   a. `<img src="" alt="" srcset="URL x-descriptor">`

   b. `<img src="" alt=" srcset="URL w-descriptor" sizes="#">`

   c.
   ```
   <picture>
     <source type="…" srcset="">
     <img src="" alt="">
   </picture>
   ```

   d.
   ```
   <picture>
     <source media="()" srcset="">
     <img src="" alt="">
   </picture>
   ```

   _____ You want the image to always fill the width of the browser window.

   _____ You want to take advantage of the file savings of the WebP image format.

   _____ You want to remove the text from an image when it is on small screens.

   _____ You want your product images to look as sharp as possible on high-resolution screens.

   _____ You want to show a close-up of the action in a news image on small screens.

   _____ You want the image to resize smaller when it is part of the layout on a large screen.

8. Challenge question: Describe what this example tells the browser to do:

```
<picture>
  <source sizes="(min-width: 480px) 80vw,
                 100vw"
          srcset="photo-200.webp 200w
                  photo-400.webp 400w,
                  photo-800.webp 800w,
                  photo-1200.webp 1200w"
          type="image/webp">
  <img src=" photo-400.jpg" alt=""
       sizes="(min-width: 480px) 80vw,
              100vw"
       srcset="photo-200.jpg 200w,
               photo-400.jpg 400w,
               photo-800.jpg 800w,
               photo-1200.jpg 1200w">
</picture>
```

9. What is cache and how does it affect web page performance?

10. Name one advantage and one disadvantage of adding an SVG to a page with the `img` element.

11. Name one advantage and one disadvantage of inline SVG.

12. When would it be appropriate to add an SVG to a page as a background image with CSS?

13. What is this bit of code describing, and when might you need to use it?

```
image/svg+xml
```

14. What is this bit of code describing, and where would you find it?

```
http://www.w3.org/2000/svg
```

PHOTO CREDITS

Many of the images in this chapter are from the fabulous royalty-free photo site, *Unsplash.com*: ravioli by Davide Ragusa, burgers by Niklas Rhöse, ice cream cone by Alex Jones, dinner table by Jay Wennington, strawberries by Priscilla Fong. From Flickr's "No Rights Restrictions" collection: fish dish by Renata Maia, muffins by Hasma Kanouni. All others are uncredited public domain images.

ELEMENT REVIEW: IMAGES

Following are the elements you learned in your exploration of image markup.

Element and attributes	Description
`img`	Inserts an inline image
`alt="text"`	Alternative text
`src="url"`	The location of the image file
`srcset="list of urls with descriptors"`	Images to use in different situations
`sizes="list media conditions and layout sizes"`	Image sizes for different layouts
`width="number"`	Width of the graphic
`height="number"`	Height of the graphic
`usemap="usemap"`	Indicates the client-side image map to use
`picture`	Container that provides multiple sources to its contained **`img`** element
`source`	Provides alternate sources for the **`img`** element
`src="URL"`	Address of the image resource
`srcset="URL"`	Images to use in different situations
`sizes="source size list"`	Image sizes for different page layouts
`media="media query"`	Query to determine applicable media
`type="media type"`	Media (MIME) type of embedded image file
`svg`	Adds an inline SVG image

CHAPTER 8

TABLE MARKUP

IN THIS CHAPTER

How tables are used

Basic table structure

Spanning rows and columns

Row and column groups

Making tables accessible

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 simple images on the page.

This chapter and the next two chapters, **Chapter 9, Forms**, and **Chapter 10, Embedded Media**, 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, forms, and media 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. 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 (that will be tackled in various CSS chapters in **Part III**).

HOW TO USE TABLES

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 schedules, product comparisons, 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.

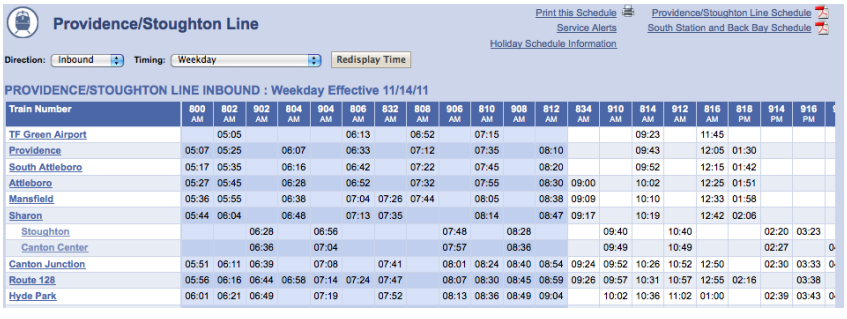
In visual browsers, the arrangement of data in rows and columns gives readers an instant understanding of the relationships between data cells and their respective header labels. Bear in mind when you are creating tables, however,

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; formBlock 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; formBlock candidate	flow	flow	globals	HTMLElement
aside	Sidebar for tangentially related content	flow; sectioning; formBlock 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; formBlock 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



Train Number	800 AM	802 AM	902 AM	804 AM	904 AM	806 AM	832 AM	808 AM	906 AM	810 AM	908 AM	812 AM	834 AM	910 AM	814 AM	912 AM	816 AM	818 PM	914 PM	916 PM
TF Green Airport		05:05				06:13		06:52		07:15					09:23		11:45			
Providence	05:07	05:25		06:07		06:33		07:12		07:35		08:10			09:43		12:05	01:30		
South Attleboro	05:17	05:35		06:16		06:42		07:22		07:45		08:20			09:52		12:15	01:42		
Attleboro	05:27	05:45		06:28		06:52		07:32		07:55		08:30	09:00		10:02		12:25	01:51		
Mansfield	05:36	05:55		06:38		07:04	07:26	07:44		08:05		08:38	09:09		10:10		12:33	01:58		
Sharon	05:44	06:04		06:48		07:13	07:35			08:14		08:47	09:17		10:19		12:42	02:06		
Stoughton			06:28		06:56				07:48		08:28			09:40		10:40			02:20	03:23
Canton Center			06:36		07:04				07:57		08:36			09:49		10:49			02:27	
Canton Junction	05:51	06:11	06:39		07:08		07:41		08:01	08:24	08:40	08:54	09:24	09:52	10:26	10:52	12:50		02:30	03:33
Route 128	05:56	06:16	06:44	06:58	07:14	07:24	07:47		08:07	08:30	08:45	08:59	09:26	09:57	10:31	10:57	12:55	02:16		03:38
Hyde Park	06:01	06:21	06:49		07:19		07:52		08:13	08:36	08:49	09:04		10:02	10:36	11:02	01:00		02:39	03:43

mbta.org

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

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. If you need rows and columns for presentation purposes, there are alternatives that use CSS to achieve the desired effect. In one approach known as CSS Tables, nested **`div`**s provide the markup, and CSS Table properties make them behave like rows and cells in the browser. You can also achieve many of the effects that previously required

table markup using Flexbox and Grid Layout techniques (see **Chapter 16, CSS Layout with Flexbox and Grid**).

That said, this chapter focuses on HTML table elements used to semantically mark up rows and columns of data as described in the HTML specification.

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.

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

`<tr>…</tr>`
Table row

`<th>…</th>`
Table header

`<td>…</td>`
Table cell data

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

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.

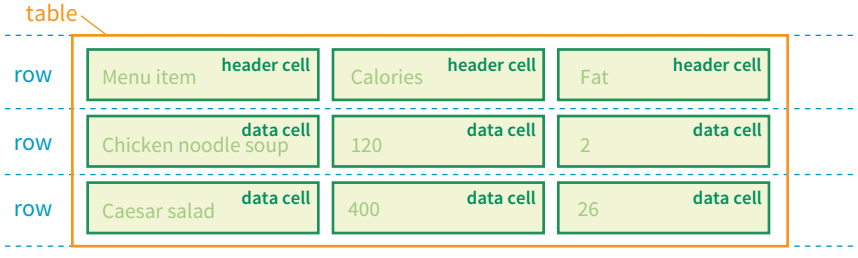


FIGURE 8-2. Tables are made up of rows that contain cells. Cells are the containers for content.

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

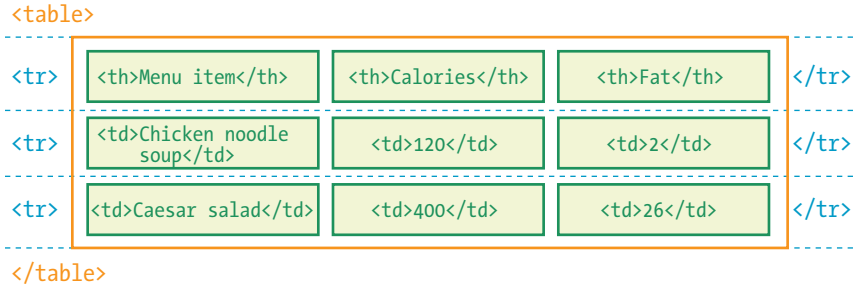


FIGURE 8-3. The elements that make up the basic structure of a table.

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. The number of columns in a table is implied 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. (There's more to the column story, which I cover in the section **"Row and Column Groups"** later in this chapter.)

Written out in a source document, the markup for the table in FIGURE 8-3 looks 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 the browser renders them.

```
<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, or even another table.

Start and end **table** tags identify the beginning and end of the tabular material. The **table** element may directly contain only some number of **tr** (row) elements, a caption and, optionally, the row and column group elements listed in the **"Row and Column Groups"** section. 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. What is worth noting is that tables always start on new lines by default in browsers.

Stylin' Tables

Once you build the structure of the table in the markup, it's no problem to add 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:

Chapter 12, Formatting Text:

- Font settings for cell contents
- Text color in cells

Chapter 13, Colors and Backgrounds:

- Background colors
- Tiling background images

Chapter 14, Thinking Inside the Box:

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

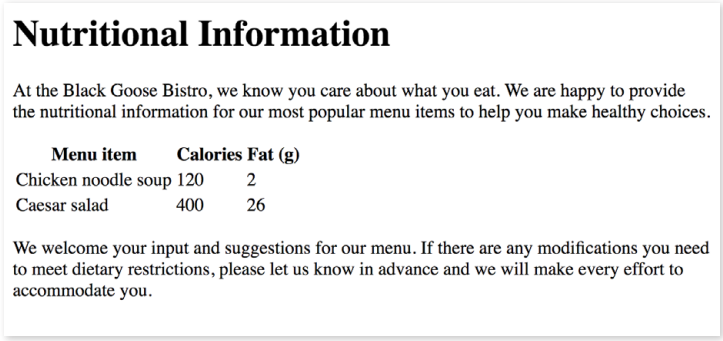
Chapter 19, More CSS Techniques:

- Special properties for controlling borders and spacing between cells

FUN WITH THE SPEC

According to the HTML5 spec, a **table** element may contain "in this order: optionally a **caption** element, followed by zero or more **colgroup** elements, followed optionally by a **thead** 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)."

Well, I'm glad we cleared that up!



Nutritional Information

At the Black Goose Bistro, we know you care about what you eat. We are happy to provide the nutritional information for our most popular menu items to help you make healthy choices.

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

We welcome your input and suggestions for our menu. If there are any modifications you need to meet dietary restrictions, please let us know in advance and we will make every effort to accommodate you.

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>
      <th>Burgers</th>
      <td>Organic Grass-fed Beef</td>
      <td>Black Bean Veggie</td>
   </tr>
   <tr>
      <th>Fries</th>
      <td>Hand-cut Idaho potato</td>
      <td>Seasoned sweet potato</td>
   </tr>
</table>
```

If you guessed that it's a table with two rows and three columns, you are correct! Two **tr** elements create two rows; one **th** and two **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, headers are a key tool for making table content accessible. Don't try to fake them 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). Instead, 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 the *materials* folder (*www.learningwebdesign.com/5e/materials*).

Note that I've added a 1-pixel border around cells with a style rule just to make the structure clear. If you would like borders on your tables, copy this **style** element into the **head** of the document(s) you create for the exercises in this chapter:

```
<style>
td, th {
  border: 1px solid gray;
}
</style>
```

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.

Album	Year
Rubber Soul	1965
Revolver	1966
Sgt. Pepper's	1967
The White Album	1968
Abbey Road	1969

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

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've added borders to reveal the cell structure in the figure, but your table won't have them unless you add the style sheet shown in EXERCISE 8-1. Again, the final markup is provided in the *materials* folder.

Some hints:

- The first and third rows show 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 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. It can also make it potentially more difficult for users with screen readers to follow.

