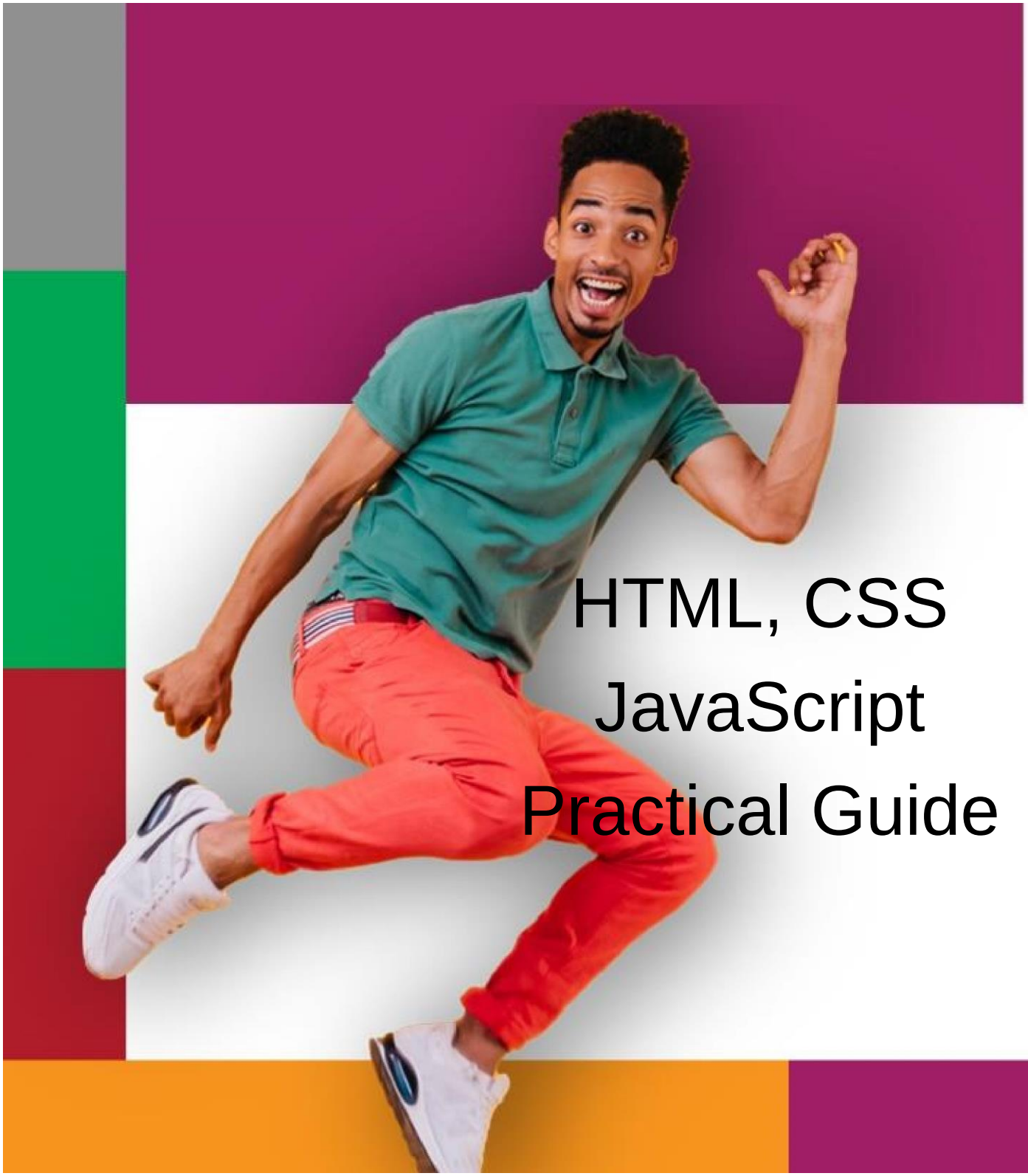


HTML, CSS JavaScript Practical Guide

Contents

1. HTML (Hypertext Markup Language)

Content so far provided a general overview of the web design environment. Now that we have covered the big concepts, it is 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 section, we will 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 section:

- 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.

Pay attention to the process, the overall structure of the document, and the new terminology.

1.1 Creating a simple page

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

Step 2: Give the document structure. You will learn about HTML element syntax and the elements that set up areas for content and metadata.

Step 3: Identify text elements. You will 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 will 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 are finished, you will have written the document for the page shown in FIGURE 1-1. It is not very fancy, but you have to start somewhere.

FIGURE 1-1. In this section, we will write the HTML document for this page in five steps.

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

Launch the text editor

In this chapter and throughout the book, we will 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.

This section shows how to open new documents in Notepad and TextEdit. Even if you have used these programs before, skim through for some special settings that will make the exercises go more smoothly. We will 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 1-2):

I. Search for “Notepad” to access it quickly. Click on Notepad to open a new document window, and you are ready to start typing.
II. 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.

III. Find “Hide extensions for known file types” and uncheck that option.

IV. Click OK to save the preference 4, and the file extensions will now be visible.

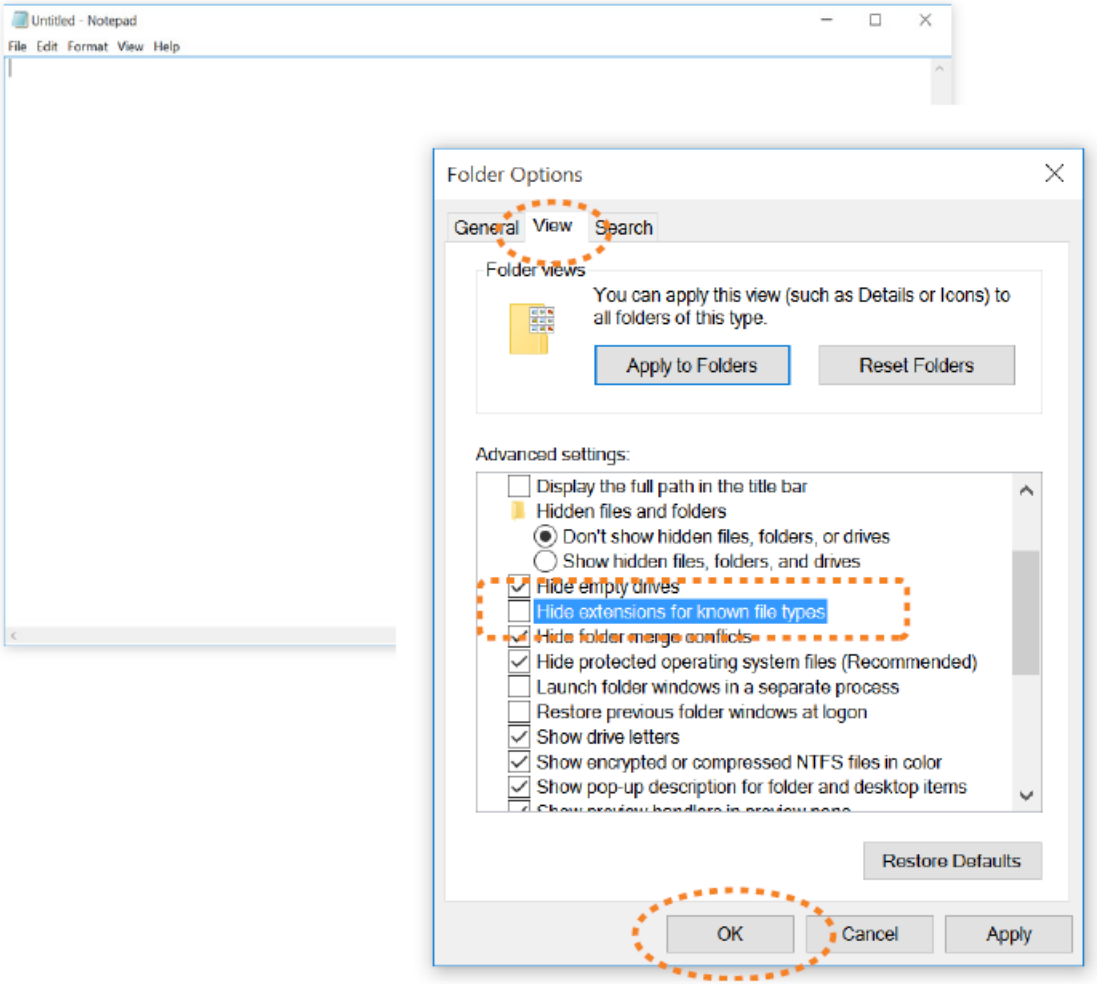


FIGURE 1-2. Creating a new document in Notepad.

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 will need to change the format, as shown in this example (FIGURE 1-3):

I. Use the Finder to look in the Applications folder for TextEdit. When you have found it, double-click the name or icon to launch the application.

II. 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. If you do not, you are in plain-text mode. Either way, there are some preferences you need to set.

III. Close that document, and open the Preferences dialog box from the TextEdit menu.

IV. Change these preferences: On the New Document tab, select Plain text. Under Options, deselect all of the automatic formatting options. On the Open and Save tab, select Display HTML files as HTML Code 5 and deselect “Add ‘.txt’ extensions to plain text files”. The rest of the defaults should be fine.

V. When you are done, click the red button in the top left corner.

VI. 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.

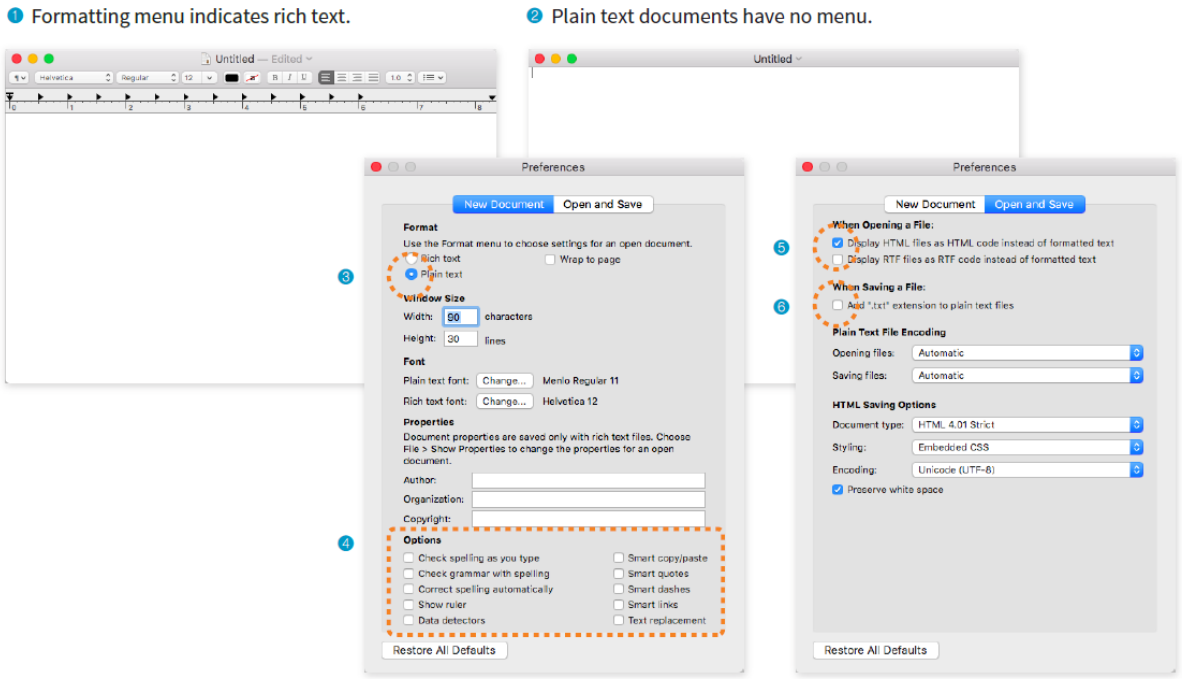


FIGURE 1-3. Launching TextEdit and choosing “Plain text” settings in the Preferences.

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 is where we begin our demonstration. EXERCISE 1-1 walks you through entering the raw text content and saving the document in a new folder.

EXERCISE 1-1. 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.

   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 1-4).
   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.

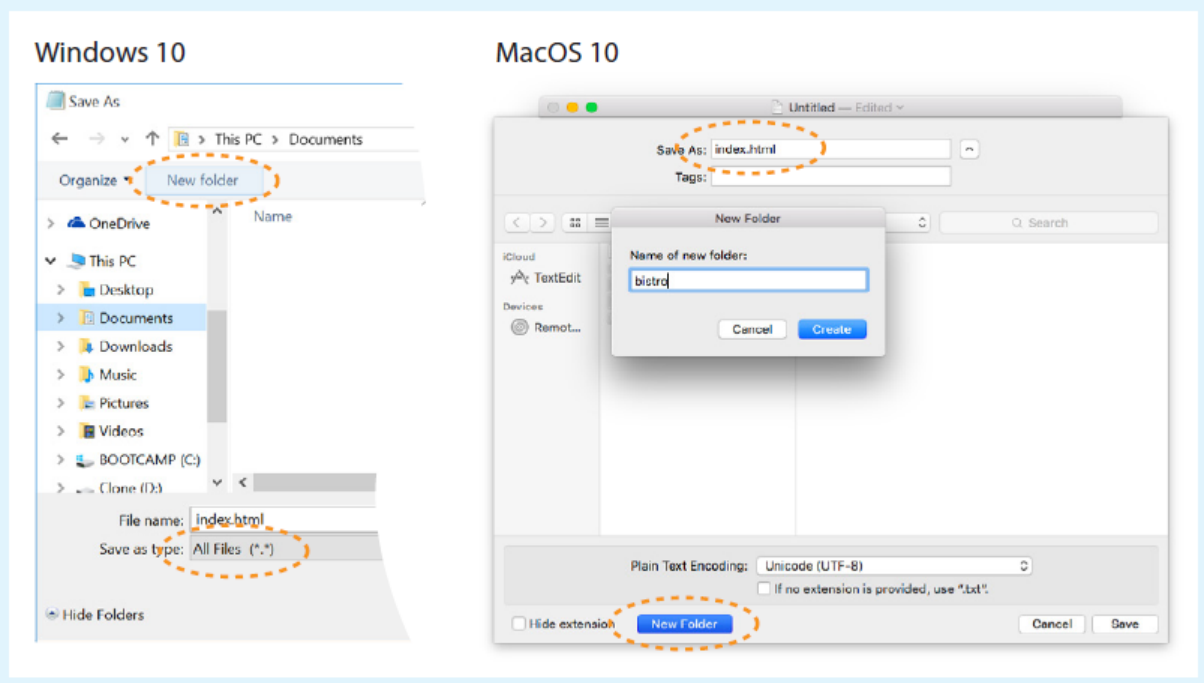


FIGURE 1-4. Saving index.html in a new folder called bistro.

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 us 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 am 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 1-5. We will talk about the results in the following section.

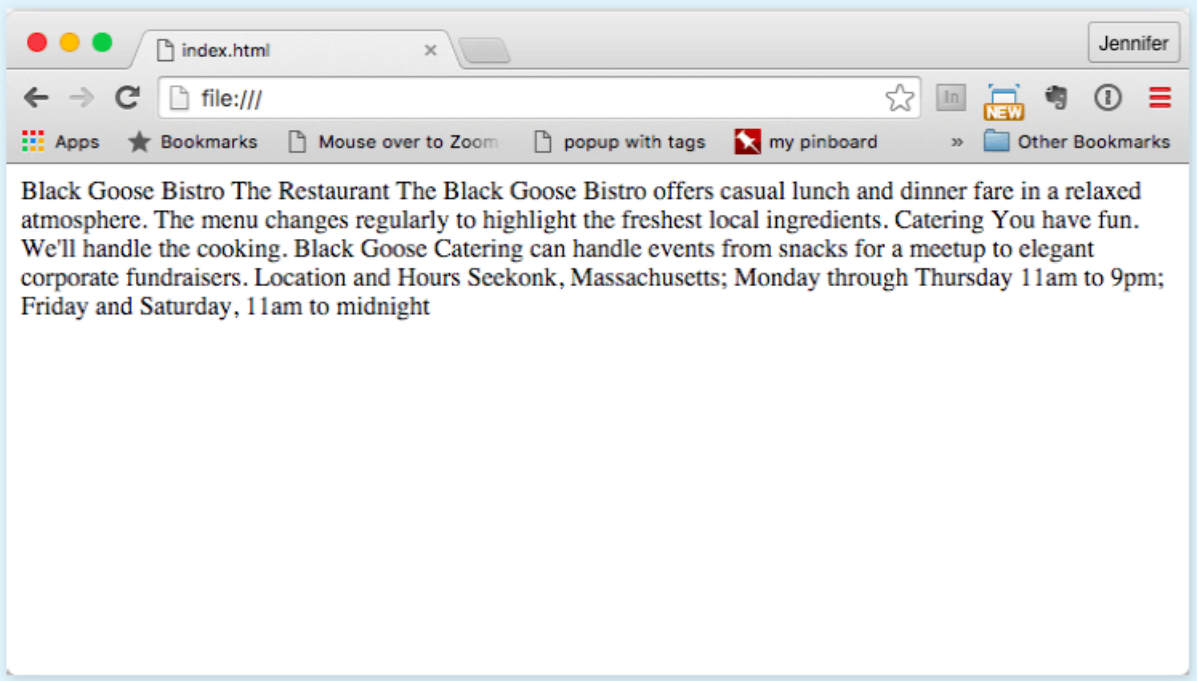


FIGURE 1-5. A first look at the content in a browser.