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 the cells to reveal the structure of the table in the screenshot).

```
<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.

WARNING

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

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.

Try your hand at column spanning in EXERCISE 8-2.

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.

If you loved spanning columns, you'll love spanning rows in EXERCISE 8-3.

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.

Space in and Between Cells

By default, tables expand just enough to fit the content of the cells, which can look a little cramped. Old versions of HTML included **cellpadding** and **cellspacing** attributes for adding space within and between cells, but they have been kicked out of HTML5 as they are obsolete, presentational markup. The proper way to adjust table cell spacing is with style sheets, of course. The **"Styling Tables"** section in **Chapter 19, More CSS Techniques** addresses 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 visitors with impaired sight. It is especially challenging to make sense of tabular material by using a screen reader, but the HTML specification provides measures to improve the experience and make your content more understandable.

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.

Some hints:

- Rows always span downward, so the "oranges" cell is part of the first row even though its content is vertically centered.
- 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.

Describing Table Content

<caption>…</caption>
Title or description to be displayed with the table

The most effective way to give sight-impaired users an overview of your table is to give it a title or description with the **caption** element. Captions display next to the table (generally, above it) and can be used 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, which adds a caption to the nutritional chart from earlier in the chapter:

```
<table>
   <caption>Nutritional Information</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 8-10, although you can use a style sheet property to move it below the table (**caption-side: bottom**).

Nutritional Information

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

FIGURE 8-10. 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 (*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. The **scope** and **headers** attributes allow authors to explicitly associate headers and their respective content.

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 by using the values **row**, **col**, **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>
```

Accessibility experts recommend that every **th** element contain a **scope** attribute to make its associated data explicitly clear.

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 -->
```

Unfortunately, support of the **id**/**headers** feature is unreliable. The recommended best practice is to create tables in a way that a simple **scope** attribute will do the job.

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 (*webaim.org/techniques/tables/data*) as an excellent starting point.

There is one more important set of elements for helping make the semantic structure of a table clear: row and column grouping elements.

ROW AND COLUMN GROUPS

The sample tables we've been looking at so far in this chapter have been stripped down to their bare essentials to make the structure clear while you're learning how tables work. But tables in the real world are not always so simple. Check out the beauty in FIGURE 8-11 from the CSS Writing Modes Level 3 spec. You can identify three groups of columns (one with headers, two with two columns each), and three groupings of rows (headers, data, and a footnote).

BROWSER SUPPORT ALERT

Although the advanced table features intended to improve accessibility have been in the specs for many years, support by screen readers and other assistive devices is unreliable at best. It is still recommended that you mark up your data semantically within table cells and that they make sense when read in order from the source, which is exactly how some of your visitors may encounter them.

Conceptual table groupings like these are marked up with row group and column group elements that provide additional semantic structure and more "hooks" for styling or scripting. For example, the row and column groups in FIGURE 8-11 were styled with thicker borders to make them stand out visually.

VIEW SOURCE

View the source of the table in FIGURE 8-11 at *www.w3.org/TR/css-writing-modes-3/#unicode-bidi* (you need to scroll down a little). The source is too long to print here, but it is clearly marked up and easy to follow. Note that it uses all the row group elements, column groups, and the **scope** attribute we saw in the last section to associate headers with rows. There are several interesting tables on this page for your source-viewing pleasure.

Bidi control codes injected by 'unicode-bidi' at the start/end of 'display: inline' boxes

'unicode-bidi' value	'direction' value: 'ltr' start	'ltr' end	'rtl' start	'rtl' end
'normal'	–	–	–	–
'embed'	LRE (U+202A)	PDF (U+202C)	RLE (U+202B)	PDF (U+202C)
'isolate'	LRI (U+2066)	PDI (U+2069)	RLI (U+2067)	PDI (U+2069)
'bidi-override'*	LRO (U+202D)	PDF (U+202C)	RLO (U+202E)	PDF (U+202C)
'isolate-override'*	FSI,LRO (U+2068,U+202D)	PDF,PDI (U+202C,U+2069)	FSI,RLO (U+2068,U+202E)	PDF,PDI (U+202C,U+2069)
'plaintext'	FSI (U+2068)	PDI (U+2069)	FSI (U+2068)	PDI (U+2069)

* The LRO/RLO+PDF pairs are also applied to the root inline box of a block container if these values of 'unicode-bidi' were specified on the block container.

FIGURE 8-11. An example of a table with row and column groups (from the CSS Writing Modes Level 3 specification).

Row Group Elements

`<thead>…</thead>`
Table header row group

`<tbody>…</tbody>`
Table body row group

`<tfoot>…</tfoot>`
Table footer row group

You can describe rows or groups of rows as belonging to a header, footer, or the body of a table by using the **thead**, **tfoot**, and **tbody** elements, respectively. Some user agents (another word for a browser) may repeat the header and footer rows on tables that span multiple pages. For example, the head and foot rows may print on every page of a multipage table. Authors may also use these elements to apply styles to various regions of a table.

Row group elements may only contain one or more **tr** elements. They contain no direct text content. The **thead** element should appear first, followed by any number of **tbody** elements, followed by an optional **tfoot**.

This is the row group markup for the table in FIGURE 8-11 (**td** and **th** elements are hidden to save space):

```
<table>
…
<thead>
  <!-- headers in these rows-->
  <tr></tr>
  <tr></tr>
  <tr></tr>
<thead>
<tbody>
  <!-- data -->
  <tr></tr>
  <tr></tr>
  <tr></tr>
  <tr></tr>
  <tr></tr>
  <tr></tr>
</tbody>
```

```
<tfoot>
  <!-- footnote -->
  <tr></tr>
</tfoot>
</table>
```

Column Group Elements

As you've learned, columns are implied by the number of cells (**td** or **th**) in each row. You can semantically group columns (and assign **id** and **class** values) using the **colgroup** element.

<colgroup>…</colgroup>
A semantically related group of columns

<col>…</col>
One column in a column group

Column groups are identified at the start of the table, just after the **caption** if there is one, and they give the browser a little heads-up as to the column arrangement in the table. The number of columns a **colgroup** represents is specified with the **span** attribute. Here is the column group section at the beginning of the table in FIGURE 8-11:

```
<table>
  <caption>…</caption>
  <colgroup></colgroup>
  <colgroup span="2"></colgroup>
  <colgroup span="2"></colgroup>
  <!-- rest of table... -->
```

That's all there is to it. If you need to access individual columns within a **colgroup** for scripting or styling, identify them with **col** elements. The previous column group section could also have been written like this:

```
<colgroup></colgroup>
<colgroup>
  <col class="start">
  <col class="end">
</colgroup>
<colgroup>
  <col class="start">
  <col class="end">
</colgroup>
```

Note that the **colgroup** elements contain no content—they only provide an indication of semantically relevant column structure. The empty **col** elements are used as handles for scripts or styles, but are not required.

NOTE

When **colgroup** *elements contain* **col** *elements, they must not have a* **span** *attribute.*

WRAPPING UP TABLES

This chapter gave you a good overview of the components of HTML tables. EXERCISE 8-4 combines most of what we've covered to give you a little more practice at authoring tables.

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.

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.

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 (**DOCTYPE**, **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 clear 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>
    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. You can tell from the figure 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 by using the **caption** element.
7. Use the **scope** attribute to make sure that the Thing A, Thing B, and Thing C headers are associated with their respective rows.
8. Finally, give the table row and column groups for greater sematic clarity. There is no **tfoot** in this table. There are two column groups: one column for headers, the rest for data. Use the **span** attribute (no need for individual column identification).
9. 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 provided in the *materials* folder.

TEST YOURSELF

The answers to these questions appear in **Appendix A**.

1. What are the parts (elements) of a basic HTML table?

2. What elements can a **`table`** contain directly (i.e., first-level children)?

3. What elements can a **`tr`** contain?

4. When would you use the **`col`** (column) element?

5. 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="2">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
`tr`	Establishes a row within a table
`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
`abbr="text"`	Alternative label for when the header cell is referenced in other contexts
`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
`caption`	Gives the table a title that displays in the browser
`colgroup`	Declares a group of columns
`span="number"`	Number of columns the column group spans; may not be used when the **`colgroup`** contains **`col`** elements
`col`	Declares a column
`span="number"`	Number of columns the column spans
`tbody`	Identifies a table body row group
`thead`	Identifies a table header row group
`tfoot`	Identifies a table footer row group

CHAPTER 9

FORMS

IN THIS CHAPTER

How forms work

Elements for adding form widgets

Making forms accessible

Form design basics

It didn't take long for the web to shift from a network of pages to read to a place where you go to get things *done*—making purchases, booking plane tickets, signing petitions, searching a site, posting a tweet...the list goes on! Web forms handle all of these interactions.

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, 12 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.

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.
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 mailing list 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 saying thank you for signing up for the mailing list. Other applications might respond by reloading the 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.

A Word About Encoding

Form data is encoded via the same method used for URLs. Spaces and other characters that are not permitted get translated into their hexadecimal equivalents. For example, each space character in the collected form data is represented by the character **+** or **%20** and a slash (**/**) character is replaced with **%2F**. You don't need to worry about this; the browser handles it automatically.

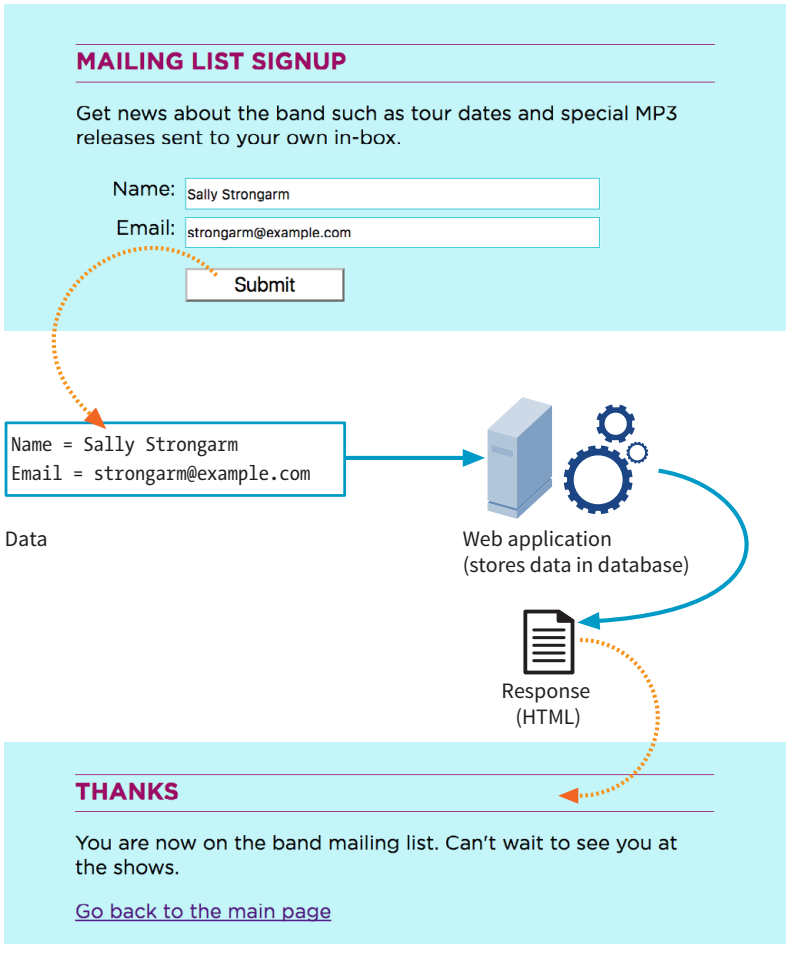


FIGURE 9-1. What happens behind the scenes when a web form is submitted.

THE FORM ELEMENT

Forms are added to web pages with (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.

<form>…</form>
Interactive form

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>
```

MARKUP TIP

Be careful not to nest **form** elements or allow them to overlap. A **form** element must be closed before the next one begins.

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 you'll need to clear out before styling them, particularly on mobile browsers. The* **fieldset**, **legend**, *and* **label** *elements used in the example improve accessibility. They are explained later in this chapter.*

```
<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="fullname" 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>
```

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 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>
```

The *.php* suffix indicates that this form is processed by a script written in the PHP scripting language, but web forms may be processed by any of the following technologies:

- PHP (*.php*) is an open source scripting language most commonly used with the Apache web server. It is the most popular and widely supported forms processing option.
- Microsoft ASP (Active Server Pages; *.asp*) is a programming environment for the Microsoft Internet Information Server (IIS).
- Microsoft's ASP.NET (Active Server Page; *.aspx*) is a newer Microsoft language that was designed to compete with PHP.
- 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 form-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 (see **Web Hosting Tip**).

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.

```
fullname = 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):

```
fullname=Sally+Strongarm&email=strongarm%40example.com
```

There are only two methods for sending this encoded data to the server: POST or GET, indicated by 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="/mailinglist.php" method="POST">...</form>
```

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/mailinglist.php?name=Sally+Strongarm&email=→
strongarm%40example.com
```

GET is inappropriate if the form submission performs an action, such as deleting something or adding data to a database, because if the user goes back, it gets submitted again.

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. In theory, only the server sees the content of this request, and thus it is the best method for sending secure information such as a home address or other personal information. In practice, make sure HTTPS is enabled on your server so the user's data is encrypted and inaccessible in transit. (HTTPS is discussed in **Chapter 2, How the Web Works**.)

WEB HOSTING TIP

If you know you want or need to work with a particular form processing language, make sure to confirm it is supported when you are shopping for a web hosting service.

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 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 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 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, let's turn our attention to 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 between options. Control types include various text-entry fields, buttons, menus, and a few controls with special functions. They are added to the document with a collection of form control elements that we'll be examining one by one in the upcoming **"The Great Form Control Roundup"** section.

As a web designer, you need 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

All form controls (except submit and reset buttons) must include a name attribute.

The job of each form control is to collect one bit of information from a user. In the previous form example, text-entry fields collect the visitor's name and email address. To use the technical term, "fullname" 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 variables.

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+is+the+best+band+ever%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 instructions provided with a ready-to-use script on your server, your system administrator, or the programmer you are working with.

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, to avoid confusion, you are advised to name each variable uniquely—that is, don't use the same name for two variables (however, there may be exceptions for which it is desirable). 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 the following:

- 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
- Numerical controls
- Color picker control

We'll pause along the way to allow you to try them out by constructing the pizza ordering form shown in FIGURE 9-2.

As you will see, the majority of controls are added to a form via the **`input`** element. The functionality and appearance of the **`input`** element changes based on the value of the **`type`** attribute in the tag. In HTML5.2, there are *twenty-two* 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.*

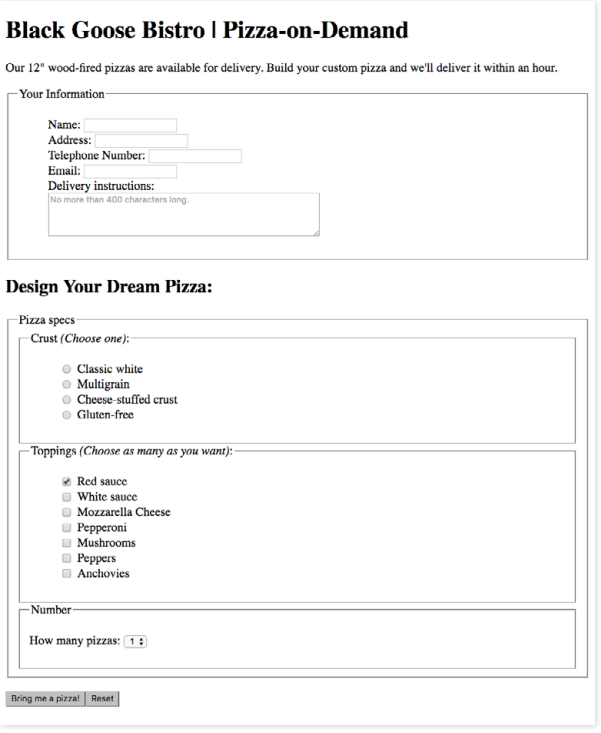


FIGURE 9-2. The pizza ordering form we'll build in the exercises in this chapter.

Text-Entry Controls

One of the most common web form tasks is entering text information. Which element you use to collect text input depends on whether users are asked to enter a single line of text (**input**) or multiple lines (**textarea**).

Be aware that if your form has text-entry fields, it needs to use the secure HTTPS protocol to protect the user-entered content while their data is in transit to the server (see the **"HTTPS, the Secure Web Protocol"** sidebar in for more information).

NOTE

The markup examples throughout this section include the **label** *element, which is used to improve accessibility. We will discuss* **label** *in the upcoming* **"Form Accessibility Features"** *section, but in the meantime, I want you to get used to seeing proper form markup.*

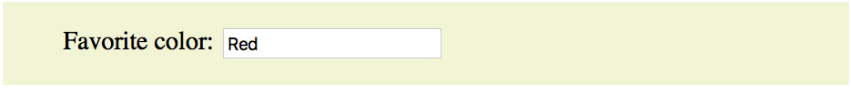
Single-line text field

`<input type="text">`
Single-line text-entry control

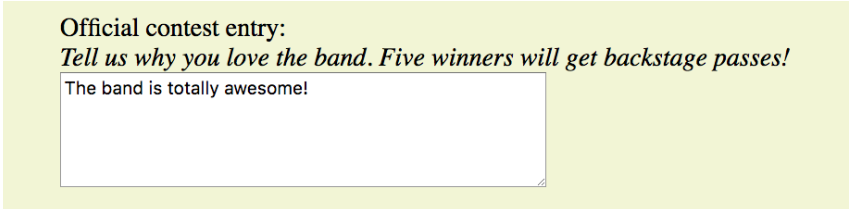
One of the most straightforward form input types is the text-entry field for entering a single word or line of text. In fact, it is the default input type, which means it is what you'll get if you forget to include the **type** attribute or include an unrecognized value. Add a text input field to a form by inserting an **input** element with its **type** attribute set to **text**, as shown here and in FIGURE 9-3:

```
<li><label>Favorite color: <input type="text" name="favcolor"
value="Red" maxlength="50"></label></li>
```

Text-entry field (`input type="text"`)



Multiline text-entry field with text content (`input type="textarea"`)



Multiline text-entry field with `placeholder` text (`input type="textarea"`)

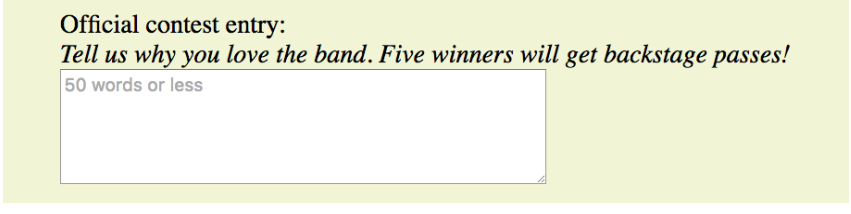


FIGURE 9-3. Examples of the text-entry control options for web forms.

There are a few attributes in there that 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. The value of the **`value`** attribute gets submitted to the server, so in this example, the value "Red" will be sent with the form unless the user changes it. As an alternative, you could use the **`placeholder`** attribute to provide a hint of what to type in the field, such as "My favorite color". The value of **`placeholder`** is not submitted with the form, and is purely a user interface enhancement. You'll see it in action in the upcoming section.

`maxlength`, `minlength`

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 form-processing program you are using requires it. The **`minlength`** attribute specifies the minimum number of characters.

NOTE

The specific rendering style of form controls varies by operating system and browser version.

BROWSER SUPPORT NOTE

Versions of Internet Explorer prior to version 11 and older versions of Android do not support **`placeholder`**.

size

The **size** attribute specifies the length of the input field in number of visible characters. It is more common, however, to use style sheets to set the size of the input area. By default, a text input widget displays at a size that accommodates 20 characters.

Multiline text-entry field

<textarea>…</textarea>
Multiline text-entry control

At times, you'll want your users to be able to enter more than just one line of text. For these instances, use the **textarea** element, which is replaced by a multiline, scrollable text entry box when displayed by the browser (FIGURE 9-3).

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 shows up in the text box when the form is displayed in the browser. It also gets sent to the server when the form is submitted, so carefully consider what goes there.

```
<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>
```

The **rows** and **cols** attributes provide a way to specify the size of the **textarea** with markup. **rows** specifies the number of lines the text area should display, and **cols** specifies the width in number of characters (although it is more common to use CSS to specify the width of the field). 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 soft line breaks (where the text naturally wraps at the edge of the box) are preserved when the form is submitted. A value of **soft** (the default) does not preserve line breaks. The **hard** value preserves line breaks when the **cols** attribute is used to set the character width of the box. The **maxlength** and **minlength** attributes set the maximum and minimum number of characters that can be typed into the field.

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 **placeholder** attribute instead. Placeholder text, unlike **textarea** content, is not sent to the server when the form is submitted. Examples of **textarea** content and placeholder text are shown in FIGURE 9-3.

```
<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" rows="5"
cols="50"></textarea>
</p>
```

disabled and readonly

The **disabled** and **readonly** attributes both prevent users from interacting with a form control, but they work slightly differently.

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 and can be applied to any form control or **fieldset**.

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 from other inputs, or they could be confusing to users who are trying to change their values. The **readonly** attribute can be used with **textarea** and text-based input controls (see TABLE 9-1 at the very end of this chapter).

The most important difference is that **readonly** fields are submitted when the form is submitted, but **disabled** ones are not.

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 by asterisk (*) or bullet (•) characters, or another character determined by the browser.

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">Password:</label><br>
 <input type="password" name="pswd" maxlength="12" id="form-pswd"></li>
```

FIGURE 9-4. Passwords are converted to bullets in the browser display.

Search, email, telephone numbers, and URLs

`<input type="search">`
Search field

`<input type="email">`
Email address

`<input type="tel">`
Telephone number

`<input type="url">`
Location (URL)

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 input types use the same attributes as the generic text input type described earlier (**name**, **maxlength**, **minlength**, **size**, and **value**), as well as a number of other attributes (see TABLE 9-1 at the end of the chapter).

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—for example, by making sure text entered in an **email** input follows the 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.

Although email, search, telephone, and URL inputs are well supported by up-to-date browsers, there may be inconsistencies in the way they are handled. Older browsers, such as Opera Mini and any version of Internet Explorer prior to 11, do not recognize them at all, but will display the default generic text input instead, which works perfectly fine.

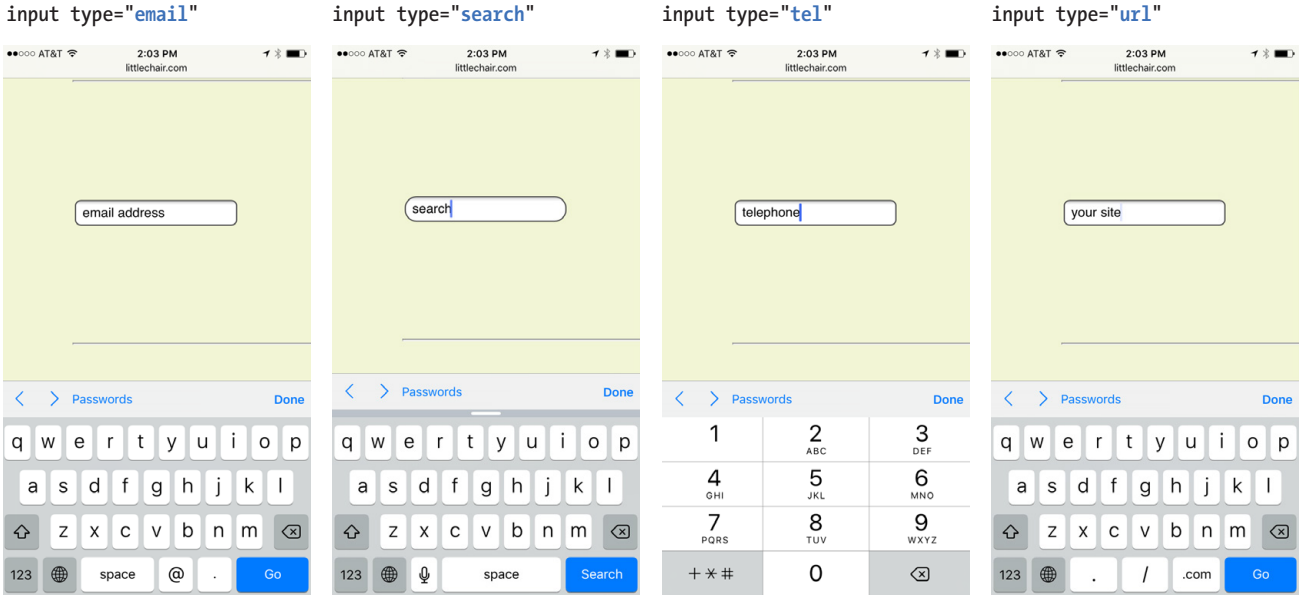


FIGURE 9-5. Safari on iOS provides custom keyboards based on the input type.

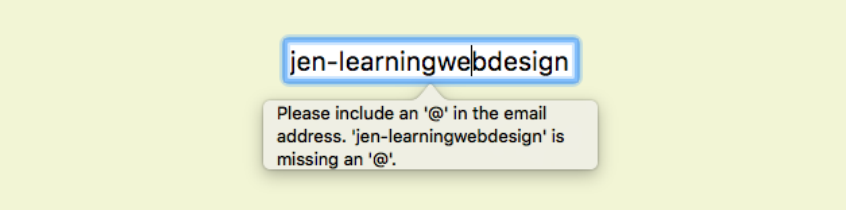


FIGURE 9-6. Opera displays a warning when input does not match the expected **email** format as part of its client-side validation support.

Drop-Down Suggestions

<datalist>…</datalist>

Drop-down menu input

The **datalist** element 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 their 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, browser support for datalists remains spotty. Chrome and Opera support it, but there is a bug that makes datalists unscrollable (i.e., unusable) if the list is too long, so it is best used for short lists of options. IE11 and Edge have buggy implementations, and Safari and iOS don't support it at all. The good news is if it is unsupported, browsers present a simple text input, which is a perfectly acceptable fallback. You could also use a JavaScript polyfill to create **datalist** functionality.