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 is not looking so good (FIGURE 1-5). The text is all run together into one block—that is 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 have not indicated

the structure of the content. That is where HTML comes in. We will 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

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 1-6.

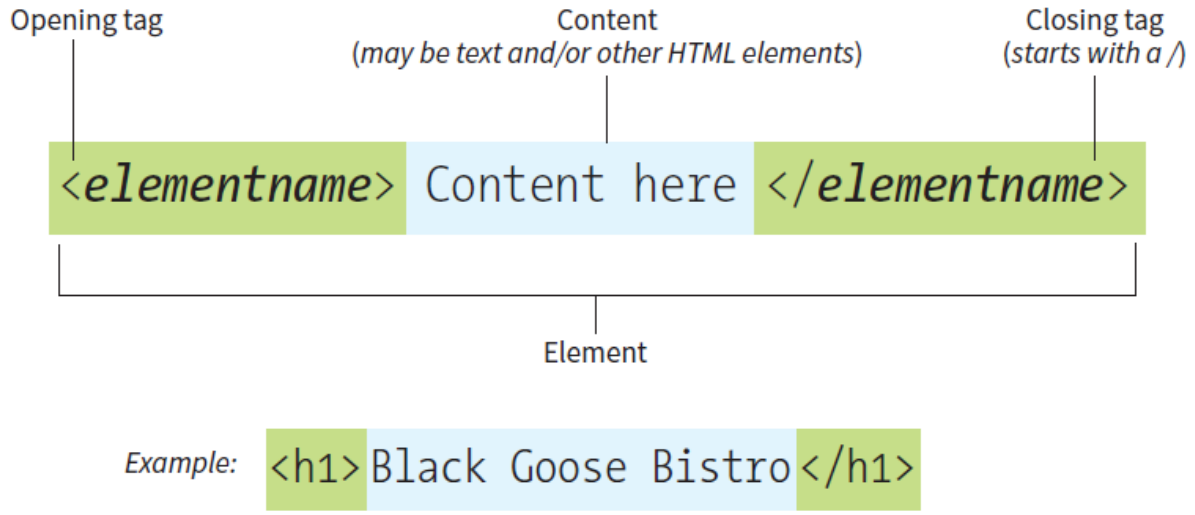


FIGURE 1-6. The parts of an HTML container element.

Tags in the text source identify elements. 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") in figure 1-7.

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 will talk about empty elements a little later in this section.

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.

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.

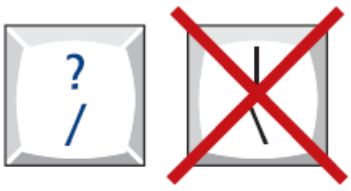


FIGURE 4-7. Slash versus backslash keys.

FIGURE 1-7. Slash versus backslash keys.

Basic Document Structure

FIGURE 1-8 shows the recommended minimal skeleton of an HTML document.

I say, “recommended” because the only element that is required in HTML is the title. However, I feel it is better, particularly for beginners, to explicitly organize documents into metadata (head) and content (body) areas. Let us take a look at what’s going on in this minimal markup example.

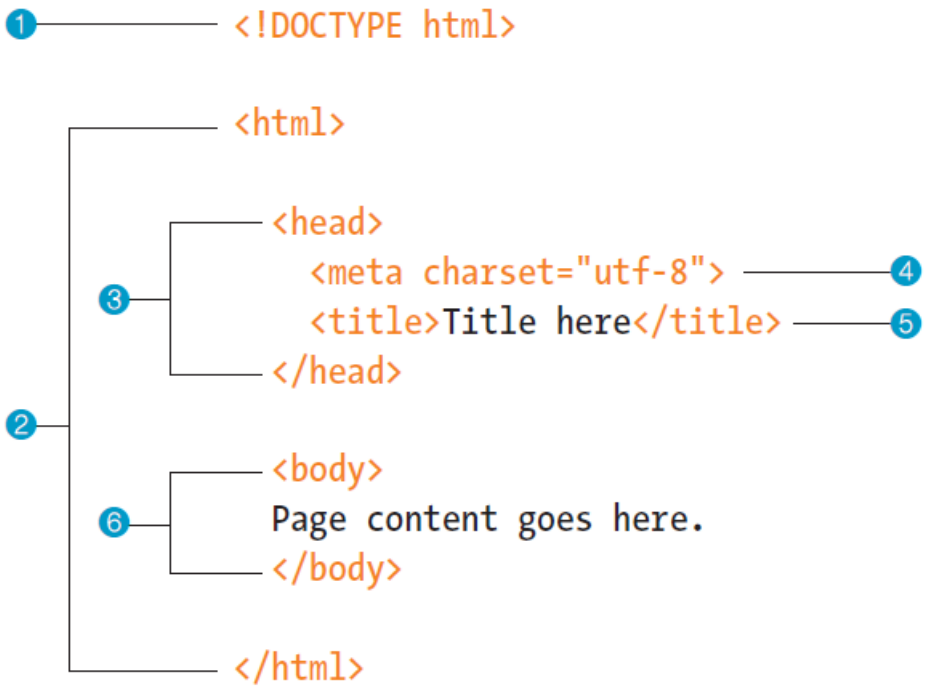


FIGURE 1-8. The minimal structure of an HTML document includes head and body contained within the html root element.

1. I do not 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.
2. 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.
3. 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.
4. *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 do not 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.
5. Also in the head is the mandatory title element. According to the HTML specification, every document must contain a descriptive title. 6. 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 1-2.

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 1-2. Adding minimal structure

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

```
<!DOCTYPE html>
```

II. 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.

III. 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.

IV. 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>
```

V. 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 1-9 shows how it should look now.

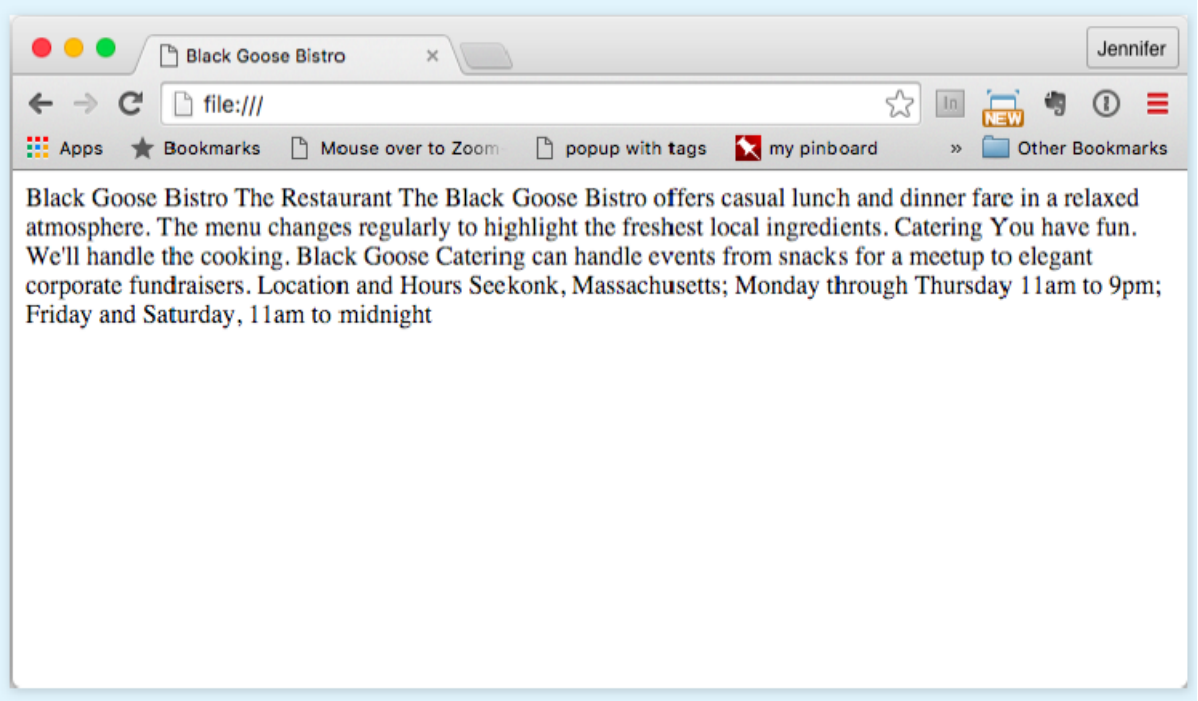


FIGURE 4-9. The page in a browser after the document structure elements have been defined.

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 1-9). 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 will 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 will do in EXERCISE 1-3. However, before we begin, I want to take a moment to talk about what we are 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 1-3.

EXERCISE 1-3. Defining text elements

1. Open the document index.html in your text editor, if it is not open already.
2. The first line of text, “Black Goose Bistro,” is the main heading for the page, so we will 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 us mark those up as paragraph (p) elements in a similar manner. Here’s the first one; you do the rest:

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

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:

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

6. Now that we have marked up the document, let us 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 1-10. If it does not, check your markup to be sure that you aren’t missing any angle brackets or a slash in a closing tag.

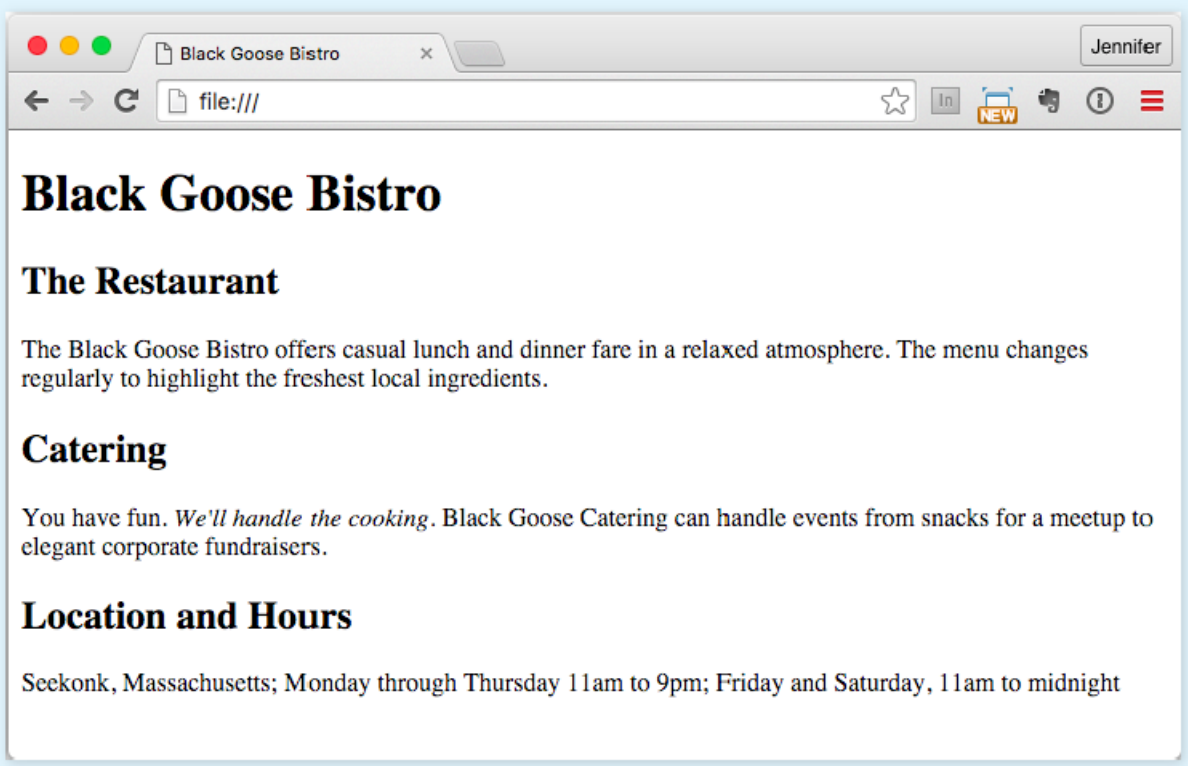


FIGURE 1-10. The home page after the content has been marked up with HTML elements.

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 1-10.

Block and Inline Elements

Although it may seem like stating the obvious, it is worth pointing out that the heading and paragraph elements start on new lines and do not run together as they did before. That is because by default, headings and paragraphs display as block elements. Browsers treat block elements as though they are in little rectangular boxes, stacked up in the page. Each block element begins on a new line, and some space is also usually added above and below the entire element by default. In FIGURE 1-11, the edges of the block elements are outlined in red.

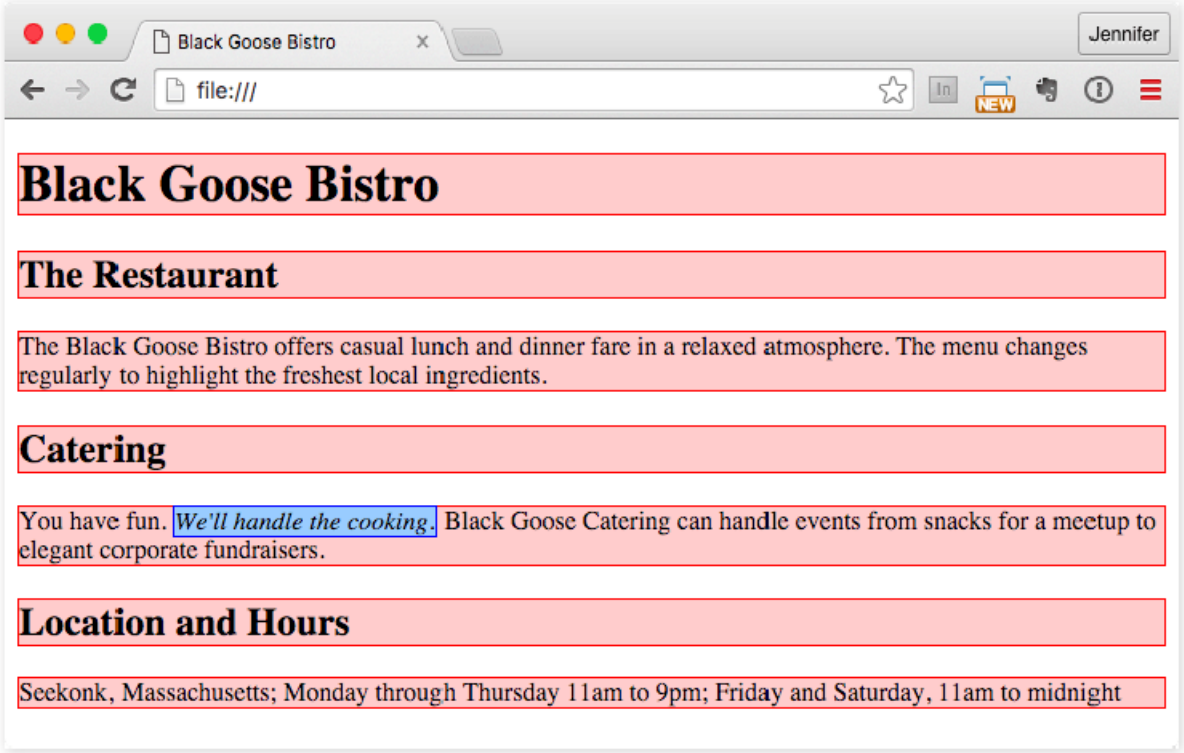


FIGURE 1-11. The outlines show the structure of the elements in the home page.

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 1-10 and 1-11 is that the browser makes an attempt to give the page some visual hierarchy by making the first-level heading the biggest and boldest thing on the page, with the second-level headings slightly smaller, and so on.

How does the browser determine what an h1 should look like? It uses a style sheet! All browsers have their own built-in style sheets (called user agent style sheets in the spec) that describe the default rendering of elements. The default rendering is similar from browser to browser (for example, h1s 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 is not 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 do not worry about the browser's default rendering.

We will 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

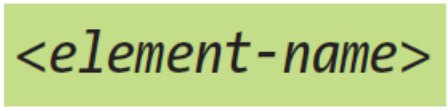
What fun is a web page with no images? In EXERCISE 1-4, we will add an image to the page with the *img* element. This give us an opportunity to introduce two more basic markup concepts: empty elements and attributes.

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 1-6: 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 do not affect its displayed content, such as the meta element that we used earlier.

FIGURE 1-12 shows the very simple syntax of an empty element (compare it to FIGURE 1-6).



Example: The br element inserts a line break.

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

FIGURE 1-12. 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 1-13 shows an img element with its required attributes labeled.

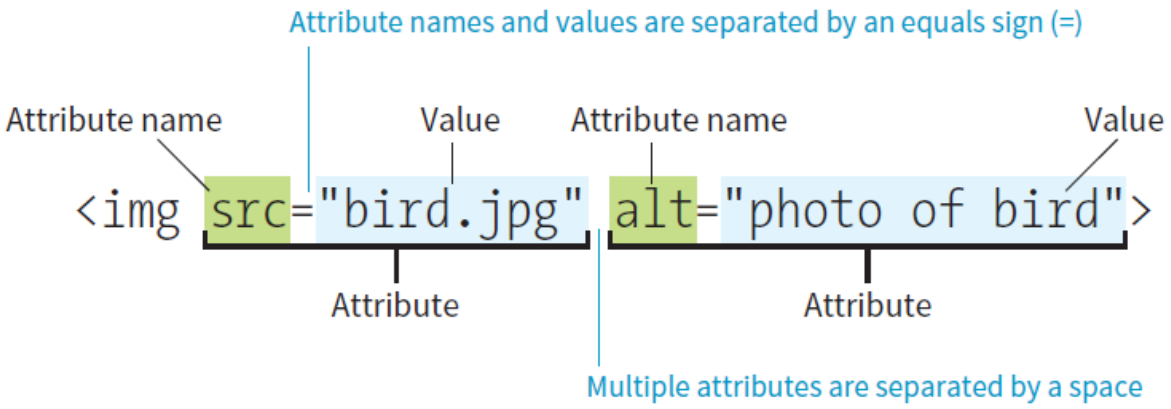


FIGURE 1-13. An img element with two attributes.

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 will 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 cannot 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 will throw a few line breaks in there as well.

EXERCISE 1-4. 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 1-14. 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>

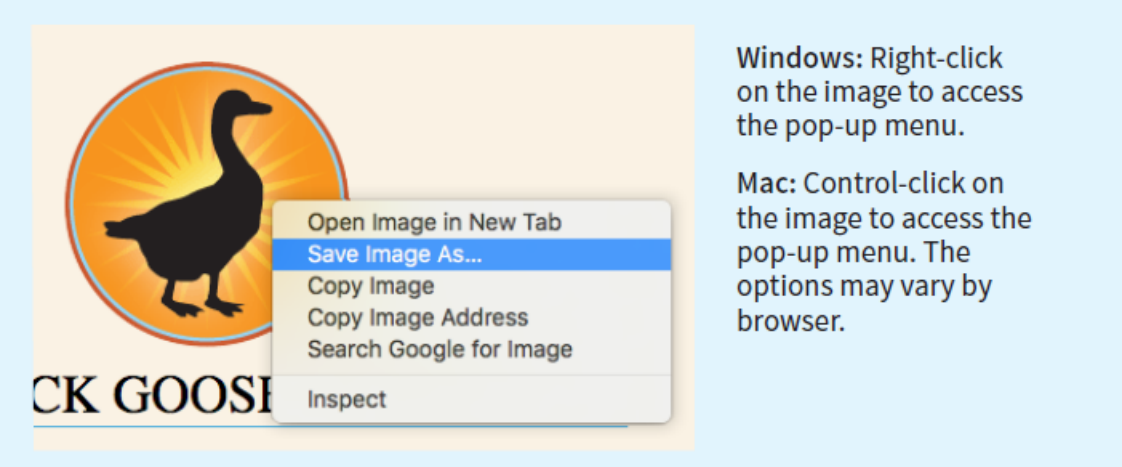


FIGURE 1-14. Saving an image file from a page on the web.

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">
Black Goose Bistro</h1>*

4. Let's break up the last paragraph into three lines for better clarity. Drop a <br> tag at the spots you'd like the line breaks to occur. Try to match the screenshot in FIGURE 1-15.
5. Now save index.html and open or refresh it in the browser window. The page should look like the one shown in FIGURE 1-15. 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.

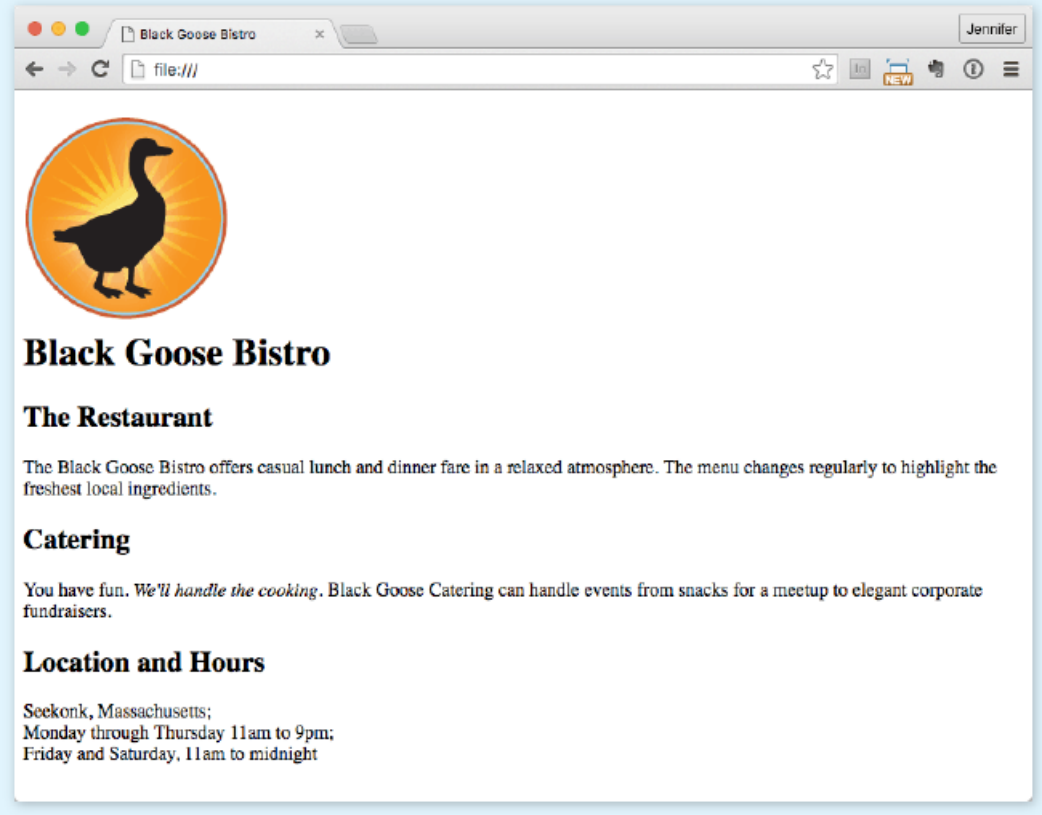


FIGURE 4-15. The Black Goose Bistro page with the logo image.

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 1-5, we will change the appearance of the text elements and the page background by using some simple style sheet rules. Do not worry about understanding them all right now. We will get into CSS in more detail in the next section. However, I want to at least give you a taste of what it means to add a "layer" of presentation onto the structure we have created with our markup.

EXERCISE 1-5. 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.
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).