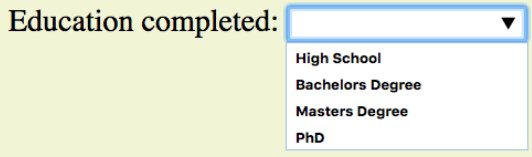


FIGURE 9-7. A **datalist** creates a pop-up menu of suggested values for a text-entry field.

WARNING

The values from form controls should be checked by the server code (PHP, ASP.NET, etc.), as they can be hacked or manipulated. So, although they make controlling and validating user input easier, it is still vital to perform server-side checks before updating the database on the server.

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 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 via 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." You can change the text on the button by 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 is because in contemporary form development, we use JavaScript to check the validity of form inputs along the way, so 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*. "When to Use the Button Element," by Chris Coyier is another helpful read (*css-tricks.com/use-button-element/*).

EXERCISE 9-1. Starting the pizza order form

Here's the scenario. You are the web designer in charge of creating an online pizza ordering form for Black Goose Bistro. The owner has handed you a sketch (FIGURE 9-9) of the form's content. 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 form. I've given you a head start by creating a bare-bones document with text content and minimal markup and styles. This document, *pizza.html*, is available online at *learningwebdesign.com/5e/materials*. The finished form is also provided.

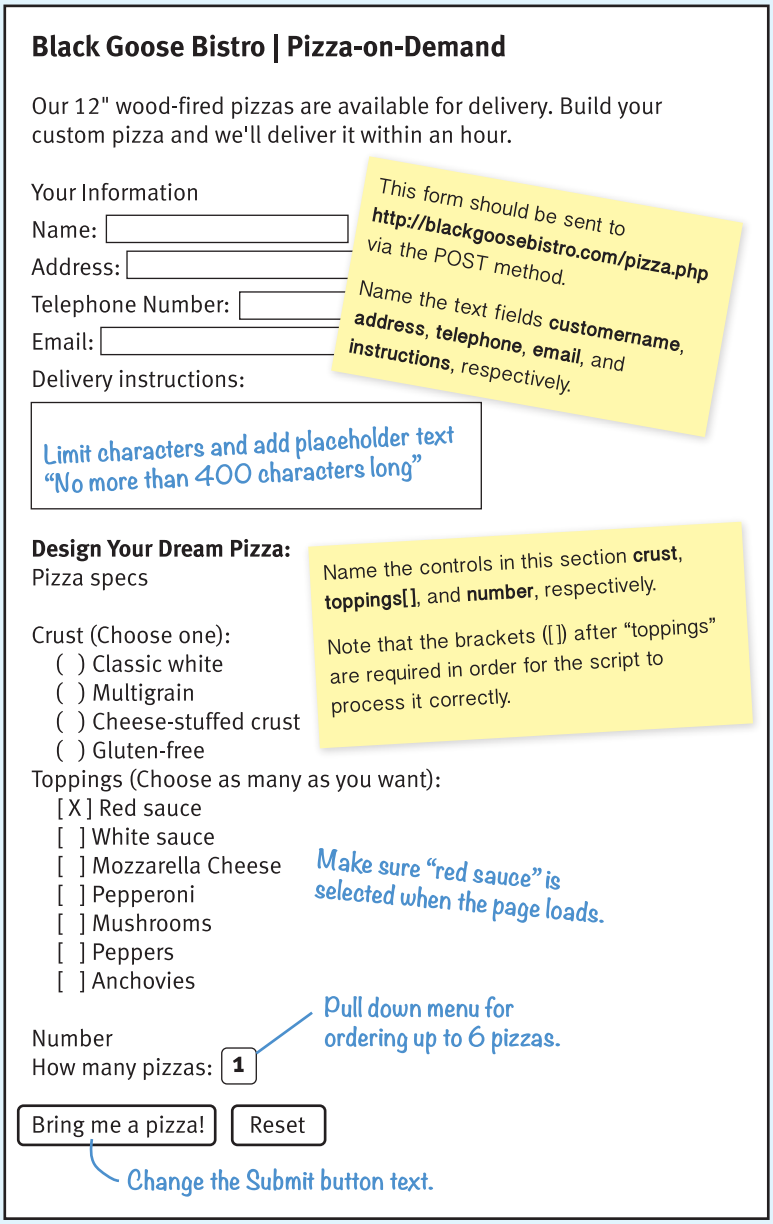


FIGURE 9-9. A sketch of the Black Goose Bistro pizza ordering form.

EXERCISE 9-1. Continued

1. Open the file *pizza.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 (keep it on one line):

```
<form action="http://www.blackgoosebistro.com/
pizza.php" method="POST">
…
</form>
```

3. In this exercise, we'll work on the "Your Information" section of the form. Start with the first four short text-entry form controls that are marked up appropriately as an unordered list. Here's the first one; you insert the other three:

```
<li>Name: <input type="text" name="customername">
</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. After "Delivery instructions:" add a line break and a multiline text area. 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>Delivery instructions:<br>
<textarea name="instructions" rows="4" cols="60"
maxlength="400" placeholder="No more than 400
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 they've asked us to change the text on the submit button.

```
<p><input type="submit" value="Bring me a
pizza!"><input type="reset"></p>
```

6. Now, save the document and open it in a browser. The parts that are finished should generally match FIGURE 9-2. If they don't, then you have some more work to do.

Once the document 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, *pizza.php* actually works, but sorry, no pizzas will be delivered.

FIGURE 9-10. You should see a response page like this if your form is working. The pizza description fields will be added in later exercises, so they will return "empty" for now.

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 with 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 "Pick-up or Delivery"). 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 OK.

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 section* **"Form Accessibility Features,"** *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 via 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 a 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.

You can decide which button is checked when the form loads by adding the **checked** attribute to the **input** element (see **Note**). In this example, the button next to "under 24" will be checked when the page loads.

Radio buttons (`input type="radio"`) Checkboxes (`input type="checkbox"`)

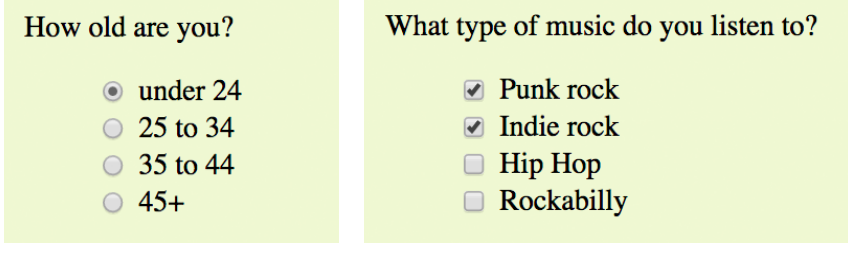


FIGURE 9-11. Radio buttons (left) are appropriate when only one selection is permitted. Checkboxes (right) are best when users may choose any number of choices, from none to all of them.

NOTE

It may look like the **checked** *attribute has no value, but it is one of the attributes in HTML that can be* minimized *to one word. Behind the scenes, the minimized* **checked** *attribute stands for the rather redundant:*

```
checked="checked"
```

One of the rules of the stricter XHTML syntax is that attributes cannot be minimized in this way.

Checkbox buttons

`<input type="checkbox">`
Checkbox button

Checkboxes are added via 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's 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 to 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>
```

Checkbox buttons also use the **checked** attribute to make them preselected when the form loads.

In EXERCISE 9-2, you'll get a chance to add both radio and checkbox buttons to the pizza ordering form.

EXERCISE 9-2. Adding radio buttons and checkboxes

The next section of the Black Goose Bistro pizza ordering form uses radio buttons and checkboxes for selecting pizza options. Open the *pizza.html* document and follow these steps:

1. In the "Design Your Dream Pizza" section, there are lists of Crust and Toppings options. The Crust options should be radio buttons because pizzas have only one crust. Insert a radio button before each option. Follow this example for the remaining crust options:

   ```
   <li><input type="radio" name="crust" value="white"> Classic white</li>
   ```

2. Mark up the Toppings options as you did the Crust options, but this time, the **type** should be **checkbox**. Be sure the variable name for each is **toppings[]**, and that the "Red sauce" option is preselected (**checked**), 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

`<select>…</select>`
Menu control

`<option>…</option>`
An option within a menu

`<optgroup>…</optgroup>`
A logical grouping of options within a menu

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.

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>
```

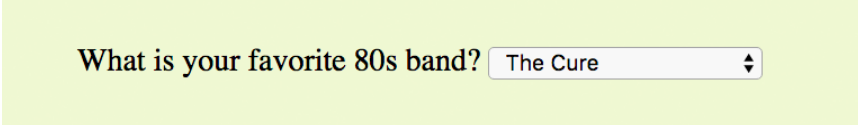


FIGURE 9-12. Pull-down menus pop open when the user clicks the arrow or bar.

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.

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):

```
<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>
```

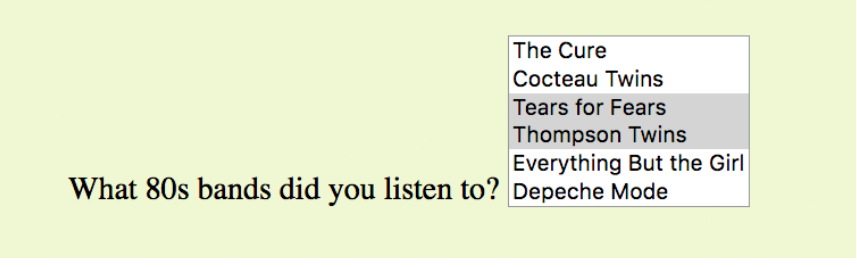


FIGURE 9-13. A scrolling menu with multiple options selected.

You may notice a few minimized 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.

Grouping menu options

You can use the **optgroup** element to create conceptual groups of options. The required **label** attribute provides the heading for the group (see **Note**). 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>
```

NOTE

The **label** *attribute in the* **optgroup** *element is not the same as the* **label** *element used to improve accessibility (discussed later in this chapter).*

FIGURE 9-14. Option groups.

In EXERCISE 9-3, you will use the **select** element to let Black Goose Bistro customers choose a number of pizzas for their order.

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 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. We add it to the form by using our old friend, the **input** element, with its **type** set to **file**.

The markup sample here (FIGURE 9-15) shows 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"></label>
</form>
```

The file upload widget varies slightly by browser and operating system, but it is generally a button that allows you to access the file organization system on your computer (FIGURE 9-15).

File input (on Chrome browser)

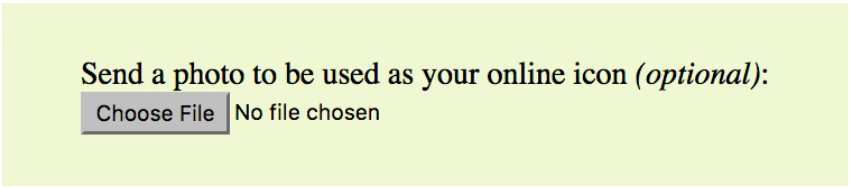


FIGURE 9-15. A file selection form field.

EXERCISE 9-3. Adding a menu

The only other control that needs to be added to the order form is a pull-down menu for selecting the number of pizzas to have delivered.

1. Insert a **select** menu element with the option to order between 1 and 6 pizzas:

```
<p>How many pizzas:
<select name="pizzas"
size="1">
    <option>1</option>
<-- 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.

It is important to note that when a form contains a file selection input element, you must specify the encoding type (**enctype**) as **multipart/form-data** in the **form** element and use the POST method.

The file input type has a few attributes. The **accept** attribute gives the browser a heads-up on what file types may be accepted (audio, video, image, or some other format identified by its media type). Adding the **multiple** attributes allows multiple files to be selected for upload. The **required** attribute, as it says, requires a file to be selected.

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.

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.

Hidden controls are added via 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/
thankyou.html">
```

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 an existing script, be sure to check the accompanying instructions to see if any hidden form variables are required.

Date and Time Controls

`<input type="date">`
Date input control

`<input type="time">`
Time input control

`<input type="datetime-local">`
Date/time control

`<input type="month">`
Specifies a month in a year

`<input type="week">`
Specifies a particular week in a year

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 with 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. As of this writing, the date and time pickers are implemented on only a few browsers (Chrome, Microsoft Edge, Opera, Vivaldi, and Android), but on non-supporting browsers, the date and time input types display as a perfectly usable text-entry field instead. FIGURE 9-16 shows date and time widgets as rendered in Chrome on macOS.

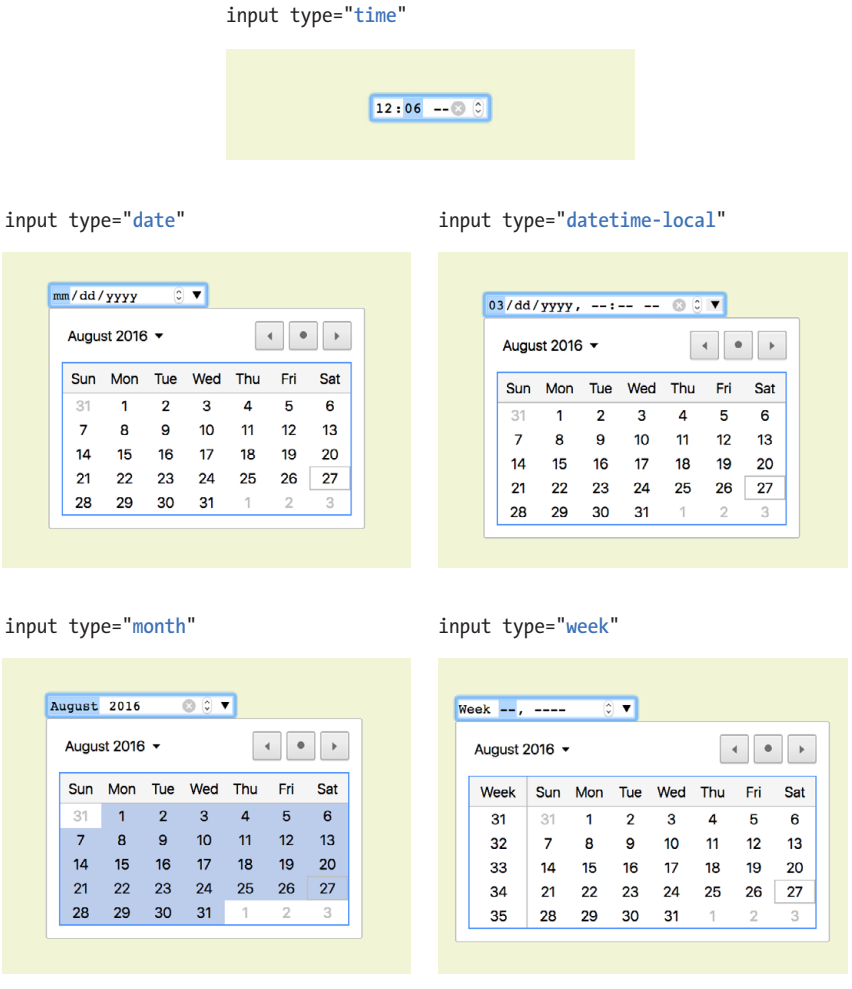


FIGURE 9-16. Date and time picker inputs (shown in Chrome on macOS).

The new date- and time-related input types are as follows:

`<input type="date" name="name" value="2017-01-14">`

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-local" name="name" value="2017-01-14T03:13:00">`

Creates a combined date/time input control with no time zone information (`YYYY-MM-DDThh:mm:ss`).

NOTE

The **value** *attribute is optional but may be included to provide a starting date or time in the widget. It is included here to demonstrate date and time formats.*

`<input type="month" name="name" value="2017-01">`

Creates a date input control that specifies a particular month in a year (`YYYY-MM`).

`<input type="week" name="name" value="2017-W2">`

Creates a date input control for specifying a particular week in a year using an ISO week numbering format (`YYYY-W#`).

Numerical Inputs

`<input type="number">`
Number input

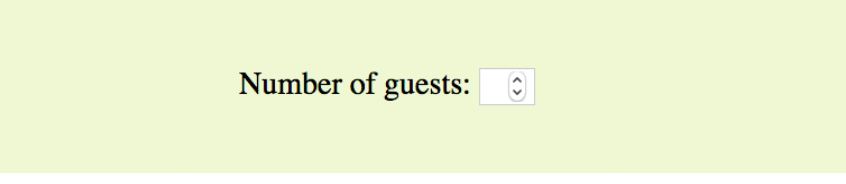
`<input type="range">`
Slider input

The **number** and **range** input types collect numerical data. For the **number** input, the browser may supply a spinner widget with up and down arrows 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="satisfaction"
min="0" max="10" step="1"></label>
```

`input type="number"`



`input type="range"`

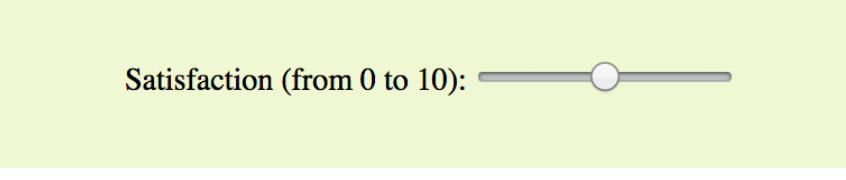


FIGURE 9-17. The **number** and **range** input types (shown in Chrome on macOS).

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. Negative values are allowed. When the element is selected, the value can be increased or decreased with the number keys on a computer keyboard, in addition to being moved with the mouse or a finger.

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, and so on; 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.

These two elements allow for only the calculated step values, not for a specified list of allowed values (such as 1, 2, 3, 5, 8, 13, 21). If you need customized values, you need to use JavaScript to program that behavior.

Because these are newer elements, browser support is inconsistent. Some UI widgets include up and down arrows for increasing or decreasing the amount, but many don't. Mobile browsers (iOS Safari, Android, Chrome for Android) currently do not support **min**, **max**, and **step**. Internet Explorer 9 and earlier do not support number and range inputs at all. Again, browsers that don't support these new input types display a standard text input field instead, which is a fine fallback.

Color Selector

`<input type="color">`
Color picker

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 in Chrome on macOS (it is the same as the macOS color picker). Non-supporting browsers—currently all versions of IE, iOS Safari, and older versions of Android—display the default text input instead.

```
<label>Your favorite color: <input type="color" name="favorite">
</label>
```

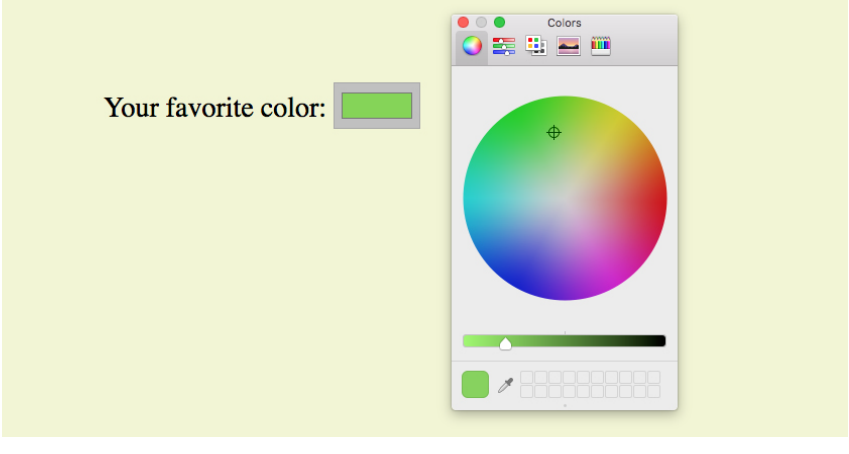


FIGURE 9-18. The **color** input type (shown in Chrome on macOS).

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 Form Elements

For the sake of completeness, let's look at the remaining form elements. These were added in HTML5 and, as of this writing, they still have spotty browser support. They 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

The **progress** element gives users feedback on the state of an ongoing process, such as a file download. It may indicate a specific percentage of completion (determinate), like a progress bar, or just indicate a "waiting" state (indeterminate), like a spinner. The **progress** element requires scripting to function.

```
Percent downloaded: <progress max="100" id="fave">0</progress>
```

meter

`<meter>…</meter>`

Represents a measurement within a range

meter 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 (they default to 0 and 100); **low** and **high** could be used to trigger warnings at undesirable levels; and **optimum** specifies a preferred value.

```
<meter min="0" max="100" name="volume">60%</meter>
```

output

`<output>…</output>`

Calculated output value

Simply put, the **output** element indicates the result of a calculation by a script or program. This example, taken from the HTML5.2 specification, uses the **output** element and JavaScript to display the sum of numbers entered into inputs a and b.

```
<form onsubmit="return false" oninput="o.value = a.valueAsNumber +
b.valueAsNumber">
<input name=a type=number step=any>
+ <input name=b type=number step=any> =
<output name=o for="a b"></output>
</form>
```

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. Not only is the resulting markup more semantically rich, but there are also more elements available to act as "hooks" for style sheet rules. Everybody wins!

Labels

`<label>…</label>`
Attaches information to form controls

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. Another advantage to using labels is that users can click or tap anywhere on them to select or focus the form control. Users with touch devices will appreciate the larger tap target.

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 you may find 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">
```

MARKUP 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-user">
<input id="form-login-passwd">
```

fieldset and legend

`<fieldset>…</fieldset>`
Groups related controls and labels

`<legend>…</legend>`
Assigns a caption to a fieldset

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 (see **Warning**):

```
<fieldset>
  <legend>Mailing List Sign-up</legend>
  <ul>
    <li><label>Add me to your mailing list <input type="radio"
    name="list" value="yes" checked></label></li>
    <li><label>No thanks <input type="radio" name="list" value="no">
  </label></li>
  </ul>
</fieldset>

<fieldset>
  <legend>Customer Information</legend>
  <ul>
    <li><label>Full name: <input type="text" name="fullname"></label>
</li>
    <li><label>Email: <input type="text" name="email"></label></li>
    <li><label>State: <input type="text" name="state"></label></li>
  </ul>
</fieldset>
```

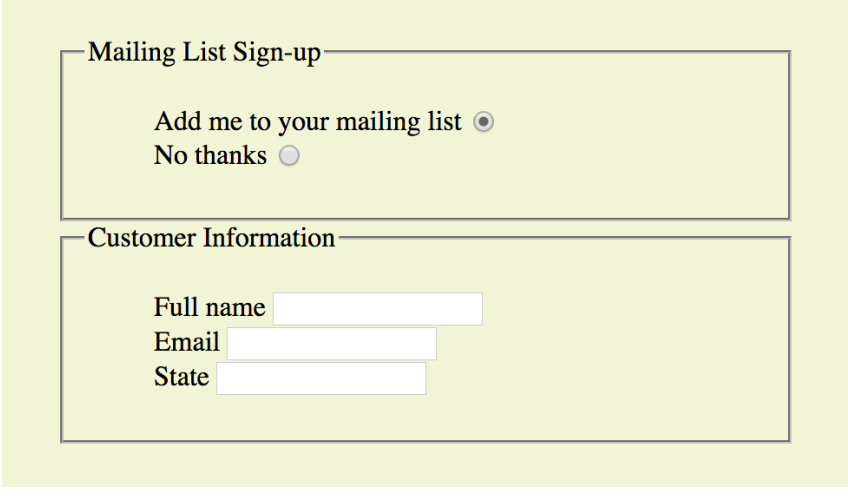


FIGURE 9-19. The default rendering of fieldsets and legends.

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.*

In EXERCISE 9-4, we'll wrap up the pizza order form by making it more accessible with labels and fieldsets.

EXERCISE 9-4. Labels and fieldsets

Our pizza ordering form is working, but we need to label it appropriately and create some **fieldsets** to make it more usable on assistive devices. Once again, open the *pizza.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 "Your Information" section 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>Your Information</legend>
  <ul>
    <li>Name: <input type="text" name="fullname">
    </li>
     …
  </ul>
</fieldset>
```

2. Next, group the Crust, Toppings, and Number questions in a big fieldset with the legend "Pizza specs" (the text is there; you just need to change it from a **p** to a **legend**):

```
<h2>Design Your Dream Pizza:</h2>
<fieldset>
<legend>Pizza specs</legend>
  Crust…
  Toppings…
  Number…
</fieldset>
```

3. Create another fieldset just for the Crust options, again changing the description in a paragraph to a **legend**. Do the same for the Toppings and Number sections. In the end, you will have three fieldsets contained within the larger "Pizza specs" fieldset. When you are done, save your document and open it in a browser. Now it should look very close to the final form shown back in FIGURE 9-2, given the expected browser differences:

```
<fieldset>
<legend>Crust <em>(Choose one)</em>:</legend>
   <ul>…</ul>
</fieldset>
```

4. OK, now let's get some labels in there. In the "Your Information" fieldset, explicitly tie the label to the text input by using the **for**/**id** label method. Wrap the description in **label** tags and add the **id** to the input. The **for**/**id** values should be descriptive and they must match. I've done the first one for you; you do the other four:

```
<li><label for="form-name">Name:</label> <input
type="text" name="fullname" 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's the first one; you do the rest:

```
<li><label><input type="radio" name="crust"
value="white"> Classic White</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.

DIY Form Widgets

Despite having dozens of form widgets straight out of HTML to choose from, it is common for developers to "roll their own" form widgets using markup, CSS, and JavaScript. This might be preferable if you want to provide custom functionality or to make the styling of the form extra-fancy. For example, you could create a drop-down menu using an unordered list inside a **div** instead of the standard **select** element:

```
<div class="select" role="listbox">
  <ul class="optionlist">
    <li class="option" role="option">Red</li>
    <li class="option" role="option">Yellow</li>
  </ul>
</div>
```

To help assistive technologies like screen readers recognize this as a form element, use the ARIA **role** attribute to describe the intended function of the **div** (a listbox) and each **li** (an option in that listbox). There are also many ARIA states and properties that make forms, both standard and custom, usable with assistive devices. For a complete list, see *www.w3.org/TR/wai-aria/states_and_properties*.

Custom form widgets require scripting and CSS well beyond the scope of this book, but I wanted you to be aware of the technique. It's also extremely easy to mess up, making a user's interaction with the form awkward and frustrating (even for sighted users), so "roll your own" with caution.

The article "How to Build Custom Form Widgets" on MDN Web Docs provides a nice overview (*developer.mozilla.org/en-US/docs/Web/Guide/HTML/Forms/How_to_build_custom_form_widgets*). You might also choose to use a premade custom widget from one of the available JavaScript Libraries like jQuery UI (*jqueryui.com*).

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) by web form expert Luke Wroblewski, and I recommend it highly. Luke's subsequent book, *Mobile First* (A Book Apart), includes tips for how to format forms in a mobile context. You can browse over a hundred articles about forms on his site at *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 as 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 the 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 you're asking for familiar information (name, 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 required information.

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 is helpful as well). Using JavaScript, you can gray out the submit button as non-functioning until all necessary data has been filled in.

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. Be sure to test in a variety of browsers to check for unpleasant surprises. **Chapter 19, More 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. You'll find the answers in **Appendix A**.

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 it?

   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" width="100" height="6">Your story.
   </textarea>
   ```

ELEMENT REVIEW: FORMS

The following table lists all of the form-related elements and attributes included in HTML 5.2 (some attributes were not covered in this chapter). The attributes for each input type are listed in TABLE 9-1.