```
<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 is 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 does not, go over the style sheet to make sure you did not 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 1-15).

FIGURE 1-16. The Black Goose Bistro page after CSS style rules have been applied.

We are 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 am 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 1-17), the remainder of the document displays in emphasized (italic) text. That is because without that slash, there is 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 1-17. When a slash is omitted, the browser doesn't know when the element ends, as is the case in this example.

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.

I have fixed the slash, but this time, let us see what would have happened if I had accidentally omitted a bracket from the end of the first <h2> tag (FIGURE 1-18).

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 do not 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.

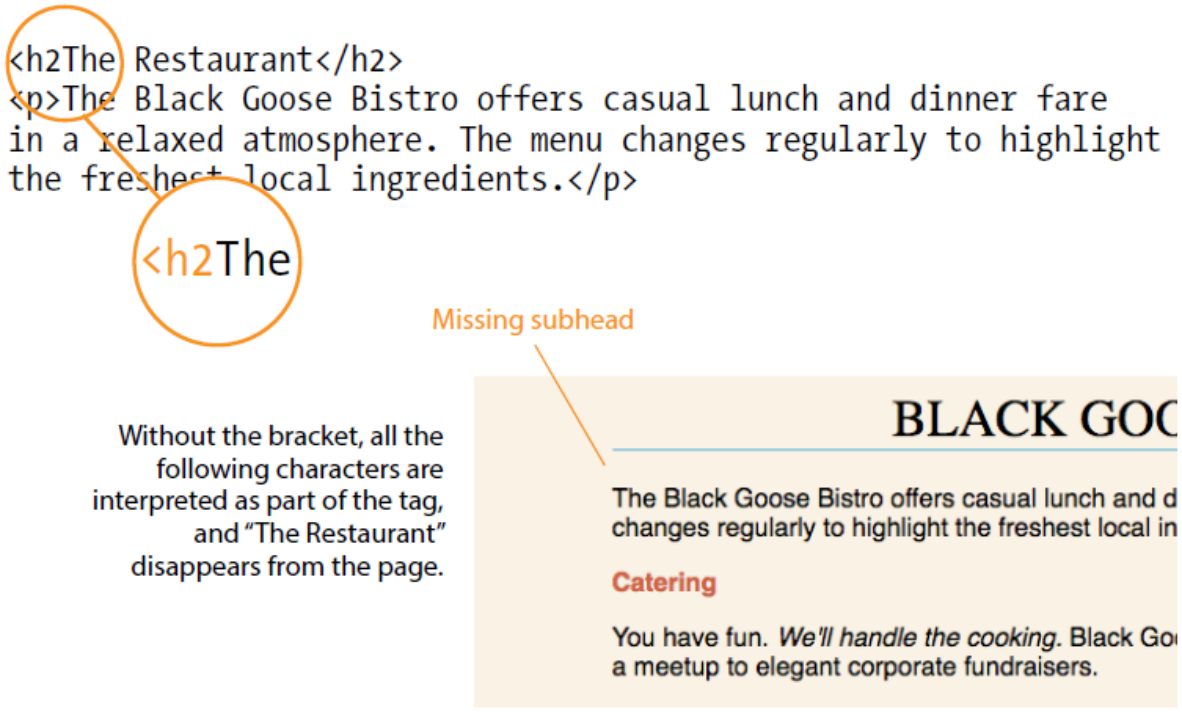


FIGURE 1-18. A missing end bracket makes the browser think the following characters are part of the tag, and therefore the headline text doesn't display.

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 do not 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 are not 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 1-19 shows the report the validator generates when I upload the version of the Bistro index.html file that does not 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.

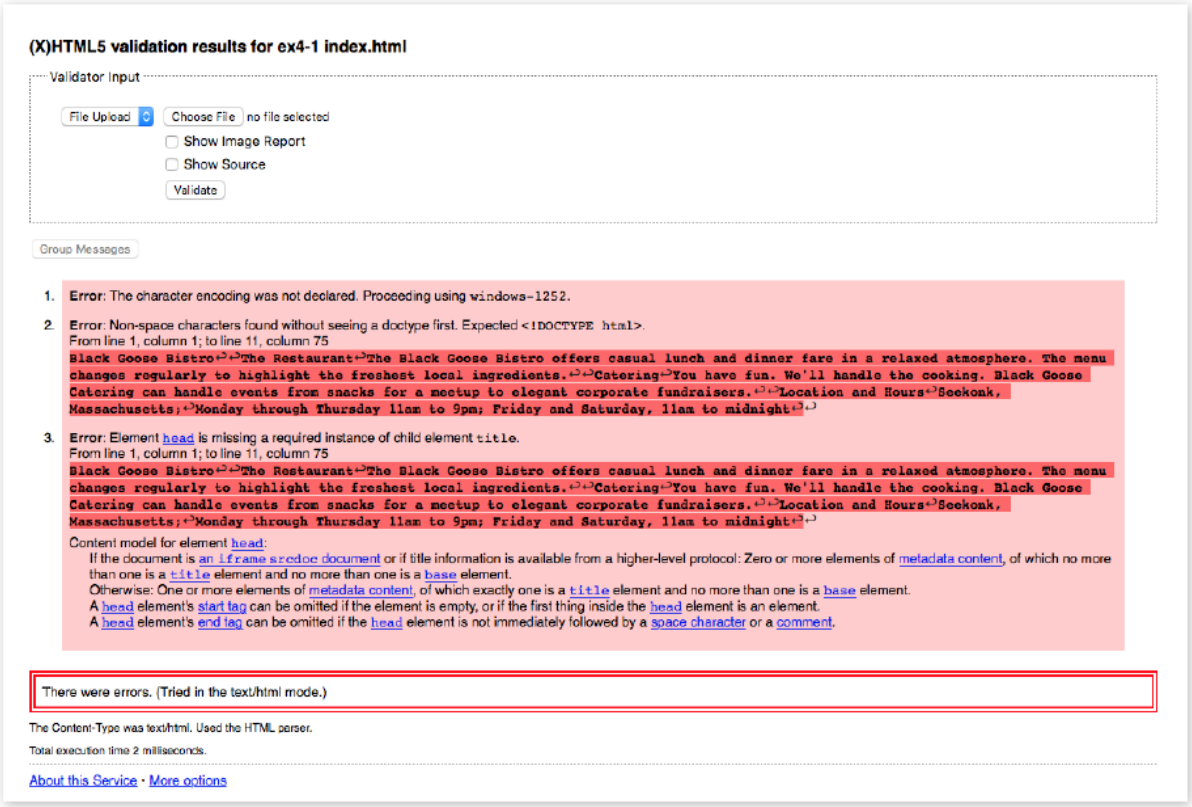


FIGURE 1-19. The (X)HTML5 Validator (Living Validator) for checking errors in HTML documents

ELEMENT REVIEW: HTML DOCUMENT SETUP

This section 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

1.2 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.

PARAGRAPHS

* … *

Paragraph element

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

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

* 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. *

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 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

In the last section, 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.

<h1>…</h1>

<h2>…</h2>

<h3>…</h3>

<h4>…</h4>

<h5>…</h5>

<h6>…</h6>

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>

* Serif typefaces have small slabs at the ends of letter strokes.*

*In general, serif fonts can make large amounts of text easier to read. *

<h3>Baskerville</h3>

<h4>Description</h4>

* Description of the Baskerville typeface. *

<h4>History</h4>

* The history of the Baskerville typeface. *

<h3>Georgia</h3>

* Description and history of the Georgia typeface. *

<h2>Sans-serif Typefaces</h2>

* Sans-serif typefaces do not have slabs at the ends of strokes. *

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 h1s, with each consecutive level in smaller text, as shown in FIGURE 2-1. You can use a style sheet to change their appearance.

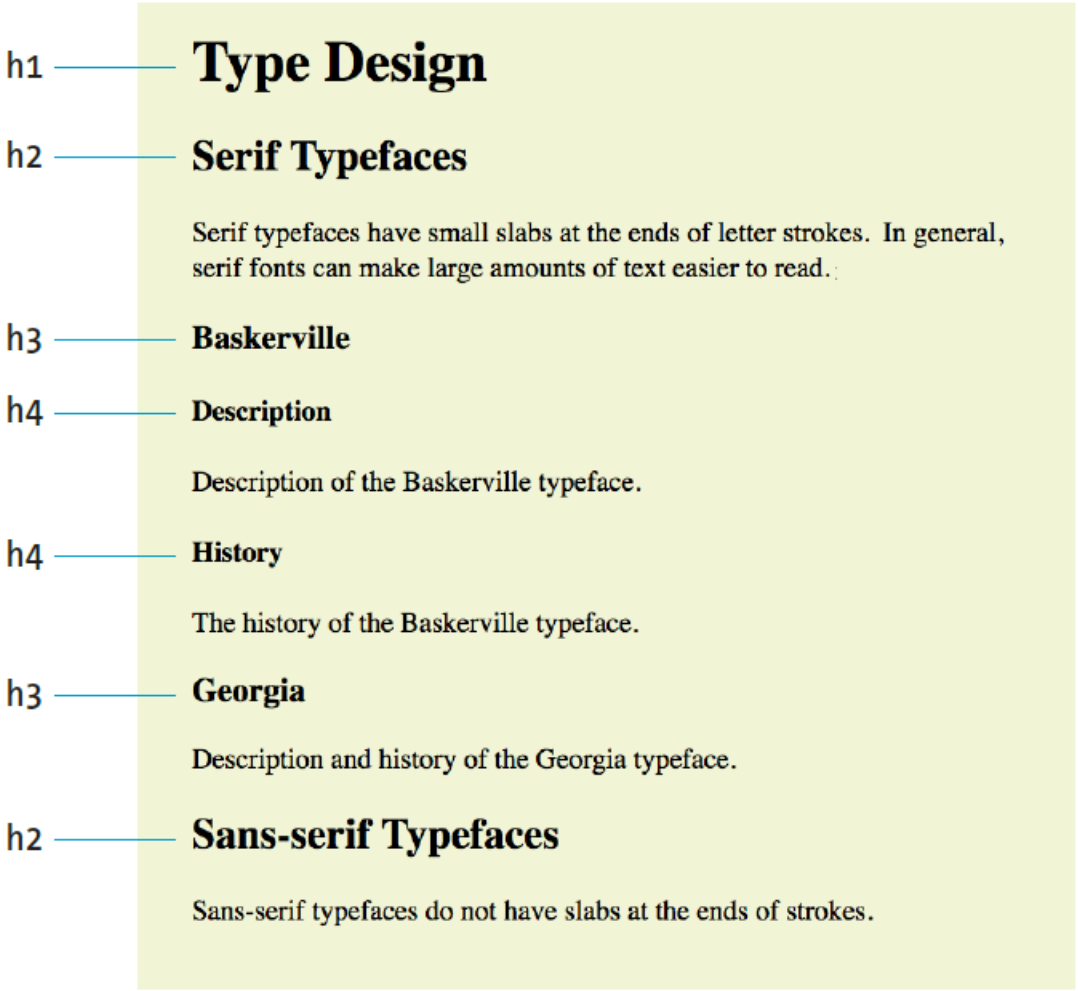


FIGURE 2-1. The default rendering of four heading levels.