Element and attributes	Description
`button`	Generic input button
`autofocus`	Automatically focuses the form control when the page is loaded
`name="text"`	Supplies a unique variable name for the control
`disabled`	Disables the input so it cannot be selected
`type="submit|reset|button"`	The type of custom button
`value="text"`	Specifies the value to be sent to the server
`menu="idvalue"`	Specifies a designated pop-up menu
`form,formaction, formenctype, formmethod, formnovalidate, formtarget`	Form submission-related attributes used for submit and reset type buttons
`datalist`	Provides a list of options for text inputs
`fieldset`	Groups related controls and labels
`disabled`	Disables all the inputs in the fieldset so they cannot be selected, edited, or submitted
`form="idvalue"`	Associates the element with a specific form
`name="text"`	Supplies a unique variable name for the control
`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`**
`accept-charset="characterset"`	Character encodings to use
`autocomplete`	Default setting for autofill feature for controls in the form
`name="text"`	Name of the form to use in the **`document.forms`** API
`novalidate`	Bypasses form control validation for this form
`target="text|_blank|_self|_parent|_top"`	Sets the browsing context

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-local|month|week|number|range|color"`	The type of input
See TABLE 9-1 *for a full list of attributes associated with each input type.*	
`disabled`	Disables the input so it cannot be selected, edited, or submitted
`form="form id value"`	Associates the control with a specified form
`label`	Attaches information to controls
`for="text"`	Identifies the associated control by its **`id`** reference
`legend`	Assigns a caption to a **`fieldset`**
`meter`	Represents a fractional value within a known range
`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 a 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
`output`	Represents the results of a calculation
`for="text"`	Creates a 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

Element and attributes	Description
`progress`	Represents the completion progress of a task (can be used even if the maximum value of the task is not known)
`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
`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
`autocomplete`	Hint for form autofill feature
`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 submitted
`disabled`	Disables the control so it cannot be selected
`form="form id value"`	Associates the control with a specified form
`inputmode`	Hint for selecting an input modality
`maxlength="text"`	Specifies the maximum number of characters the user can enter
`minlength="text"`	Specifies the minimum 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 the 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, while **`soft`** does not

TABLE 9-1. *Available attributes for each input type*

Attribute	submit	reset	button	text	password	checkbox	radio	image	file	hidden
accept									•	
alt								•		
autocomplete				•	•					
autofocus	•	•	•	•	•	•	•	•	•	
checked						•	•			
disabled	•	•	•	•	•	•	•	•	•	•
form	•	•	•	•	•	•	•	•	•	•
formaction	•							•		
formenctype	•							•		
formmethod	•							•		
formnovalidate	•							•		
formtarget	•							•		
height								•		
list				•						
max										
min										
maxlength				•	•				•	
minlength				•	•				•	
multiple									•	
name	•	•	•	•	•	•	•	•	•	•
pattern				•	•					
placeholder				•	•					
readonly				•	•					
required				•	•	•	•		•	
size				•	•				•	
src								•		
step										
value	•	•	•	•	•	•	•		•	•
width								•		

Attribute	email	telephone, search, url	number	range	date, time, datetime-local, month, week	color
accept						
alt						
autocomplete	•	•	•	•	•	•
autofocus	•	•	•	•	•	•
checked						
disabled	•	•	•	•	•	•
form	•	•	•	•	•	•
formaction						
formenctype						
formmethod						
formnovalidate						
formtarget						
height						
list	•	•	•	•	•	•
max			•	•	•	
min			•	•	•	
maxlength	•	•				
minlength	•	•				
multiple	•				•	
name	•	•	•	•	•	•
pattern	•	•				
placeholder	•	•				
readonly	•	•	•			
required	•	•	•		•	
size	•	•				
src						
step			•	•	•	
value	•	•	•	•	•	•
width						

CHAPTER 10

EMBEDDED MEDIA

IN THIS CHAPTER

The HTML specification defines embedded content as follows:

> *content that imports another resource into the document, or content from another vocabulary that is inserted into the document*

In **Chapter 7, Adding Images**, you saw examples of both parts of that definition because images are one type of embedded content. The **`img`** and **`picture`** elements point to an external image resource using the **`src`** or **`srcset`** attributes, and the **`svg`** element embeds an image file written in the SVG vocabulary right in the page.

But images certainly aren't the only things you can stick in a web page. In this chapter, we'll look at other types of embedded content and their respective markup, including the following:

- A window for viewing an external HTML source (**`iframe`**)
- Multipurpose embedding elements (**`object`** and **`embed`**)
- Video and audio players (**`video`** and **`audio`**)
- A scriptable drawing area that can be used for animations or game-like interactivity (**`canvas`**)

WINDOW-IN-A-WINDOW (IFRAME)

`<iframe>…</iframe>`
A nested browsing window

The **`iframe`** (short for inline frame) element lets you embed a separate HTML document or other web resource in a document. It has been around for many years, but it has recently become one of the most popular ways to share content between sites.

For example, when you request the code to embed a video from YouTube or a map from Google Maps, they provide iframe-based code to copy and paste into your page. Many other media sites are following suit because it allows them to control aspects of the content you are putting on your page. Inline frames have also become a standard tool for embedding ad content that might have been handled with Flash back in the day. Web tutorial sites may use inline frames to embed code samples on pages.

Adding an **iframe** to the page creates a little window-in-a-window (or a nested browsing context, as it is known in the spec) that displays the external resource. You place an inline frame on a page similarly to an image, specifying the source (**src**) of its content. The **width** and **height** attributes specify the dimensions of the frame. The content in the **iframe** element itself is fallback content for browsers that don't support the element, although virtually all browsers support iframes at this point.

In this very crude example, the parent document displays the web page *glossary.html* in an inline frame (FIGURE 10-1). This iframe has its own set of scrollbars because the embedded HTML document is too long to fit. To be honest, you don't often see iframes used this way in the wild (except for code examples, perhaps), but it is a good way to understand how they work.

```
<h1>An Inline Frame</h1>

<iframe src="glossary.html" width="400" height="250" >
  Read the <a href="glossary.html">glossary</a>.
</iframe>
```

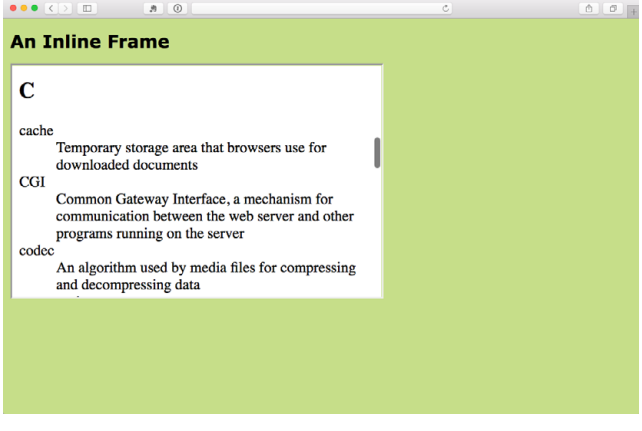


FIGURE 10-1. Inline frames (added with the **iframe** element) are like a browser window within the browser that displays external HTML documents and resources.

In modern uses of **iframe**, the window is not so obvious. In fact, there is usually no indication that there is an embedded frame there at all, as shown by the Google Maps example in FIGURE 10-2.

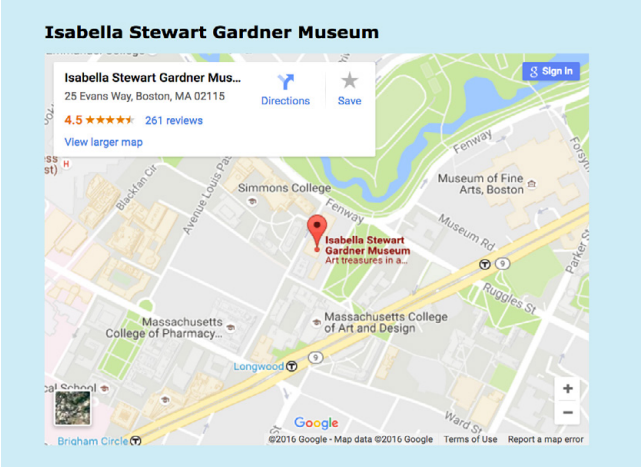


FIGURE 10-2. The edges of an **iframe** are usually not detectable, as shown in this embedded Google Map.

There are some security concerns with using iframes because they may act like open windows through which hackers can sneak. The **sandbox** attribute puts restrictions on what the framed content can do, such as not allowing forms, pop ups, scripts, and the like.

Iframe security is beyond the scope of this chapter, but you'll need to brush up if you are going to make use of iframes on your site. I recommend the MDN Web Docs article "From object to iframe: Other Embedding Technologies" (*developer.mozilla.org/en-US/docs/Learn/HTML/Multimedia_and_embedding/Other_embedding_technologies*), which provides a good overview of iframe security issues.

To get a feel for how iframes work, use one to embed your favorite video on a page in EXERCISE 10-1.

EXERCISE 10-1. Embedding a video with iframe

If you'd like to poke around with an **iframe**, it's easy to grab one from YouTube to embed your favorite video on a page.

Start by creating a new HTML document, including the basic structural elements that we covered in **Chapter 4, Creating a Simple Page**.

Go to YouTube and once you are on the page for your chosen video, look for the Share button; then choose the Embed option. The **iframe** code is there for you to copy and paste. If you click "Show more," there will be further configuration options. Just copy the **iframe** code and paste it into the new HTML document. Open it in a browser, and you're done!

MULTIPURPOSE EMBEDDER (OBJECT)

`<object>…</object>`
Represents external resource

`<param>`
Parameters of an object

In the early days, web browsers were extremely limited in what they were able to render, so they relied on plug-ins to help them display media that they couldn't handle natively. Java applets, Flash movies, RealMedia (an old web video and audio format), and other media required third-party plug-ins in order to be played in the browser. Heck, even JPEG images once required a plug-in to display.

A plug-in is software that gives a browser functionality that it doesn't have natively.

To embed those media resources on the page, we used the **object** and **embed** elements. They have slightly different uses. The **object** element is a multipurpose object placer. It can be used to place an image, create a nested browsing context (like an iframe), or embed a resource that must be handled by a plug-in. The **embed** element was for use with plug-ins only.

To put it frankly, although still in use, **object** is going out of style, and **embed** is all but extinct (I've tucked it away in a brief sidebar). Media like Java applets and Flash movies are disappearing fast, and modern browsers use APIs to display many types of media natively. In addition, mobile browsers as well as the desktop Microsoft Edge browser don't support plug-ins.

That said, let's take a look at the **object** element. At its most minimal, the **object** element uses the **data** attribute to point to the resource and the **type** attribute to provide its MIME type. Any content within the **object** element tags will be used as a fallback for browsers that don't support the embedded resource type. Here is a simple **object** element that places an SVG image on the page and provides a PNG fallback:

```
<object data="picture.svg" type="image/svg+xml">
  <img src="picture.png" alt="">
</object>
```

Additional attributes for the **object** element are available and vary according to the type of media it is placing. The media format may also require that the **object** contain a number of **param** elements that set parameters specific to that type of media.