THEMATIC BREAKS (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>
```

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 2-2:

```
<h3>Times</h3>

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

<hr>

<h3>Georgia</h3>

* Description and history of the Georgia typeface. *

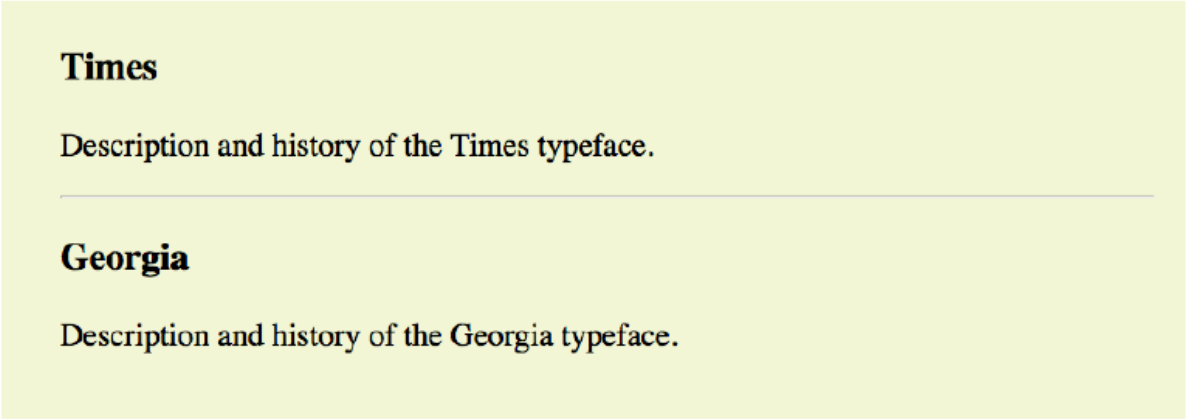


FIGURE 2-2. The default rendering of a thematic break (horizontal rule).

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.

`<ul>…</ul>`

Unordered list

`<li>…</li>`

List item within an unordered list

Unordered Lists

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 <ul> 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 2-3).

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>
```

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

FIGURE 2-3. The default rendering of the sample unordered list. The browser adds the bullets automatically.

However, 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 2-4. 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.

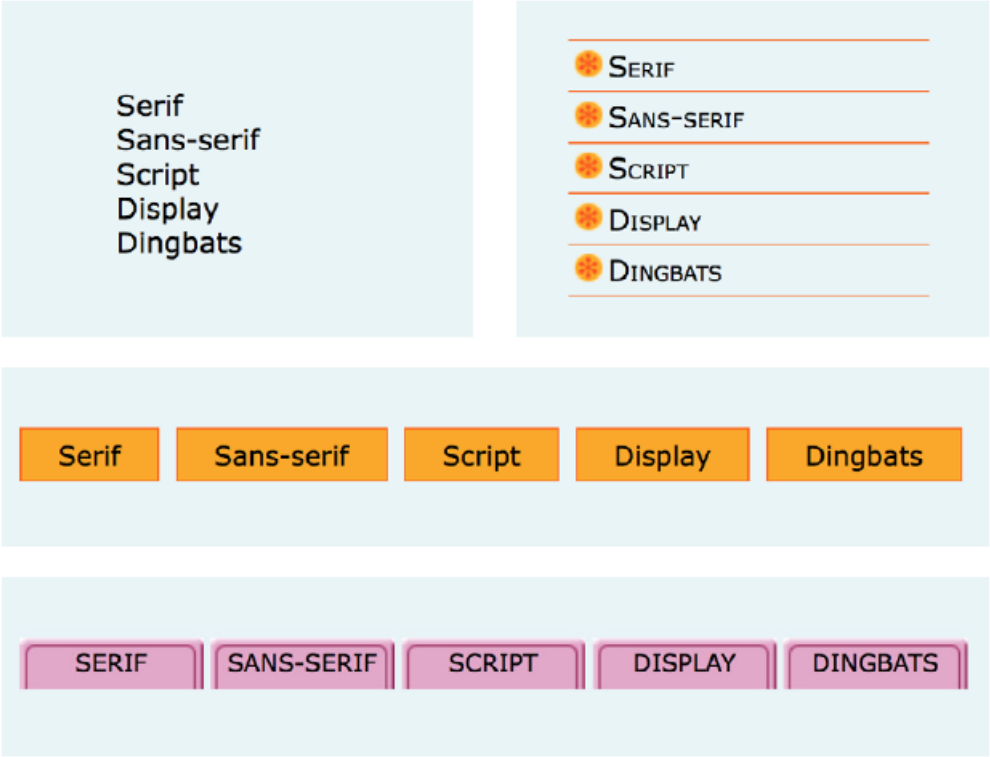


FIGURE 2-4. With style sheets, you can give the same unordered list many looks.

Ordered Lists

…

Ordered list

…

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.

Ordered list elements must contain one or more list item elements, as shown in this example and in FIGURE 2-5:

```
<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 2-5. The default rendering of an ordered list. The browser adds the numbers automatically.

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.*

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.

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.

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 2-6):

```
<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>
```

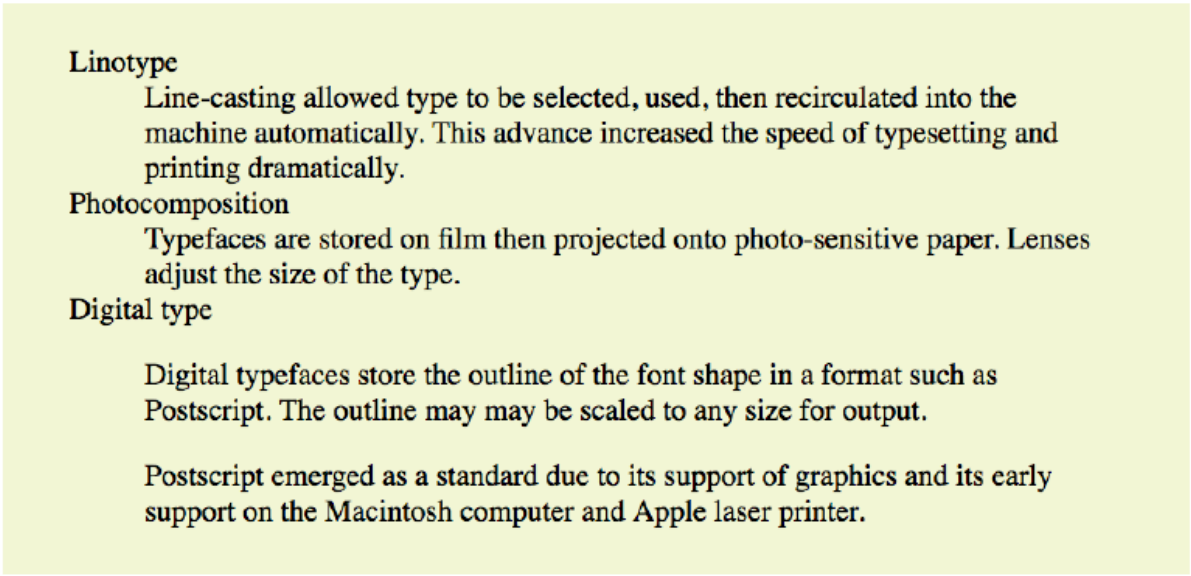


FIGURE 2-6. The default rendering of a definition list. Definitions are set off from the terms by an indent.

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 section).

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:

* Renowned type designer, Matthew Carter, has this to say about his profession: *

<blockquote>

* Our alphabet hasn't changed in eons; there isn't much latitude in what a designer can do with the individual letters. *

* 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. *

</blockquote>

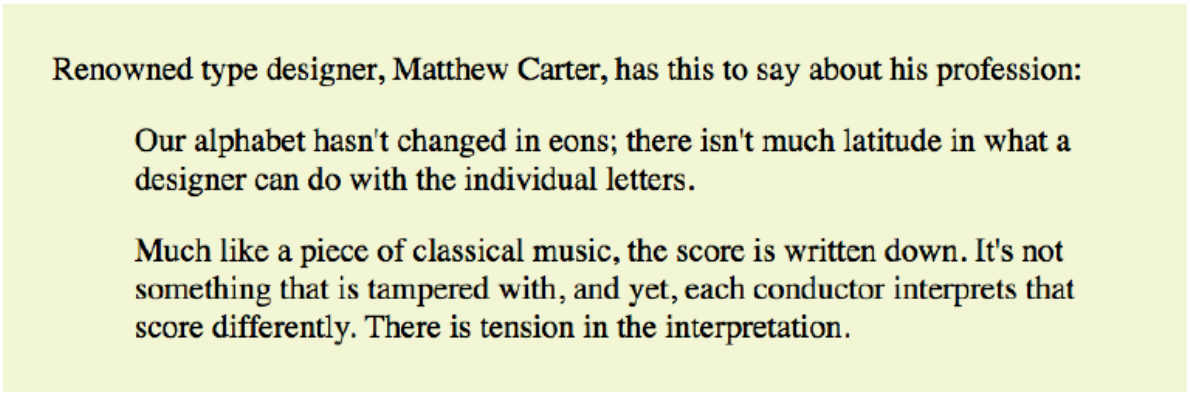


FIGURE 2-7 shows the default rendering of the blockquote example. This can be altered with CSS.

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

NOTE

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

In the previous section, 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.

The pre element in this example displays as shown in FIGURE 2-8. The second part of the figure shows the same content marked up as a paragraph (p) element for comparison.

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

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

This is an example of text with a lot of curious whitespace.

FIGURE 2-8. 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 selfcontained 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 2-1, you'll get a chance to mark up a document yourself and try out the basic text elements we've covered so far.

EXERCISE 2-1. 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

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

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>

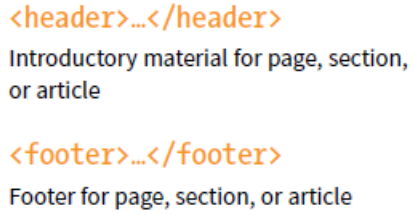
<header>…</header>

```
<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



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>
```

NOTE

The `<a href="/"></a>` *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.*

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>
        <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

NOTE

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

Long documents are easier to use when they are divided into smaller parts.

For example, books are divided into chapters, and newspapers have sections for local news, sports, comics, and so on. To divide long web documents into thematic sections, use the aptly named section element. Sections typically 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>
    <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

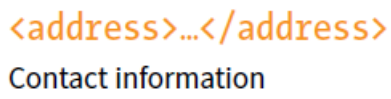
The nav element gives developers a semantic way to identify navigation for a site. Earlier in this chapter, we saw an unordered list that might be used as the top-level navigation for a font catalog site. Wrapping that list in a nav element makes its purpose explicitly clear:

```
<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



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 2-1 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 2-9), 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 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 2-9 should hold no surprises.

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

TABLE 2-1. Text-level semantic elements

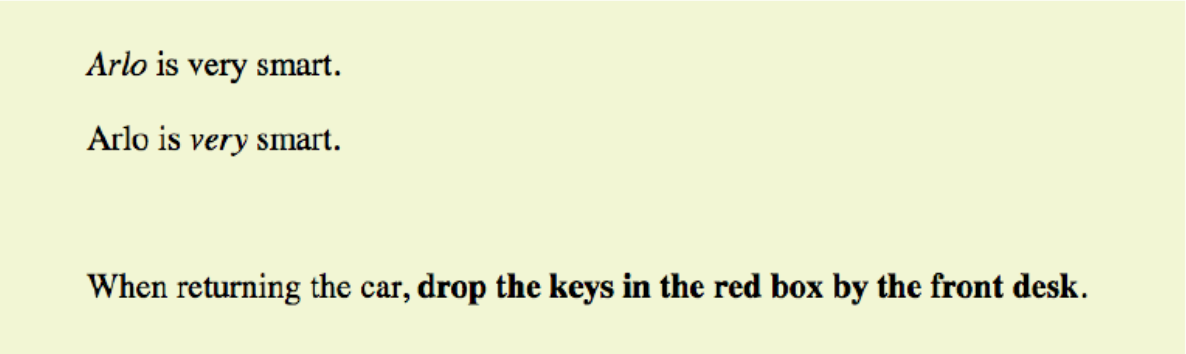


FIGURE 2-9. The default rendering of emphasized and strong text.

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 2-10). 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]

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**.

CSS Property: For bold text, use font-weight. Example: font-weight: bold;

Example: The slabs at the ends of letter strokes are called

*serifs. *

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: Simply change the font and Voila!, a new

*personality! *

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: Scala Sans was designed by <s>Eric Gill</s> Martin Majoor. *

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: textdecoration: underline

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

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

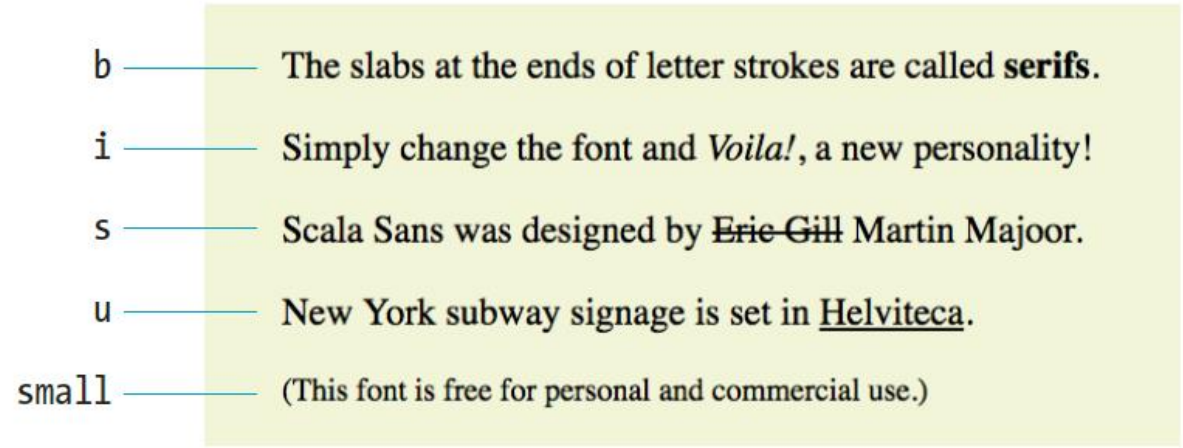
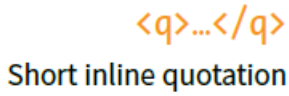


FIGURE 2-10. The default rendering of b, i, s, u, and small elements.

Short quotations



Use the quotation (q) element to mark up short quotations, such as “To be or not to be,” in the flow of text, as shown in this example (FIGURE 2-11):

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.

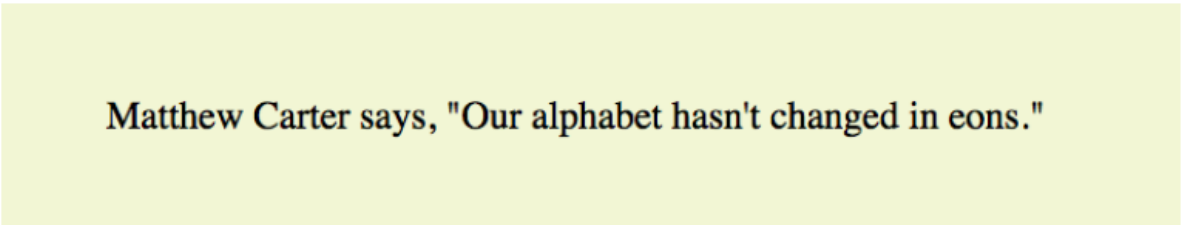


FIGURE 2-11. Browsers add quotation marks automatically around q elements.

Abbreviations and acronyms

<abbr>…</abbr>

Abbreviation or acronym

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.*

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>
```

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

`<cite>…</cite>`

Citation

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

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

Defining terms

`<dfn>…</dfn>`

Defining term

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

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

Program code elements

`<code>…</code>`

Code

`<var>…</var>`

Variable

`<samp>…</samp>`

Program sample

`<kbd>…</kbd>`

User-entered keyboard strokes

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

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

Subscript and superscript

_…

Subscript

[…]

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 2-12 shows how these examples of subscript and superscript typically render in a browser.

* H₂0 *

* E=MC² *

FIGURE 2-12. 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 2-13.

* ... 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>. *

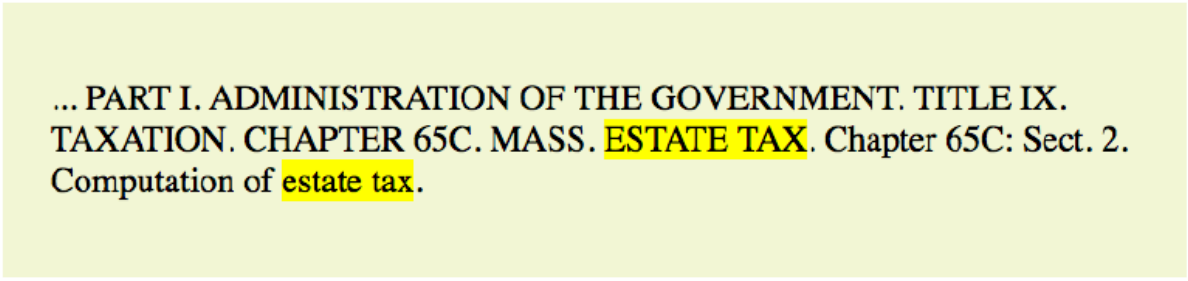


FIGURE 2-13. In this example, search terms are identified with mark elements and given a yellow background with a style sheet so they are easier for the reader to find.