The embed Element

The **embed** element was created by Netscape for use with plug-in technologies. It has always been well supported, but it wasn't adopted into a formal specification until HTML5. With so many other options for embedding media, the **embed** element is not as useful as it once was. It is often used as a fallback when there is a good reason to support extremely old browser versions.

embed is an empty element that points to an external resource with the **src** attribute:

```
<embed type="video/quicktime"
src="movies/hekboy.mov"
width="320" height="256">
```

There are additional media-specific attributes that set parameters similar to the **param** element, but I'm not going to cover them all here. In fact, I think that's all there is to say about **embed**.

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. We are likely to occasionally see Flash players on the desktop, but the trajectory away from plug-ins and toward standard web technologies seems clear.

In this example, **param** elements specify whether the movie starts automatically (no) or has visible controls (yes):

```
<object type="video/quicktime" data="movies/hekboy.mov" width="320"
height="256">
  <param name="autostart" value="false">
  <param name="controller" value="true">
</object>
```

VIDEO AND AUDIO

Until recently, browsers did not have built-in capabilities for handling video or sound, so they used plug-ins to fill in the gap. With the development of the web as an open standards platform, and with broadband connections allowing for heftier downloads than previously, it seemed to be time to make multimedia support part of browsers' out-of-the-box capabilities. Enter the new **video** and **audio** elements and their respective APIs (see the **"API"** sidebar).

The Good News and the Bad News

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

But if you're envisioning a perfect world where all browsers are supporting video and audio in perfect harmony, I'm afraid it's not that simple. Although they have all lined up on the markup and JavaScript for embedding 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 video or audio 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.

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 is compatible with 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.

■ TERMINOLOGY

API

An API (Application Programming Interface) is a standardized set of commands, data names, properties, actions, and so on, that lets one software application communicate with another. HTML5 introduced a number of APIs that give browsers programmable features that previously could only be achieved with third-party plug-ins.

Some APIs have a markup component, such as embedding multimedia with the new HTML5 **video** and **audio** elements (Media Player API). Others happen entirely behind the scenes with JavaScript or server-side components, such as creating web applications that work even without an internet connection (Offline Web Application API).

The W3C is 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.

A list of all HTML5 APIs and specs in development is available at *html5-overview.net*, maintained by Erik Wilde. You will also find introductions to better-known APIs in **Appendix D**.

Meet the video formats

For video, the most common options are as follows:

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. All current browsers that support HTML5 video can play MPEG-4 files with the H.264 codec. The newer H.265 codec (also known as HEVC, High Efficiency Video Coding) is in development and reduces the bitrate by half, but is not well supported as of this writing.

WebM container + VP8 video codec + Vorbis audio codec. "WebM" is a container format that has the advantage of being open source and royalty-free. It uses the *.webm* file extension. It was originally designed to work with VP8 and Vorbis codecs.

WebM container + VP9 video codec + Opus audio codec. The VP9 video codec from the WebM project offers the same video quality as VP8 and H.264 at half the bitrate. Because it is newer, it is not as well supported, but it is a great option for browsers that can play it.

Ogg container + Theora video codec + Vorbis audio codec. This is typically called "Ogg Theora," and the file should have an *.ogv* suffix. All of the codecs and the container in this option are open source and unencumbered by patents or royalty restrictions, but some say the quality is inferior to other options. In addition to new browsers, it is supported on some older versions of Chrome, Firefox, and Android that don't support WebM or MP4, so including it ensures playback for more users.

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 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 10-1 lists which browsers support the various video options (see the **"Server Setup"** sidebar).

FURTHER READING

For a thorough introduction to HTML video and audio, I recommend *Beginning HTML5 Media: Make the Most of the New Video and Audio Standards for the Web* by Silvia Pfeiffer and Tom Green (Apress).

Server Setup

In TABLES 10-1 and 10-2, the Type column identifies the MIME type of each media format. If your site is running on the Apache server, to make sure that video and audio files are served correctly, you may need to add their respective types to the server's *.htaccess* file. The following example adds the MP4 type/subtype and extensions:

```
AddType video/mp4 mp4 m4v
```

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 10-2).

MP3. The MP3 (short for MPEG-1 Audio Layer 3) format is a codec and container in one, with the file extension.*mp3*. It has become ubiquitous as a music download format.

WAV. The WAV format (*.wav*) is also a codec and container in one. This format is uncompressed so it is only good for very short clips, like sound effects.

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.

WebM container + Opus audio codec. Opus is a newer, more efficient audio codec that can be used with WebM.

FOR FURTHER EXPLORATION

HLS (HTTP Streaming Video)

If you are serious about web video, you should become familiar with HLS (HTTP Streaming Video), a streaming format that can adapt its bitrate on the fly. The HLS Wikipedia entry is as good a place as any to get started: *en.wikipedia.org/wiki/HTTP_Live_Streaming*.

TABLE 10-1. *Video support in desktop and mobile browsers (as of 2017)*

Format	Type	IE	MS Edge	Chrome	Firefox	Safari	Opera	Android	iOS Safari
MP4 (H.264)	`video/mp4 mp4 m4v`	9.0+	12+	4+	Yes*	3.2+	25+	4.4+	3.2+
WebM (VP8)	`video/webm webm webmv`	–	–	6+	4.0+	–	15+	2.3+	–
WebM (VP9)	`video/webm webm webmv`	–	14+	29+	28+	–	16+	4.4+	–
Ogg Theora	`video/ogg ogv`	–	–	3.0+	3.5+	–	13+	2.3+	–

** Firefox version varies by operating system.*

TABLE 10-2. *Audio support in current browsers (as of 2017)*

Format	Type	IE	MS Edge	Chrome	Firefox	Opera	Safari	iOS Safari	Android
MP3	`audio/mpeg mp3`	9.0+	12+	3.0+	22+	15+	4+	4.1	2.3+
WAV	`audio/wav or audio/wave`	–	12+	8.0+	3.5+	11.5+	4+	3.2+	2.3+
Ogg Vorbis	`audio/ogg ogg oga`	–	–	4.0+	3.5+	11.5+	–	–	2.3+
MPEG-4/AAC	`audio/mp4 m4a`	11.0+	12+	12.0+	–	15+	4+	4.1+	3.0+
WebM/Vorbis	`audio/webm webm`	–	–	6.0+	4.0+	11.5+	–	–	2.3.3+
WebM/Opus	`audio/webm webm`	–	14+	33+	15+	20+	–	–	–

Adding a Video to a Page

`<video>…</video>`
Adds a video player to the page

I guess it's about time we got to the markup for adding a video to a web page (this is an HTML chapter, 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 10-3 shows a movie with the default player in the Chrome browser.

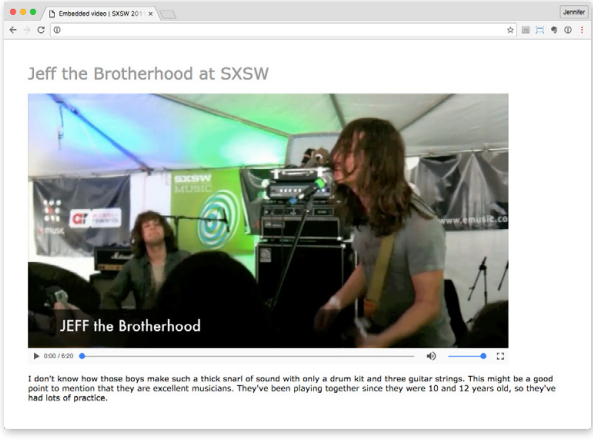


FIGURE 10-3. An embedded movie using the **`video`** element (shown in Chrome on a Mac).

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

- **Handbrake** (*handbrake.fr*) is a popular open source tool for converting to MPEG4 with H.264, H.265, VP8, and Theora. It is available for Windows, macOS, and Linux.
- **Firefogg** (*firefogg.org*) is an extension to Firefox for converting video to the WebM (VP8 and VP9) and Ogg Theora formats. Simply install the Firefogg extension to Firefox (cross-platform); then visit the Firefogg site and convert video by using its online interface.
- **FFmpeg** (*ffmpeg.org*)is an open source, command-line tool for converting just about any video format. If you are not comfortable with the command line, there are a number of software packages (some for pay, some free) that offer a user interface to FFmpeg to make it more user-friendly.
- **Freemake** (*freemake.com*) is a free video and audio conversion tool for Windows that supports over 500 media formats.

Audio conversion

- **Audio Converter** (*online-audio-converter.com*) is one of the free audio and video tools from *123Apps.com* that converts files to MP3, WAV, OGG, and more.
- **Media.io** (*media.io*) is a free web service that converts audio to MP3, WAV, and OGG.
- **MediaHuman Audio Converter** (*www.mediahuman.com/audio-converter/*) is free for Mac and Windows and can convert to all of the audio formats listed in this chapter and more. It has an easy drag-and-drop interface, but is pretty much no-frills.
- **Max** (*sbooth.org/Max/*) is an open source audio converter (Mac only).
- **Audacity** (*www.audacityteam.org*) is free, open source, cross-platform audio software for multitrack recording and editing. It can import and export files in many of the formats listed in this chapter.

Here is a simple video element that embeds a movie and player on a web page:

```
<video src="highlight_reel.mp4" width="640" height="480"
poster="highlight_still.jpg" controls autoplay>
  Your browser does not support HTML5 video. Get the <a
href="highlight_reel.mp4">MP4 video</a>
</video>
```

Browsers that do not support **video** display whatever content is provided within the **video** element. In this example, it provides a link to the movie that your visitor could download and play in another player.

There are also some 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 an image that is shown 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 create your own custom player interface using CSS and JavaScript if you want more consistency across browsers.

autoplay

Makes the video start playing automatically after 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. **autoplay** does not work on iOS Safari and some other mobile browsers in order to protect users from unnecessary data downloads.

In addition, the **video** element can use the **loop** attribute to make the video play again after it has finished (ad infinitum), **muted** for playing the video track without the audio, 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 clicks 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**.

Providing video format options

Do you remember back in **Chapter 7** when we supplied multiple image formats with the **picture** element using a number of **source** elements? Well, **picture** got that idea from **video**!

As you've seen, it is not easy to find one video format to please all browsers (although MPEG4/H.264 gets close). In addition, new efficient video formats like VP9 and H.265 are available but not supported in older browsers. Using **source** elements, we can let the browsers use what they can.

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 following example provides a video clip in the souped-up WebM/VP9 format for supporing browsers, as well as an MP4 and Ogg Theora for other browsers. This will cover pretty much all browsers that support HTML5 video (see the sidebar **"Flash Video Fallback"**).

```
<video id="video" controls poster="img/poster.jpg">
  <source src="clip.webm" type="video/webm">
  <source src="clip.mp4" type="video/mp4">
  <source src="clip.ogg" type="video/ogg">
  <a href="clip.mp4">Download the MP4 of the clip.</a>
</video>;
```

Custom video players

One of the powerful things about the **video** element and the Media Player API is that the system allows for a lot of customization. You can change the appearance of the control buttons with CSS and manipulate the functionality with JavaScript. That is all beyond the scope of this chapter, but I recommend the article "Creating a Cross-Browser Video Player" by Eric Shepherd, Chris Mills, and Ian Devlin (*developer.mozilla.org/en-US/Apps/Fundamentals/Audio_and_video_delivery/cross_browser_video_player*) for a good overview.

You may also be interested in trying out a prefab video player that provides good looks and advanced performance such as support for streaming video formats. You can implement many of them by adding a line or two of JavaScript to your document and then by using the **video** element, so it's not hard to get started. There's a nice roundup of plug-and-play video player options listed at VideoSWS (*videosws.praegnanz.de/*).

Flash Video Fallback

Older browsers—most notably Internet Explorer versions 8 and earlier—do not support **video**. If f IE8 is making a significant blip in your site statistics, you may choose to provide a Flash movie fallback. The "Creating a Cross-Browser Video Player" article mentioned previously has thorough explanation of the technique. Another article worth a read is Kroc Camen's "Video for Everybody" (*camendesign.com/code/video_for_everybody*). It is a bit dated, but I'm sure would be helpful, balanced with your up-to-date browser support knowledge.

Adding Audio to a Page

`<audio>…</audio>`
Adds an audio file to the page

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. FIGURE 10-4 shows how the audio player might look when it's rendered in the browser.

```
<p>Play "Percussion Gun" by White Rabbits</p>

<audio id="whiterabbits" controls preload="auto">
  <source src="percussiongun.mp3" type="audio/mp3">
  <source src="percussiongun.ogg" type="audio/ogg">
  <source src="percussiongun.webm" type="audio/webm">
  <p>Download "Percussion Gun":</p>
  <ul>
    <li><a href="percussiongun.mp3">MP3</a></li>
    <li><a href="percussiongun.ogg">Ogg Vorbis</a></li>
  </ul>
</audio>
```

FIGURE 10-4. Audio player as rendered in Firefox.