Dates and times

`<time>…</time>`

Time data

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."*

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 2-14. 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.

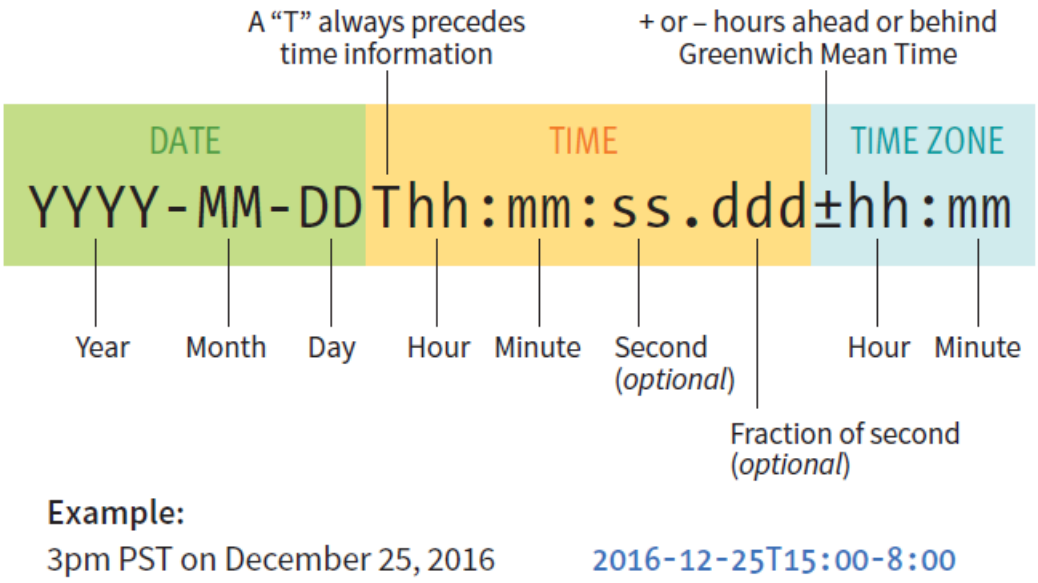


FIGURE 2-14. Standardized date and time syntax.

Here are a few examples of valid values for datetime:

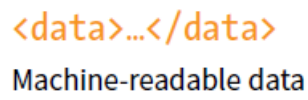
- Time only: 9:30 p.m.
  <time datetime="21:30">9:30p.m.</time>
- Date only: June 19, 2016
  <time datetime="2016-06-19">June 19, 2016</time>
- Date and time: Sept. 5, 1970, 1:11a.m.
  <time datetime="1970-09-05T01:11:00">Sept. 5, 1970, 1:11a.m.</time>
- Date and time, with time zone information: 8:00am 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.

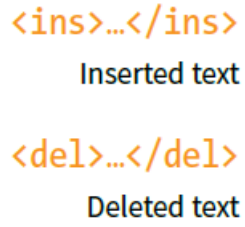
It can be used for all sorts of data, including dates, times, measurements, weights, microdata, and so on. The required value attribute provides the machine-readable information. Here are a couple of examples:

<data value="12">Twelve</data>

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

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



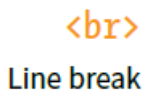
The ins and del elements are used to mark up edits indicating parts of a document that have been inserted or deleted (respectively). These elements rely on style rules for presentation (i.e., there is no dependable browser default).

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

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 2-15:

* So much depends
upon

a red wheel
barrow *

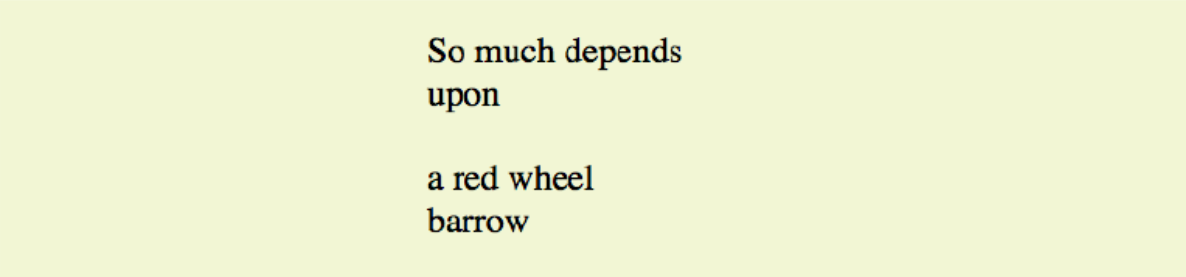


FIGURE 2-15. Line breaks are inserted at each br element. (Example extracted from "The Red Wheelbarrow" by William Carlos Williams.)

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:

* Times
*

*Georgia
*

Garamond

* *

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

<ul>

*
Times
*

*
Georgia
*

*
Garamond
*

*
*

Word breaks

<wbr>

Word break

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.*

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 2-16). 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>
```

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

FIGURE 2-16. When there is not enough room for a word to fit on a line, it will break at the location of the wbr element.

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 2-2.

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 2-2. 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>

* posted by BGB, November 15, 2016 *

</header>

<h2>Low and Slow</h2>

* 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. *

* blackgoose@example.com*

*555-336-1800 *

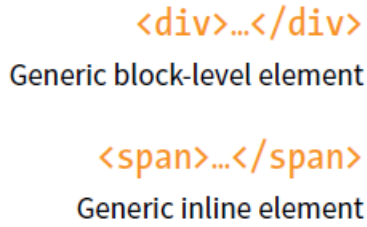
* Warning: Sous vide cooked salmon is not*

pasteurized. Avoid it if you are pregnant or have

*immunity issues. *

</article>

GENERIC ELEMENTS (DIV AND SPAN)



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 divs less often now that we have better semantic sectioning elements in HTML5.

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 spans 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 spans add meaning to what otherwise might be a random string of digits. As a bonus, the span element enables us to apply the same style to phone numbers throughout the site (for example, ensuring line breaks never happen within them, using a CSS white-space: nowrap declaration).

It makes the information recognizable not only to humans but 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

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.

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>
```