If you have only one audio file, you can simply use the **src** attribute instead. 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="jetfighter.mp3" autoplay loop></audio>
```

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

Adding Text Tracks

`<track>…</track>`
Adds synchronized text to embedded media

The **track** element provides a way to add text that is synchronized with the timeline of a video or audio track. Some uses include the following:

- **Subtitles** in alternative languages
- **Captions** for the hearing impaired
- **Descriptions** of what is happening in a video for the sight impaired
- **Chapter titles** to allow for navigation through the media
- **Metadata** that is not displayed but can be used by scripts

Clearly, adding text tracks makes the media more accessible, but it has the added bonus of improving SEO (Search Engine Optimization). It can also allow for deep linking, linking to a particular spot within the media's timeline.

FIGURE 10-5 shows how captions might be rendered in a browser that supports the **track** element.

FIGURE 10-5. A video with captions.

Use the **track** element inside the **video** or **audio** element you wish to annotate. The **track** element must appear after all the **source** elements, if any, and may include these attributes:

src

Points to the text file.

kind

Specifies the type of text annotation you are providing (**subtitles**, **captions**, **descriptions**, **chapters**, or **metadata**). If **kind** is set to **subtitle**, you must also specify the language (**srclang** attribute) by using a standardized IANA two-letter language tag (see **Note**).

label

Provides a name for the track that can be used in the interface for selecting a particular track.

default

Marks a particular track as the default and it may be used on only one track within a media element.

NOTE

The full list of two-letter language codes is published at www.iana.org/assignments/language-subtag-registry/language-subtag-registry.

The following code provides English and French subtitle options for a movie:

```
<video width="640" height="320" controls>
  <source src="japanese_movie.mp4" type="video/mp4">
  <source src="japanese_movie.webm" type="video/webm">
  <track src="english_subtitles.vtt"
         kind="subtitles"
         srclang="en"
         label="English subtitles"
         default>
  <track src="french.vtt"
         kind="subtitles"
         srclang="fr"
         label="Sous-titres en français">
</video>
```

WebVTT

You'll notice in the previous example that the track points to a file with a *.vtt* suffix. That is a text file in the WebVTT (Web Video Text Tracks) format that contains a list of cues. It looks like this:

```
WEBVTT

00:00:01.345 --> 00:00:03.456
Welcome to Artifact [applause]

00:00:06.289 --> 00:00:09.066
There is a lot of new mobile technology to discuss.

00:00:06.289 --> 00:00:13.049
We're glad you could all join us at the Alamo Drafthouse.
```

Cues are separated by empty line spaces. Each cue has a start and end time in *hours:minutes:seconds:milliseconds* format, separated by an "arrow" (`-->`). The cue text (subtitle, caption, description, chapter, or metadata) is on a line below. Optionally, an ID can be provided for each cue on the line above the time sequence.

You can probably guess that there's a lot more to mastering text tracks for video and audio. Take a look at the following resources:

- "Adding Captions and Subtitles to HTML5 Video" at MDN Web Docs (*developer.mozilla.org/en-US/Apps/Fundamentals/Audio_and_video_delivery/Adding_captions_and_subtitles_to_HTML5_video*)
- Subtitle tutorial on Miracle Tutorials (*www.miracletutorials.com/how-to-create-captionssubtitles-for-video-and-audio-in-webtvv-srt-dfxp-format/*)
- The WebVTT specification at the W3C is available at *www.w3.org/TR/webvtt1/*

If you'd like to play around with the **`video`** element, spend some time with EXERCISE 10-2.

NOTE

Other timed text formats include SRT captioning (replaced by WebVTT) and TML/DFXP, which is maintained by the W3C and supported by Internet Explorer but it is not recommended in the HTML5 specification for **`track`**.

EXERCISE 10-2. Embedding a video player

In this exercise, you'll add a video to a page with the **video** element. In the materials for **Chapter 10**, you will find the small movie about wind tunnel testing in MPEG-4, OGG/Theora, and WebM formats.

1. Create a new document with the proper HTML5 setup, or you can use the same document you used in EXERCISE 10-1.
2. Start by adding the **video** element with the **src** attribute pointed to *windtunnel.mp4* because MP4 video has the best browser support. Be sure to include the width (320 pixels) and height (262 pixels), as well as the **controls** attribute so you'll have a way to play and pause it. Include some fallback copy within the **video** element—either a message or a link to the video:

```
<video src="windtunnel.mp4" width="320"
height="262" controls>
  Sorry, your browser doesn't support HTML5 video.
</video>
```

3. Save and view the document in your browser. If you see the fallback message, your browser is old and doesn't support the **video** element. If you see the controls but no video, it doesn't support MP4, so try it again with one of the other formats.
4. The **video** element is pretty straightforward so you may feel done at this point, but I encourage you to play around with it a little to see what happens. Here are some things to try:
   - Resize the video player with the **width** and **height** attributes.
   - Add the **autoplay** attribute.
   - Remove the **controls** attribute and see what that's like as a user.
   - Rewrite the **video** element using **source** elements for each of the three provided video formats.

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 for drawing with 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 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.

It is worth noting that the canvas drawing area is raster-based, meaning that it is made up of a grid of pixels. This sets it apart from the other drawing standard, SVG, which uses vector shapes and paths that are defined with points and mathematics.

The good news is that every current browser supports the **canvas** element as of this writing, with the exception of Internet Explorer 8 and earlier (see **Note**). It has become so well established that Adobe's Animate software (the replacement for Flash Pro) now exports to canvas format.

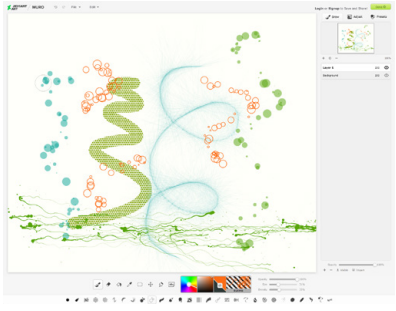
FIGURE 10-6 shows a few examples of the **canvas** element used to create games, drawing programs, an interactive molecule structure tool, and an

NOTE

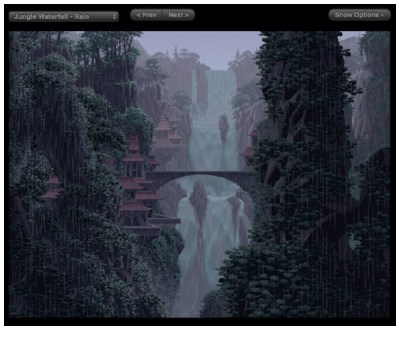
If you have a good reason to support IE8, the FlashCanvas JavaScript library (flashcanvas.net) adds canvas support using the Flash drawing API.

asteroid animation. You can find more examples at EnvatoTuts+ (*code.tutsplus.com/articles/21-ridiculously-impressive-html5-canvas-experiments--net-14210*), on David Walsh's blog (*davidwalsh.name/canvas-demos*), as well as the results of your own web search.

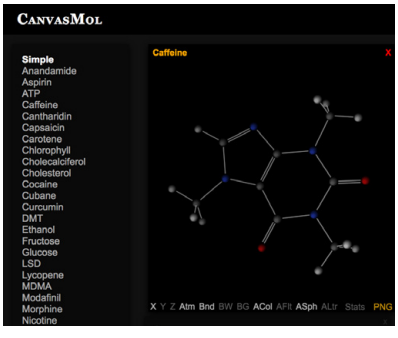
mahjong.frvr.com/



muro.deviantart.com/



www.effectgames.com/demos/canvascycle/



alteredqualia.com/canvasmol/

FIGURE 10-6. 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

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>…</canvas>
Adds a 2-D dynamic drawing area

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

The markup just clears a space upon which the drawing will happen. You can affect the drawing space itself with CSS (add a border or a background color, for example), but all of the contents of the canvas are generated by scripting and cannot be selected for styling with CSS.

Drawing with JavaScript

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

Sanders Kleinfeld created the following code example for his book *HTML5 for Publishers* (O'Reilly). He was kind enough to allow me to use it in this book. FIGURE 10-7 shows the simple smiley face we'll be creating with the Canvas API.

FIGURE 10-7. The finished product of our "Hello Canvas" example. See the original at *examples.oreilly.com/0636920022473/my_first_canvas/my_first_canvas.html*.

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 inline comments. 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 *www.w3.org/TR/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, Second Edition*, by Steve Fulton and Jeff Fulton (O'Reilly).
- If video is more your speed, try this tutorial by David Geary: *HTML5 Canvas for Developers* (*shop.oreilly.com/product/0636920030751.do*).

TEST YOURSELF

We've looked at all sorts of ways to stick things in web pages in this chapter. We've seen how to use **`iframe`** to create a "window-in-a-window" for displaying external web resources; **`object`** for resources that require plug-ins, video and audio players; and the **`canvas`** 2-D scriptable drawing space. Now see if you were paying attention. As always, answers are in **Appendix A**.

1. What is a "nested browsing context," and how would you create one?

2. Why would you use the **`sandbox`** attribute with an **`iframe`**?

3. Name some instances when you might need to know the MIME type for your media file.

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

   a. Ogg _____
   b. H.264 _____
   c. VP8 _____
   d. Vorbis _____
   e. WebM _____
   f. Theora _____
   g. Opus _____
   h. MPEG-4 _____

5. What does the **`poster`** attribute do?

6. What is a *.vtt* file?

7. List at least two differences between SVG and Canvas.

8. List the two Canvas API functions you would use to draw a rectangle and fill it with red. You don't need to write the whole script.

ELEMENT REVIEW: EMBEDDED MEDIA

The following elements are used to embed media files of many types into web pages.

Element and Attributes	Description
`audio`	Embeds an audio player on the page
`src="URL"`	Address of the resource
`crossorigin="anonymous|use-credentials"`	How the element handles requests from other origins (servers)
`preload="auto|none|metadata"`	Indicates how much the media resource should be buffered on page load
`autoplay`	Indicates the media can play as soon as the page is loaded
`loop`	Indicates the media file should start playing again automatically once it reaches the end
`muted`	Disables the audio output
`controls`	Indicates the browser should display a set of playback controls for the media file
`canvas`	Represents a two-dimensional area that can be used for rendering dynamic bitmap graphics
`height`	The height of the canvas area
`width`	The width of the canvas area
`embed`	Embeds a multimedia object that requires a plug-in for playback on the page. Certain media types require custom attributes not listed below.
`src="URL"`	Address of the media resource
`type="media type"`	The media (MIME) type of the media
`width="number"`	The horizontal dimension of the video player in pixels
`height="number"`	The vertical dimension of the video player in pixels
`iframe`	Creates a nested browsing context to display HTML resources in a page
`src="URL"`	Address of the HTML resource
`srcdoc="HTML source code"`	The HTML source of a document to display in the inline frame
`name="text"`	Assigns a name to the inline frame to be referenced by targeted links
`sandbox="allow-forms|allow-pointer-lock|allow-popups|allow-same-origin|allow-scripts|allow-top-navigation"`	Security rules for nested content
`allowfullscreen`	Indicates the objects in the inline frame are allowed to use **`requestFullScreen()`**
`width="number"`	The horizontal dimension of the video player in pixels
`height="number"`	The vertical dimension of the video player in pixels

Element and Attributes	Description
`object`	A generic element for embedding an external resource
`data="URI"`	Address of the resource
`type="media type"`	The media (MIME) type of the resource
`typemustmatch`	Indicates the resource is to be used only if the value of the **`type`** attribute and the content type of the resource match
`name="text"`	The name of the object to be referenced by scripts
`form="form ID"`	Associates the **`object`** with a **`form`** element
`width="number"`	The horizontal dimension of the video player in pixels
`height="number"`	The vertical dimension of the video player in pixels
`param`	Supplies a parameter within an **`object`** element
`name="text"`	Defines the name of the parameter
`value="text"`	Defines the value of the parameter
`source`	Allows authors to specify multiple versions of a media file (used with **`video`** and **`audio`**)
`src="text"`	The address of the resource
`type="media type"`	The media (MIME) type of the resource
`track`	Specifies an external resource (text or audio) that is timed with a media file that improves accessibility, navigation, or SEO
`kind="subtitles|captions|descriptions|chapters|metadata"`	Type of text track
`src="text"`	Address of external resource
`srclang="valid language tag"`	Language of the text track
`label="text"`	A title for the track that may be displayed by the browser
`default`	Indicates the track should be used by default if it does not override user preferences
`video`	Embeds a video player on the page
`src="URL"`	Address of the resource
`crossorigin="anonymous|use-credentials"`	How the element handles requests from other origins (servers)
`poster="URL"`	The location of an image file that displays as a placeholder before the video begins to play
`preload="auto|none|metadata"`	Hints how much buffering the media resource will need
`autoplay`	Indicates the media can play as soon as the page is loaded
`loop`	Indicates the media file should start playing again automatically once it reaches the end
`muted`	Disables the audio output
`controls`	Indicates the browser should display a set of playback controls for the media file
`width="number"`	Specifies the horizontal dimension of the video player in pixels
`height="number"`	Specifies the vertical dimension of the video player in pixels