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>
```

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.

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, 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.

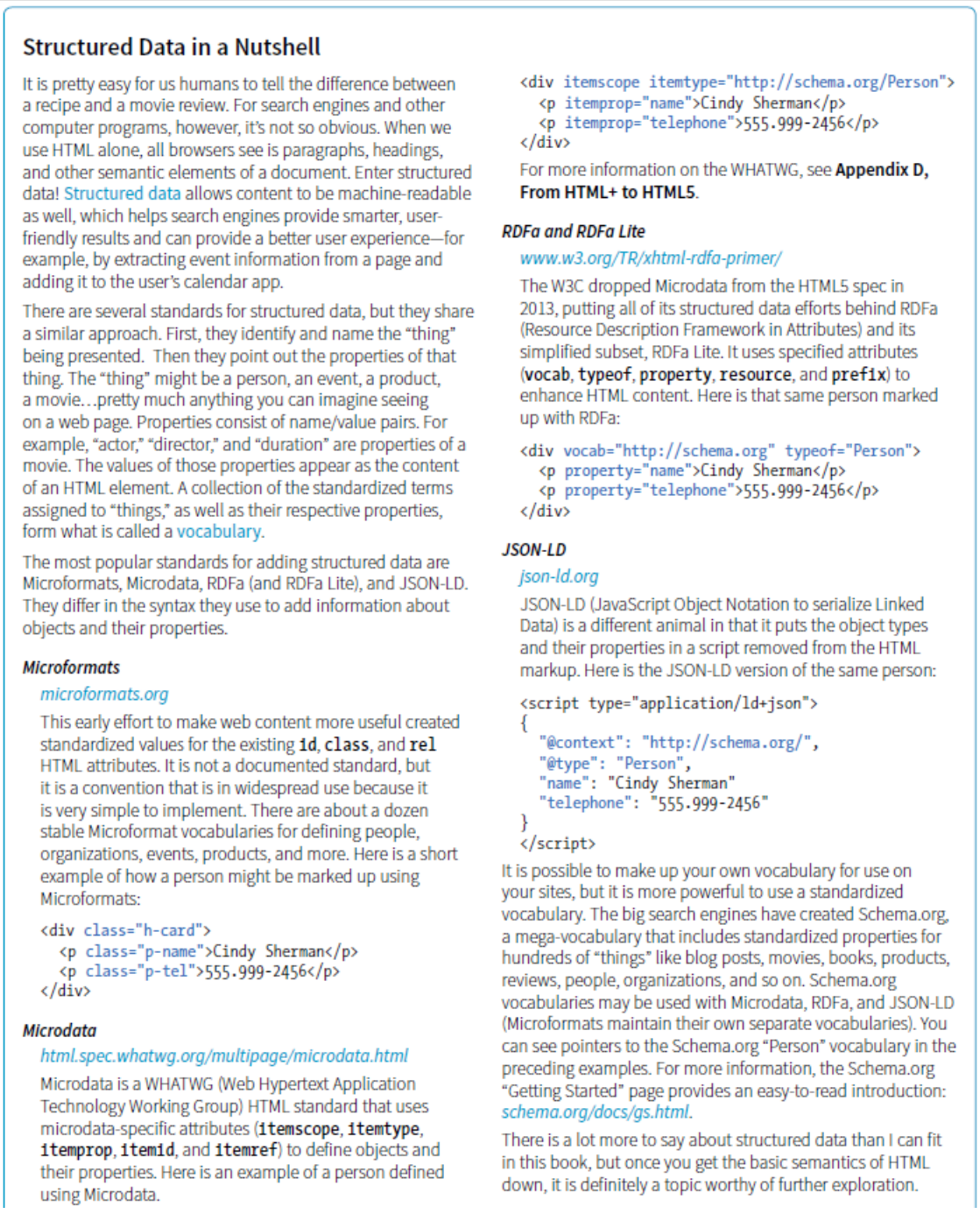
Many HTML elements are plainly understood when you look at (or read) only the HTML source. Elements like the title, headings, lists, images, and tables have implicit meanings in the context of a page, but generic elements like div and span lack the semantics necessary to be interpreted by an assistive 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.

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:

* You are no longer connected to the*

*server. *

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.

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 2-17):

* 3 tsp. < 3 Tsp. *

or:

* 3 tsp. < 3 Tsp. *

3 tsp. < 3 Tsp.

FIGURE 2-17. 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

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 2-2.)

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

TABLE 2-2. Syntax characters and their character references

Invisible or ambiguous characters

Some characters have no graphic display and are difficult to see in the markup (TABLE 2-3). 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 nonjoiners prevent neighboring characters from joining to form ligatures or other connected forms.

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)		
(non-printing)	Zero-width non-joiner	‌	‌	‌
(non-printing)	Zero-width joiner	‍	‍	‍

TABLE 2-3. Invisible characters and their character references

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 2-4 lists some special characters that may be less straightforward to type into the source.

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;

TABLE 2-4. Special characters and their character references

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 2-3 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 2-3. 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 2-18.

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

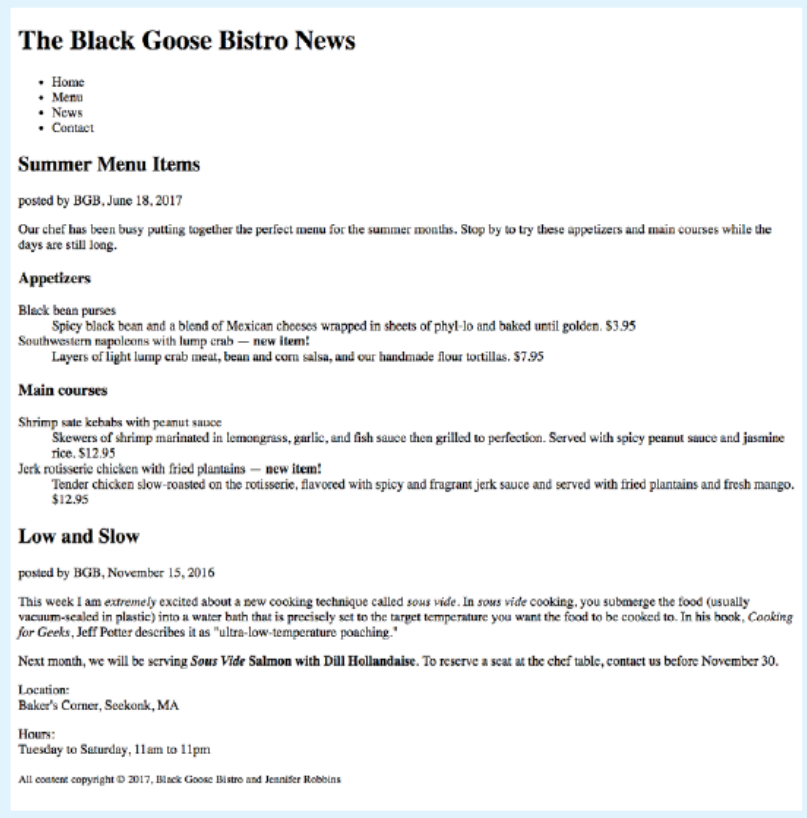
* This week I am extremely excited about a new cooking technique called <dfn>sous vide</dfn>. In sous vide cooking, you submerge the food (usually vacuum-sealed in plastic) into a water bath that is precisely set to the target temperature you want the food to be cooked to. In his book, <cite>Cooking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

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

Location: Baker's Corner, Seekonk, MA

Hours: Tuesday to Saturday, 11am to 11pm

All content copyright 2017, Black Goose Bistro and Jennifer Robbins



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 frugrant 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 2-18. 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 2-2.
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 © 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?

1.3 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).

<a>…

Anchor element (hypertext link)

To make a selection of text a link, simply wrap it in opening and closing <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.

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 3-1.

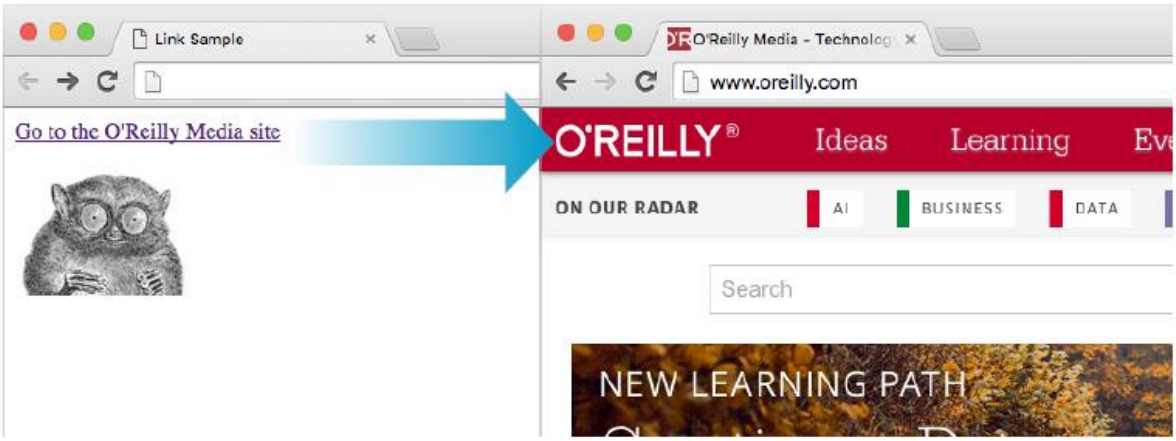


FIGURE 3-1. When a user clicks or taps the linked text or image, the page specified in the anchor element loads in the browser window.

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 3-2). 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 section, we'll add links using absolute and relative URLs to my cooking website, Jen's Kitchen (see FIGURE 3-3). Absolute URLs are easy, so let's get them out of the way first.

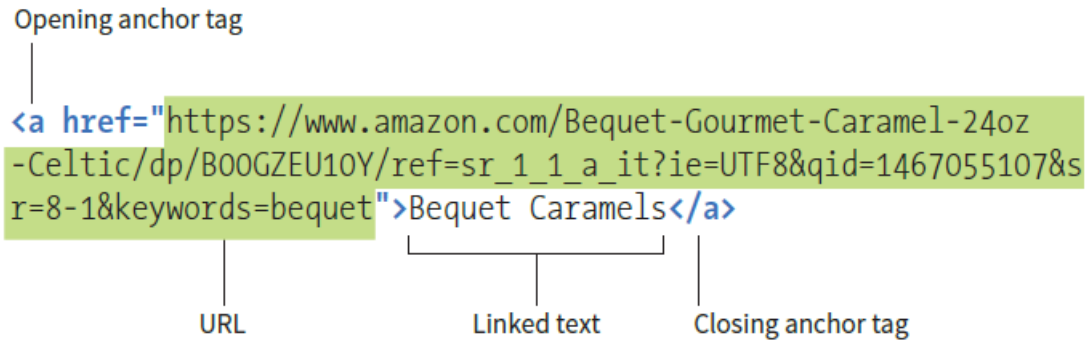


FIGURE 3-2. An example of a long URL. Although it may make the anchor tag look confusing, the structure is the same.

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

Note!

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.

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 3-3).

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.

*
<a>The Food Network
*

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

*
The Food Network
*

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 3-1.

EXERCISE 3-1. 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 3-2 through 4-8 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 3-4. 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.

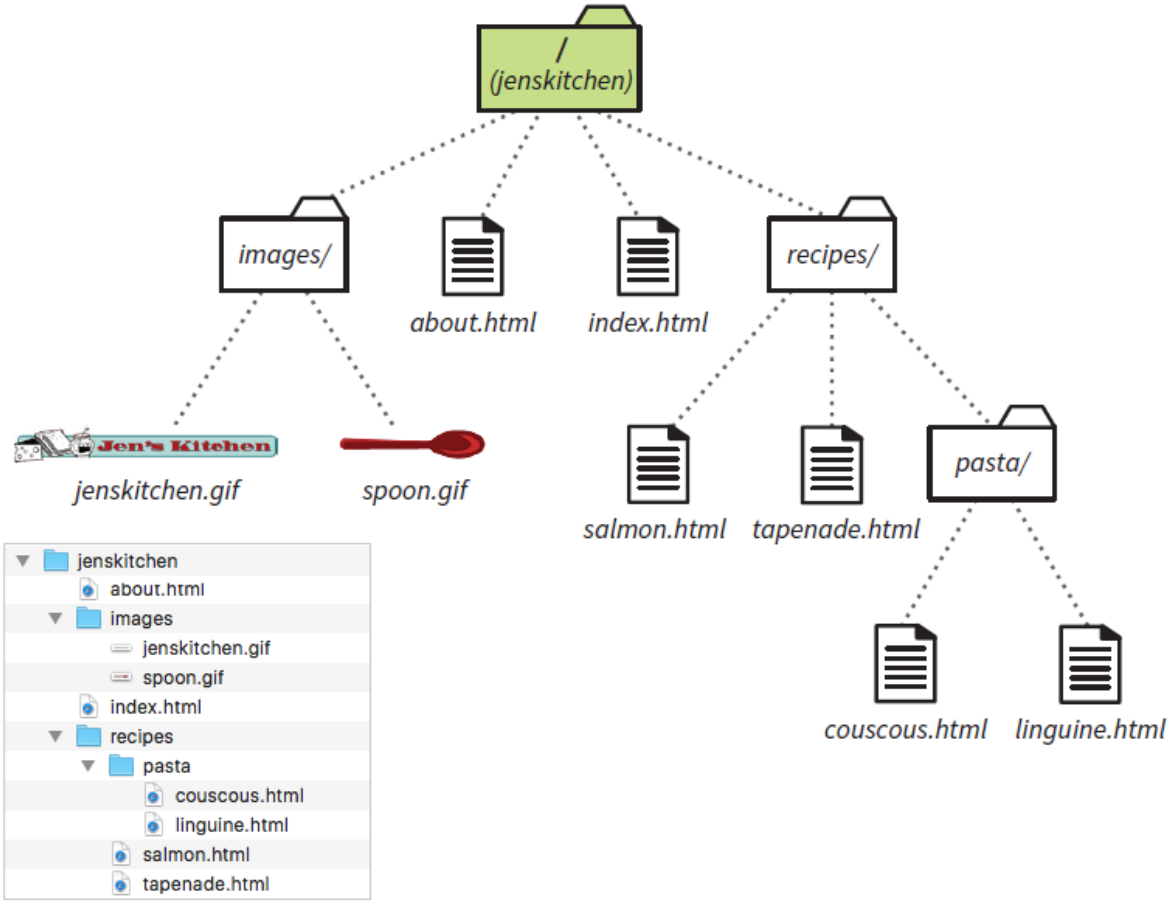


FIGURE 3-4. A diagram of the jenskitchen site structure.

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.

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.

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 3-5):

About the site...

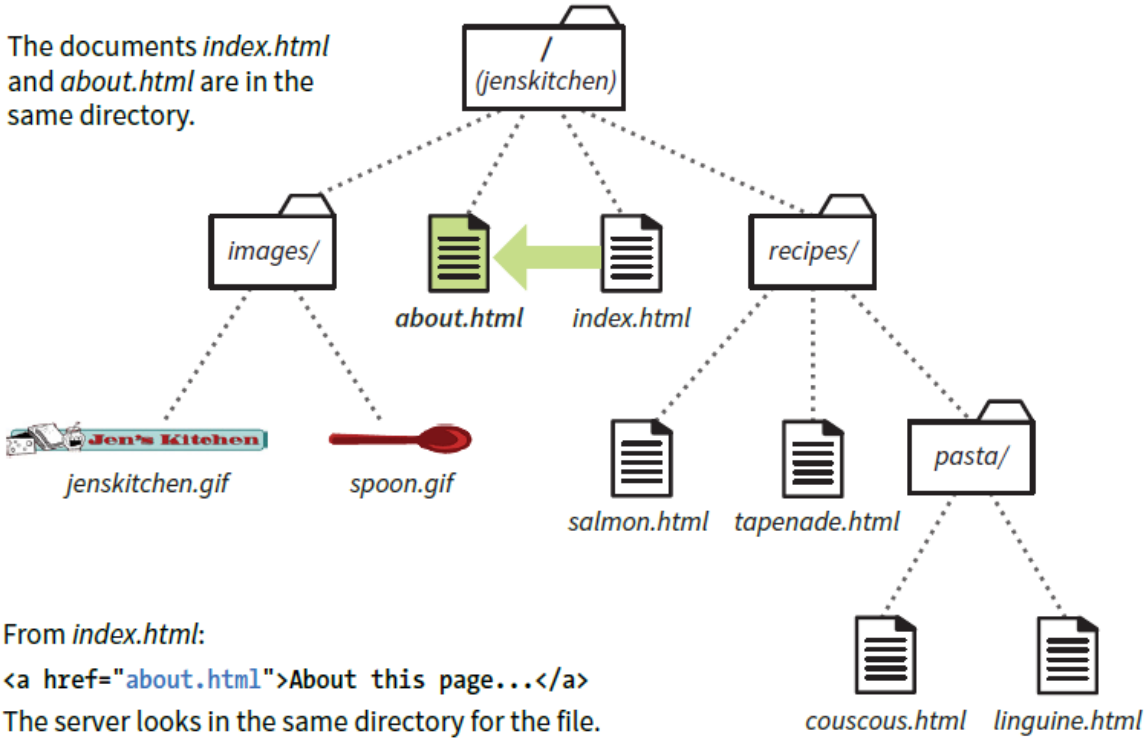


FIGURE 3-5. Writing a relative URL to another document in the same directory.

EXERCISE 3-2. 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:

* Back to the home page *

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):

*
Garlic Salmon
*

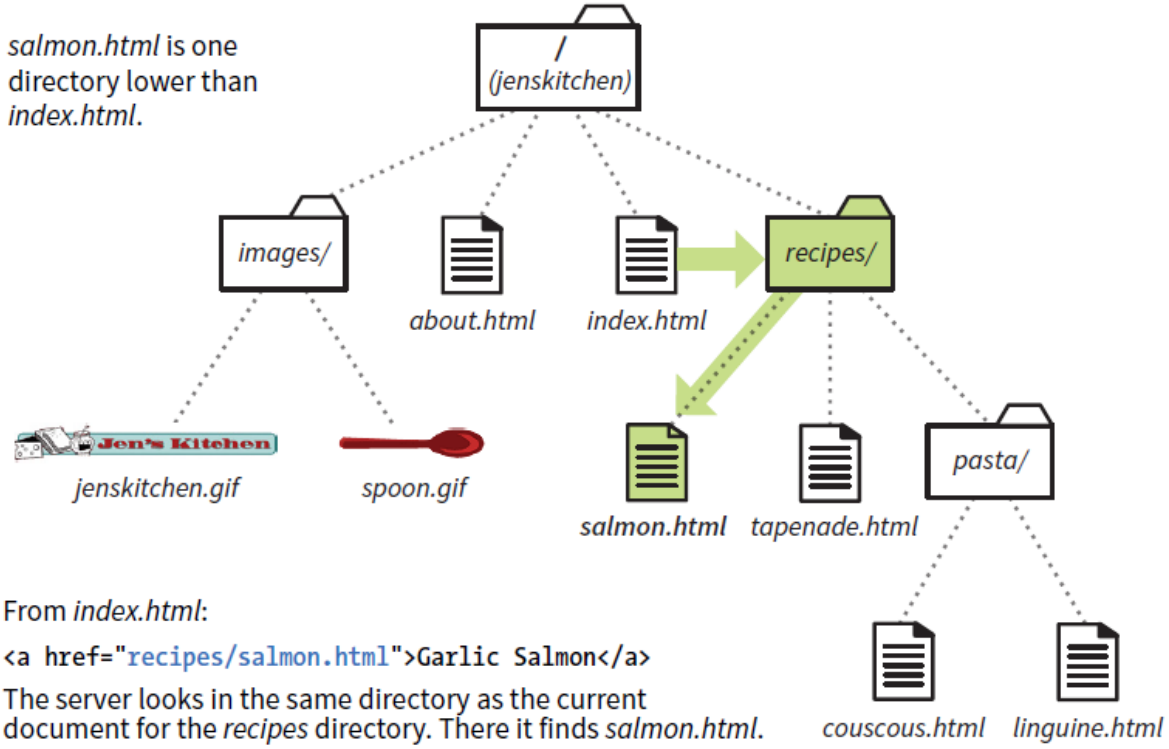


FIGURE 3-6. Writing a relative URL to a document that is one directory level lower than the current document.

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

EXERCISE 3-3. 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:

*
Tapenade (Olive Spread)
*

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 3-7):

*
Couscous...
*

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.

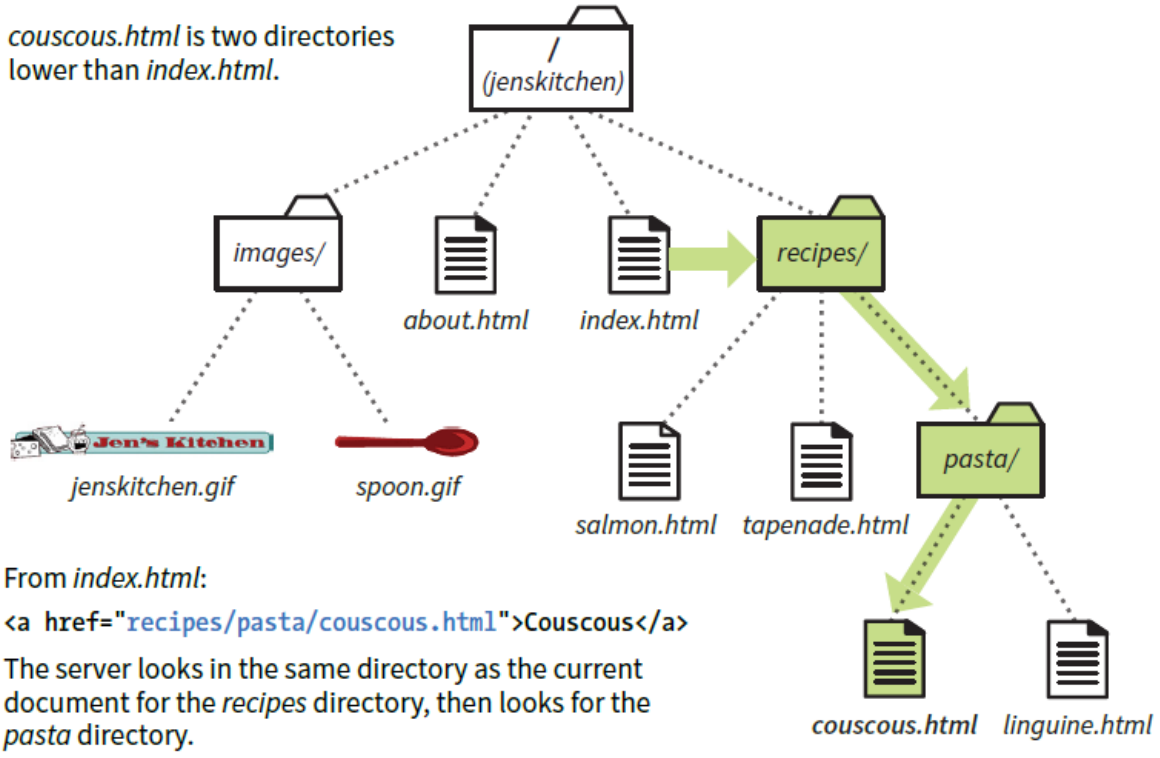


FIGURE 3-7. Writing a relative URL to a document that is two directory levels lower than the current document. You can try it yourself in EXERCISE 3-4.

EXERCISE 3-4. 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:

*
Linguine with Clam Sauce
*

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

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-dotslash” (../). 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 3-8):

* [Back to home page] *

Note that the ../ stands in for the name of the higher directory, and we don't need to write out jenskitchen in the pathname.

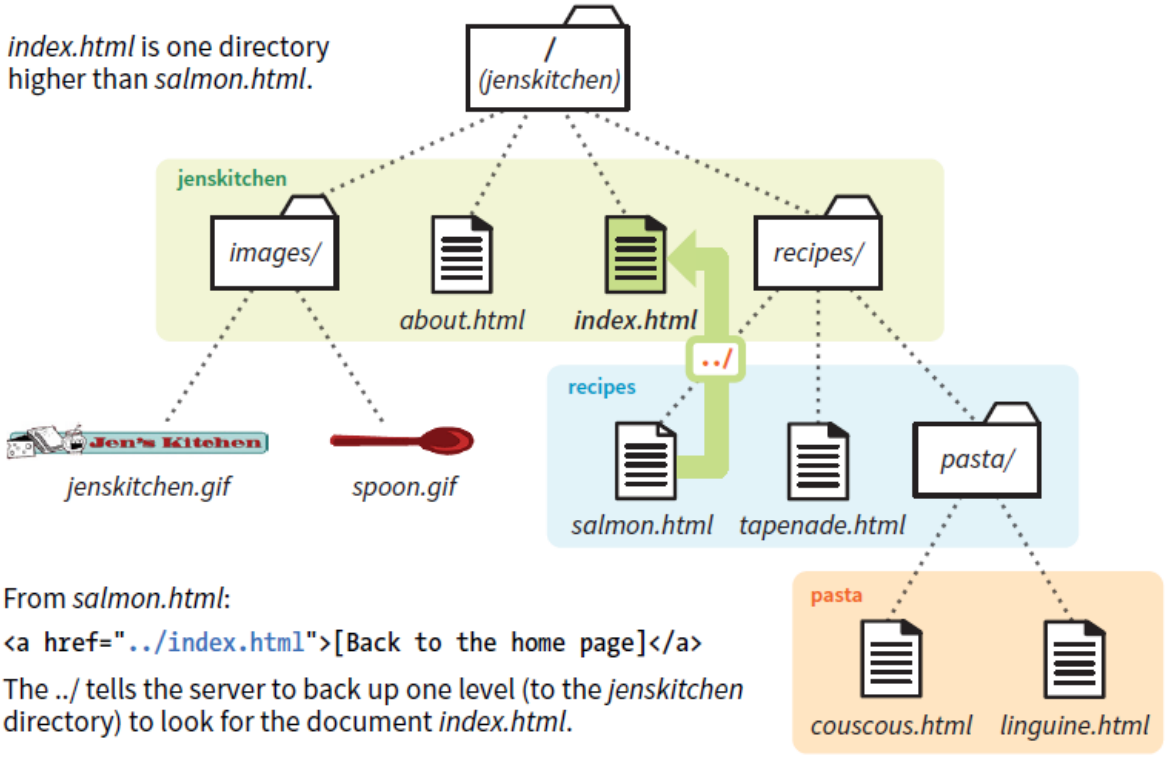


FIGURE 3-8. Writing a relative URL to a document that is one directory level higher than the current document.

Try adding a dot-dot-slash pathname to a higher directory in EXERCISE 3-5. 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 3-9).

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

* [Back to home page] *

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 3-6).

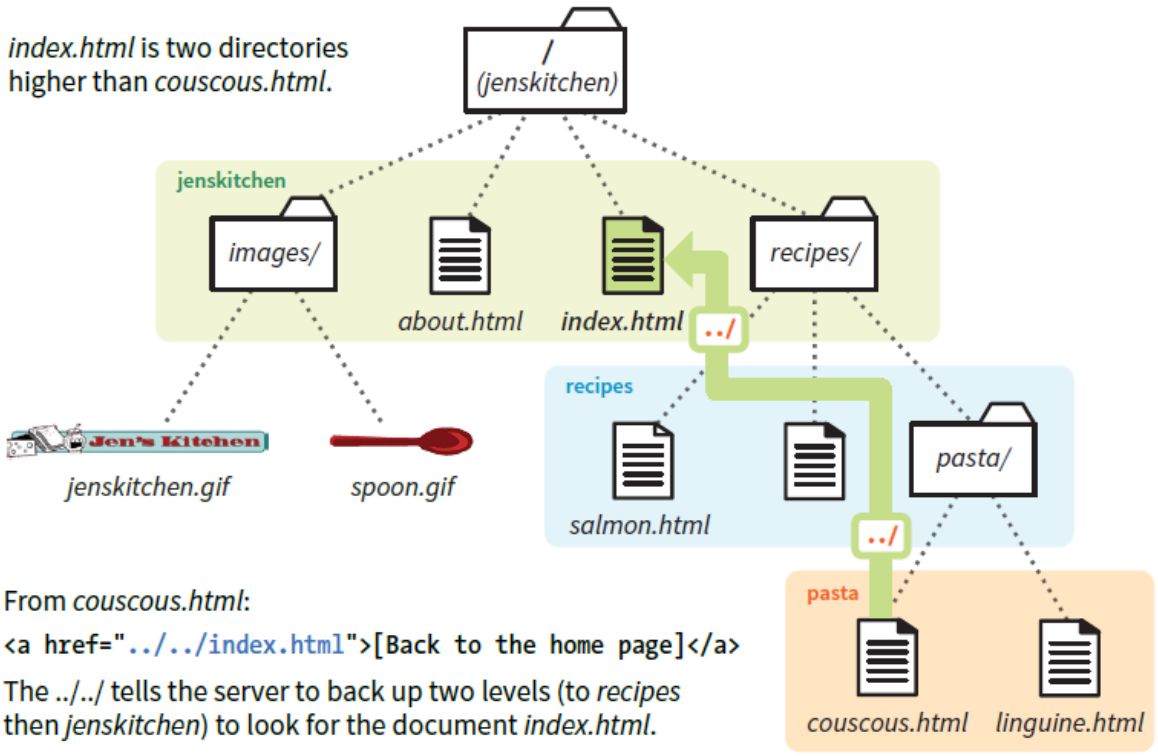


FIGURE 3-9. Writing a relative URL to a document that is two directory levels higher than the current document.

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 3-5. 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:

* [Back to the home page] *

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

EXERCISE 3-6. 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:

* [Back to the home page] *

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

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 3-10):

Garlic Salmon

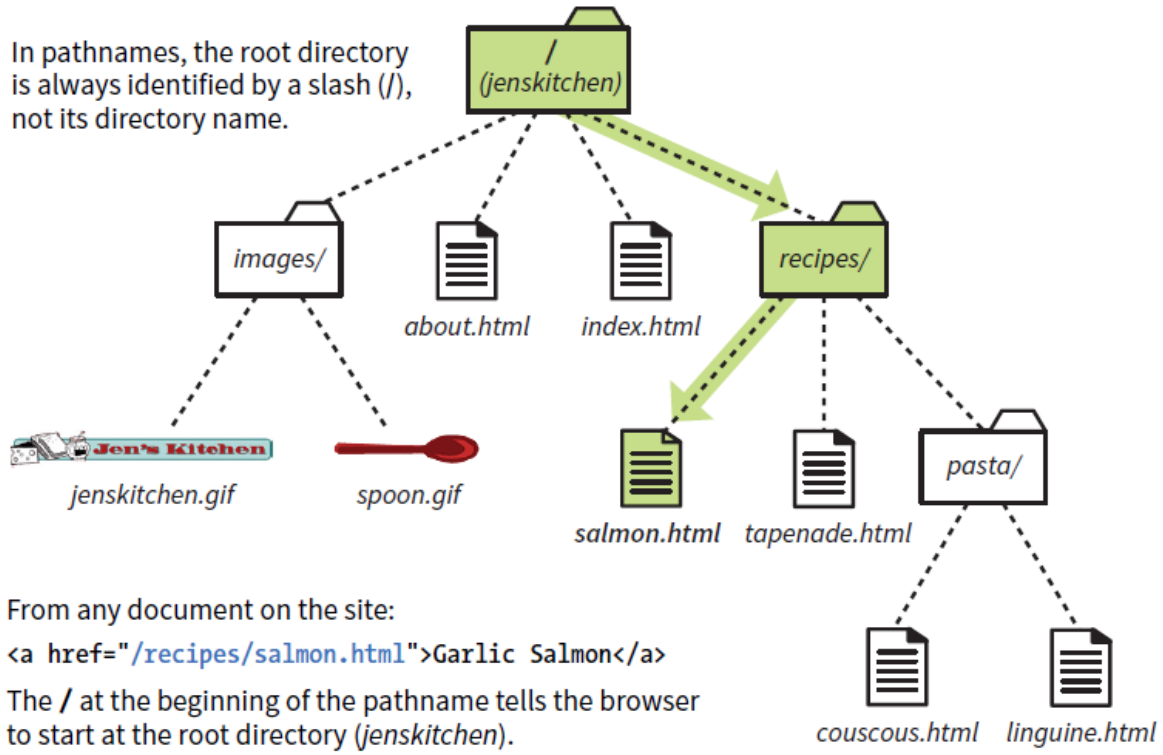


FIGURE 3-10. Writing a relative URL starting at the root directory.

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 3-7. 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 section.

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

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.

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 3-11 1):

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

Step 2: Linking to the destination

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

At the top of the page, I'll create a link down to the "startH" fragment 2. 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 3.

Identify the destination by using the id attribute.

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

<dl>

<dt>hexadecimal</dt>

...

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

* ... | F | G | H | I | J ... *

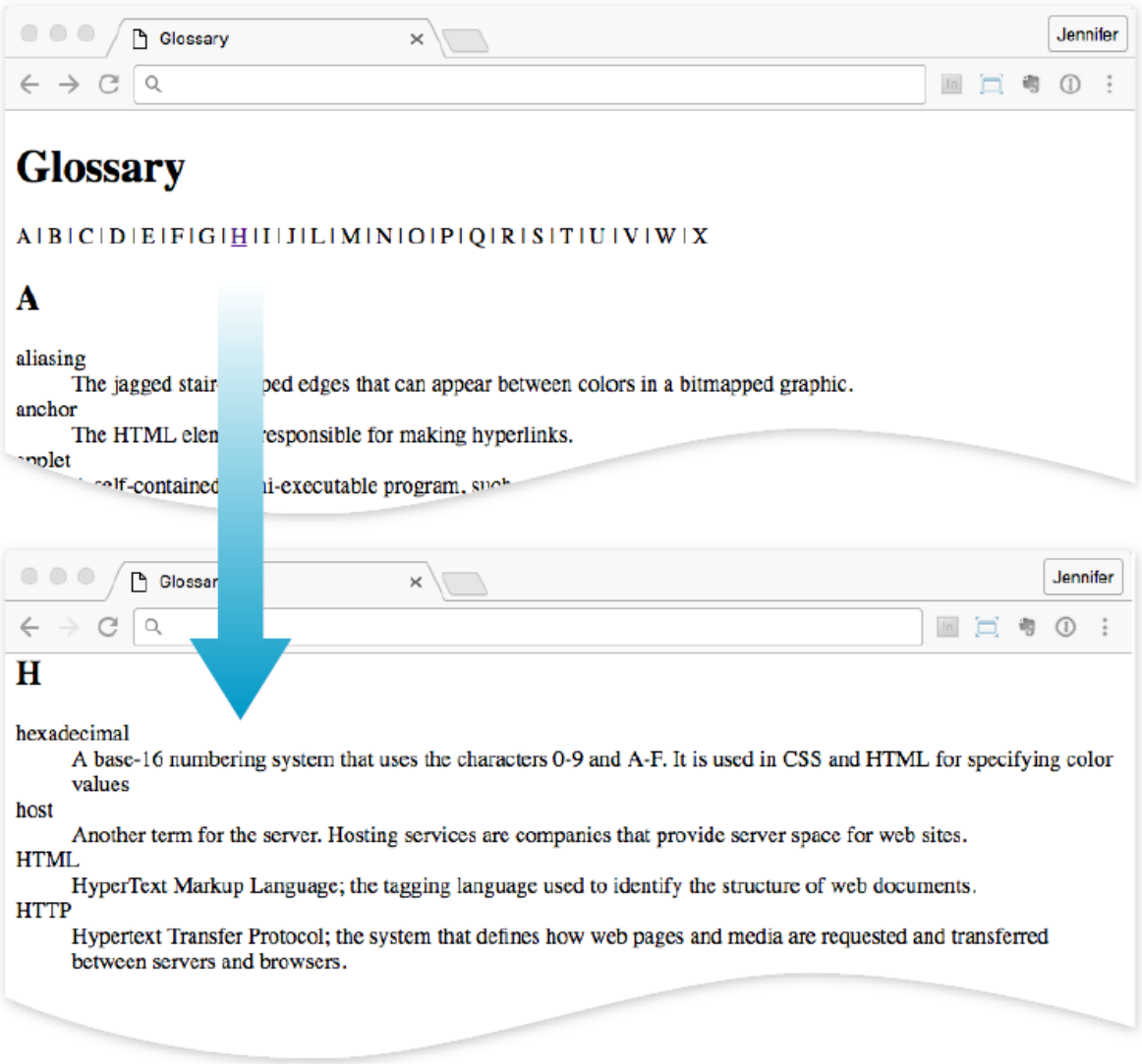


FIGURE 3-11. Linking to a specific destination (a fragment) within a single web page.

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:

See the Glossary, letter H

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:

See the Glossary, letter H

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 3-8 gives you an opportunity to add links to fragments in the example glossary page.

EXERCISE 3-8.

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 3-11.

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

   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:
   TOP

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.

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).

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">

1.4 Adding images

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 4-1, 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 4-1, 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 section.

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.

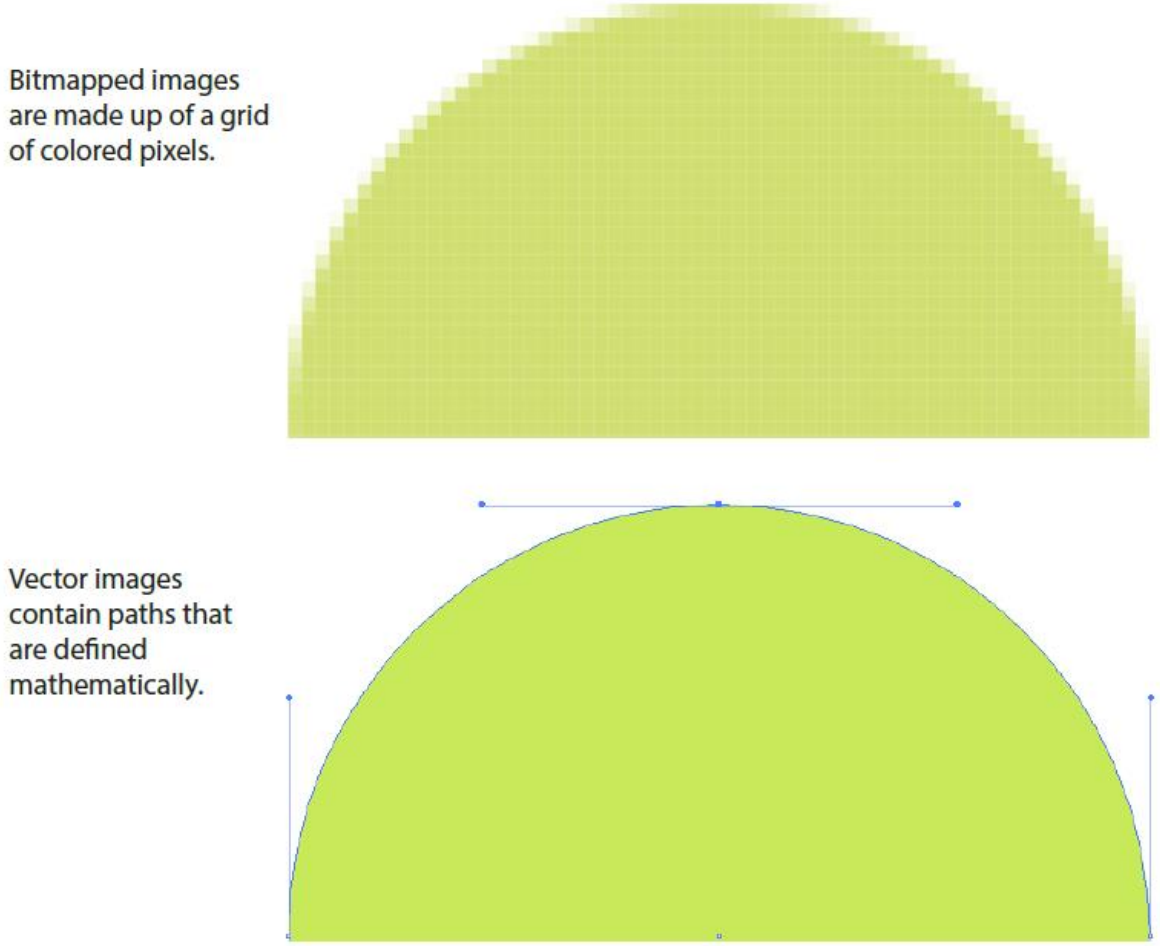


FIGURE 4-1. A comparison of circles saved in bitmapped and vector formats.

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

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, 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 4-2:

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

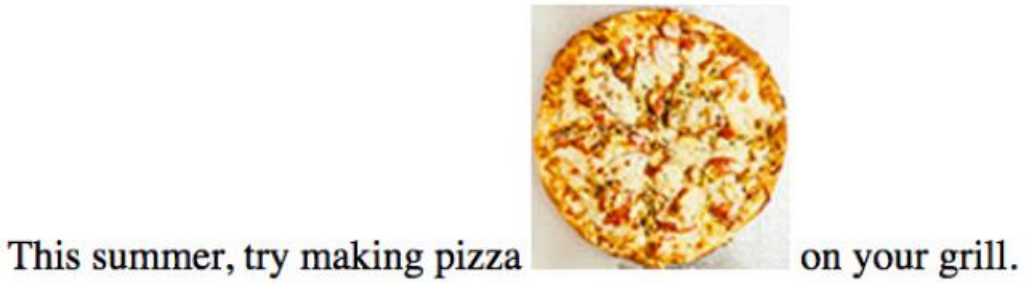


FIGURE 4-2. By default, images are aligned with the baseline of the surrounding text and do not cause a line break.

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 dialup internet connections or other slow networks, like the expensive WiFi at luxury hotels.

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 4-3 demonstrates the inline nature of image elements. When the browser window is resized, a line of images reflows to fill the new width.

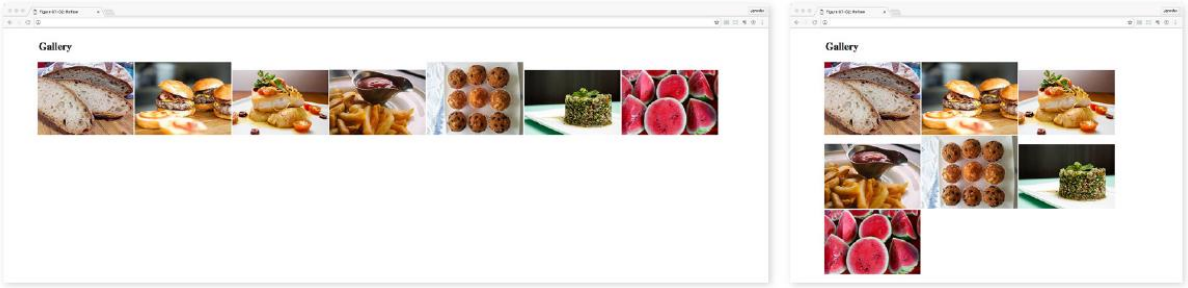


FIGURE 4-3. Inline images are part of the normal document flow. They reflow when the browser window is resized.

- 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 4-2. 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.

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="">

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 4-4).

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:

High school application <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):

<img src="/images/LWD_cover.png"

alt="Learning Web Design site">

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="">

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

With image displayed

If you're  and you know it clap your hands.

Firefox

If you're happy and you know it clap your hands.

Chrome (Mac & Windows)

If you're happy and you know it clap your hands.

MS Edge (Windows)

If you're happy and you know it clap your hands.

Safari (iOS)

If you're happy and you know it clap your hands.

Safari (Mac)

If you're and you know it clap your hands.

FIGURE 4-4. 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.

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 4-4.

Providing the Dimensions with width and height

`width="number"`

Image width in pixels

`height="number"`

Image height in pixels

The width and height attributes indicate the dimensions of the image in number of pixels. 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.

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 section.

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 4-5). 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.

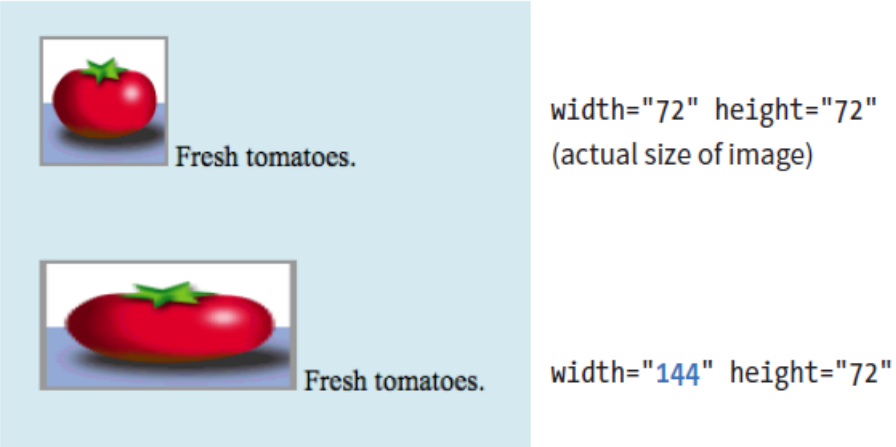


FIGURE 4-5. Browsers resize images to match the provided width and height values, but you should not resize images this way.

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 4-1.

EXERCISE 4-1. 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 4-6). 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 (<br>) 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="closeup
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!

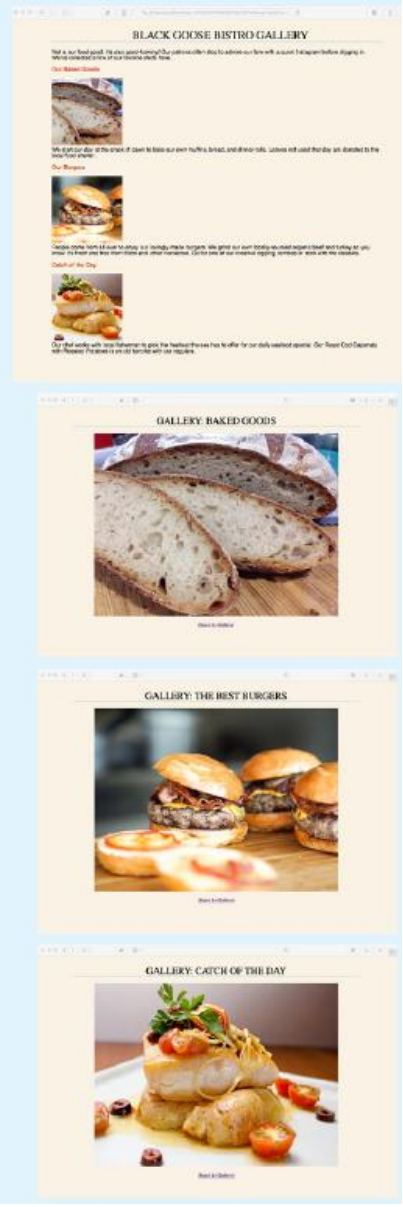


FIGURE 4-6. Photo gallery pages.

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 4-7). 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 4-7. A simple SVG made up of a rectangle and text.

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:

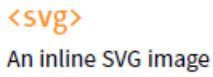
- You cannot apply styles to the items within the SVG by using an external style sheet, such as a .css file applied to the whole page. The .svg file may include its own internal style sheet using the style element, however, for styling the elements within it. You can also apply styles to the img element 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.

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 4-2, only this time our pizza is a vector image drawn with circles and inserted with the svg element (FIGURE 4-8). 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):

```
<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>

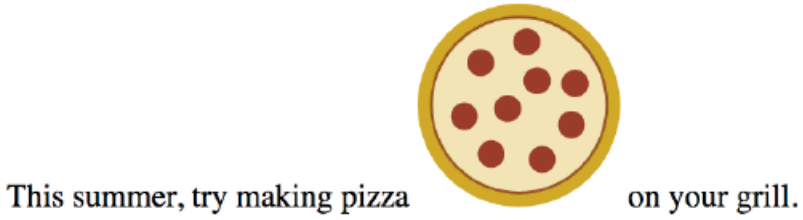


FIGURE 4-8. This pizza image is an SVG made up of 11 circle elements. Instead of an img element, the SVG source code is placed right in the HTML document with an svg element.

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.

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.

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.

Are you ready to give SVGs a spin? Try out some of the embedding techniques we discussed in EXERCISE 4-2.

EXERCISE 4-2. 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 the first section. 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 4-9).

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 4-9. 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>

* Please visit our social media pages *

* *

</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:

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

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 4-9). 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.

1.5 Table markup

HOW TO USE TABLES

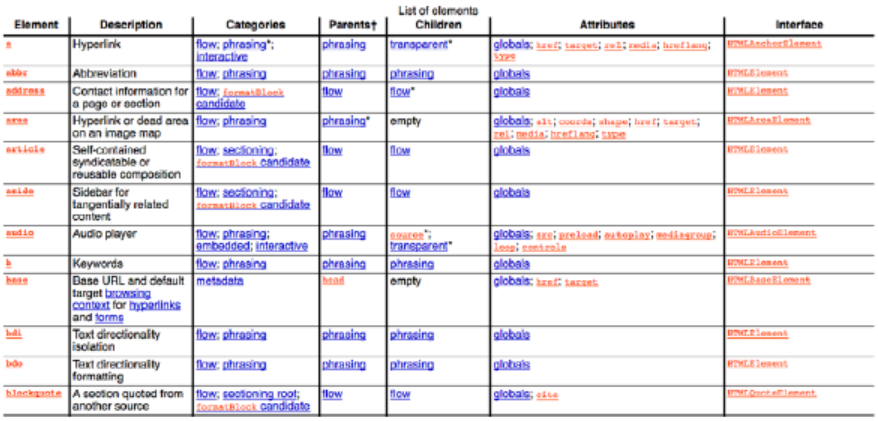
HTML tables were created for instances when you need to add tabular material (data arranged into rows and columns) to a web page. Tables may be used to organize schedules, product comparisons, statistics, or other types of information, as shown in FIGURE 5-1. Note that "data" doesn't necessarily mean 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, 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 divs 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.

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.



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

w3c.org

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

wikipedia.org

mbta.org

FIGURE 5-1. Examples of tables used for tabular information, such as charts, calendars, and schedules.

MINIMAL TABLE STRUCTURE

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

`<tr>…</tr>`
Table row

`<th>…</th>`
Table header

`<td>…</td>`
Table cell data

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

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

FIGURE 5-2 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.

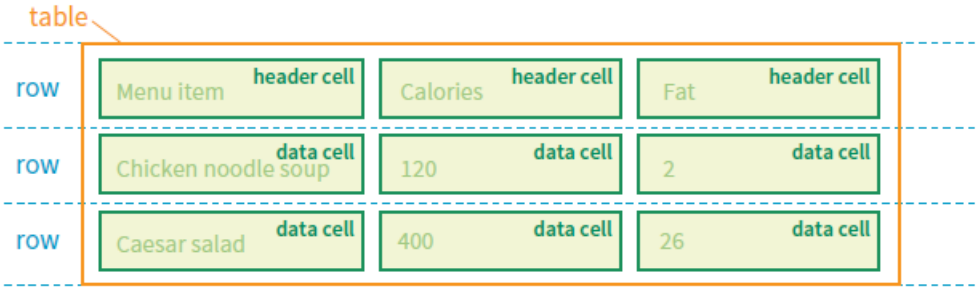


FIGURE 5-2. Tables are made up of rows that contain cells. Cells are the containers for content.

Simple enough, right? Now let's look at how those parts translate into elements (FIGURE 5-3).

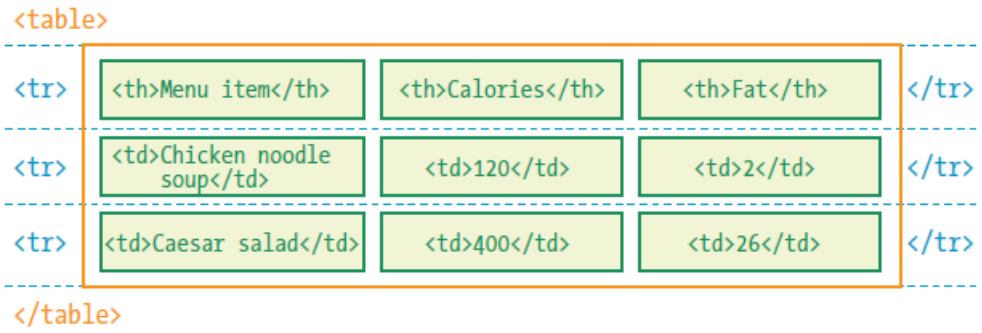


FIGURE 5-3. The elements that make up the basic structure of a table.

FIGURE 5-3 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.

Written out in a source document, the markup for the table in FIGURE 5-3 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.

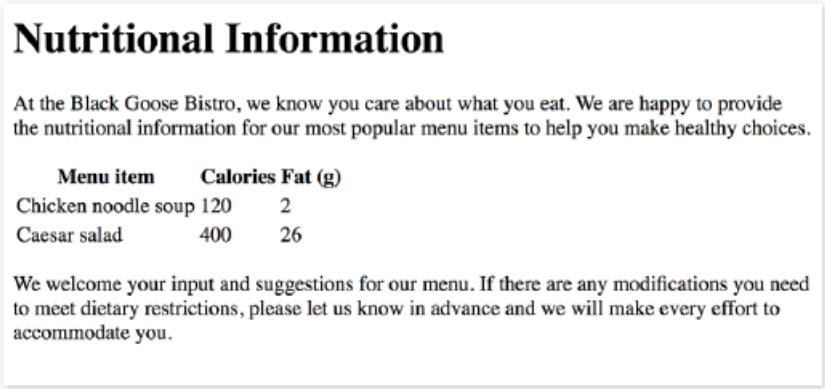
You can put any content in a cell: text, a graphic, or even another table.

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 5-4 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.



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 5-4. 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 5-4, 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 tds 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 5-1.

EXERCISE 5-1.

Making a simple table

Try writing the markup for the table shown in FIGURE 5-5. 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 5-5. Write the markup for this table.

SPANNING CELLS

One fundamental feature of table structure is cell spanning, which is the stretching of a cell to cover several rows or columns. Spanning cells allows you to create complex table structures, but it has the side effect of making the markup a little more difficult to keep track of. 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 5-6).

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 5-6. The colspan attribute stretches a cell to the right to span the specified number of columns.

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

Try your hand at column spanning in EXERCISE 5-2.

EXERCISE 5-2.

Column spans

Try writing the markup for the table shown in FIGURE 5-7. 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 5-1.

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.

FIGURE 5-7. Practice column spans by writing the markup for this table.

Row Spans

Row spans, created with the rowspan attribute, work just like column spans, but they cause the cell to span downward over several rows. In this example, the first cell in the table spans down three rows (FIGURE 5-8).

```
<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 5-3.

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

FIGURE 5-8. 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.

EXERCISE 5-3.

Row spans

Try writing the markup for the table shown in FIGURE 5-9. 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.

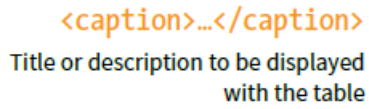
apples	oranges	pears
bananas		pineapple
lychees		

FIGURE 5-9. Practice row spans by writing the markup for this table.

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.

Describing Table Content



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 5-10, 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 5-10. The table caption is displayed above the table by default.

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.

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 5-11 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).

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 5-11 were styled with thicker borders to make them stand out visually

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

'unicode-bidi' value	'direction' value			
	'ltr'		'rtl'	
	start	end	start	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 5-11. 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

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

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

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.

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 5-11:

```
<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.

WRAPPING UP TABLES

This section gave you a good overview of the components of HTML tables.

EXERCISE 5-4 combines most of what we've covered to give you a little more practice at authoring tables.

EXERCISE 5-4. 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 5-12.

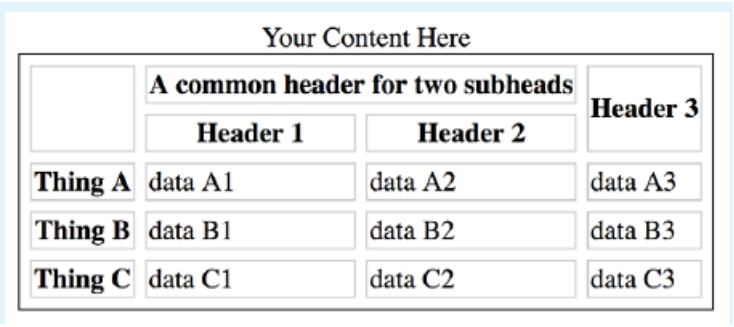


FIGURE 5-12. The table challenge.

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.

1.6 Forms

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 6-1).
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.

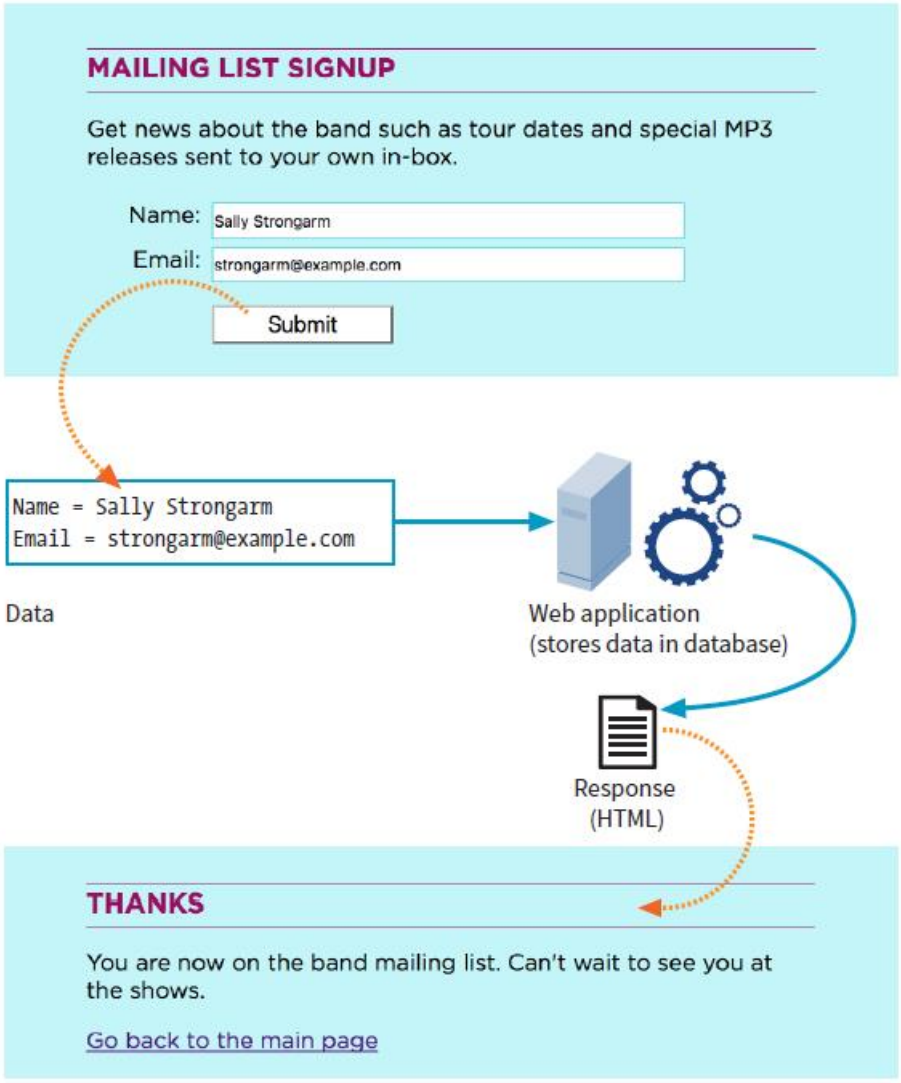


FIGURE 6-1. What happens behind the scenes when a web form is submitted.

THE FORM ELEMENT

`<form>…</form>`

Interactive form

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.

This sample source document contains a form similar to the one shown in FIGURE 6-1:

```
<!DOCTYPE html>
<html>
<head>
    <title>Mailing List Signup</title>
    <meta charset="utf-8">
</head>
<body>
    <h1>Mailing List Signup</h1>
    <form action="/mailinglist.php" method="POST">
        <fieldset>
            <legend>Join our email list</legend>
            <p>Get news about the band such as tour dates and special MP3 releases sent to your own in-box.</p>
            <ol>
                <li><label for="firstlast">Name:</label>
                <input type="text" name="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.

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 6-1 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.

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 section, 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.

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

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.

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 6-2.

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.

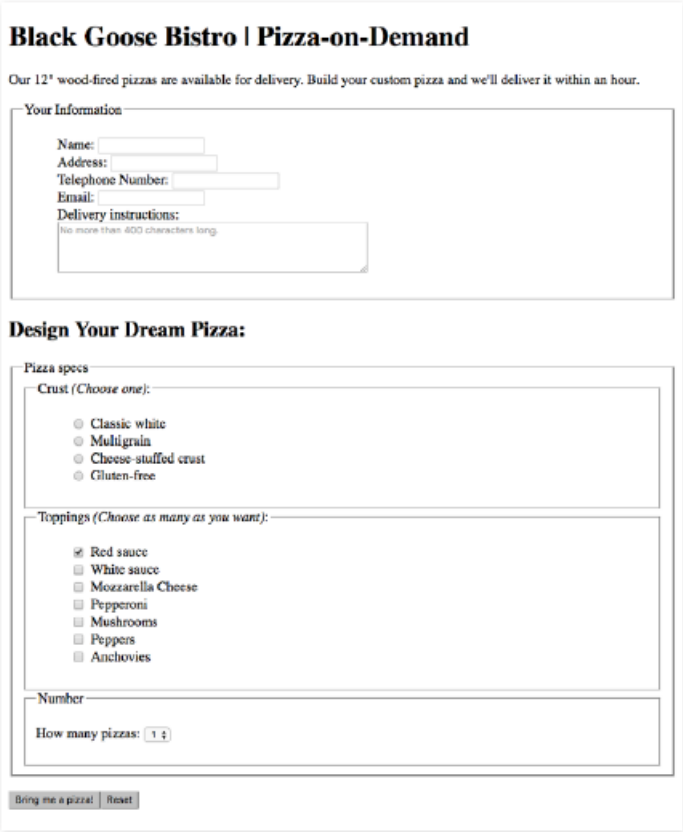


FIGURE 6-2. The pizza ordering form we'll build in the exercises in this section.

Text-Entry Controls

```
<input type="text">
```

Single-line text-entry control

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).

Single-line text field

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 6-3:

*
<label>Favorite color: <input type="text" name="favcolor"*

*value="Red" maxlength="50"></label>
*

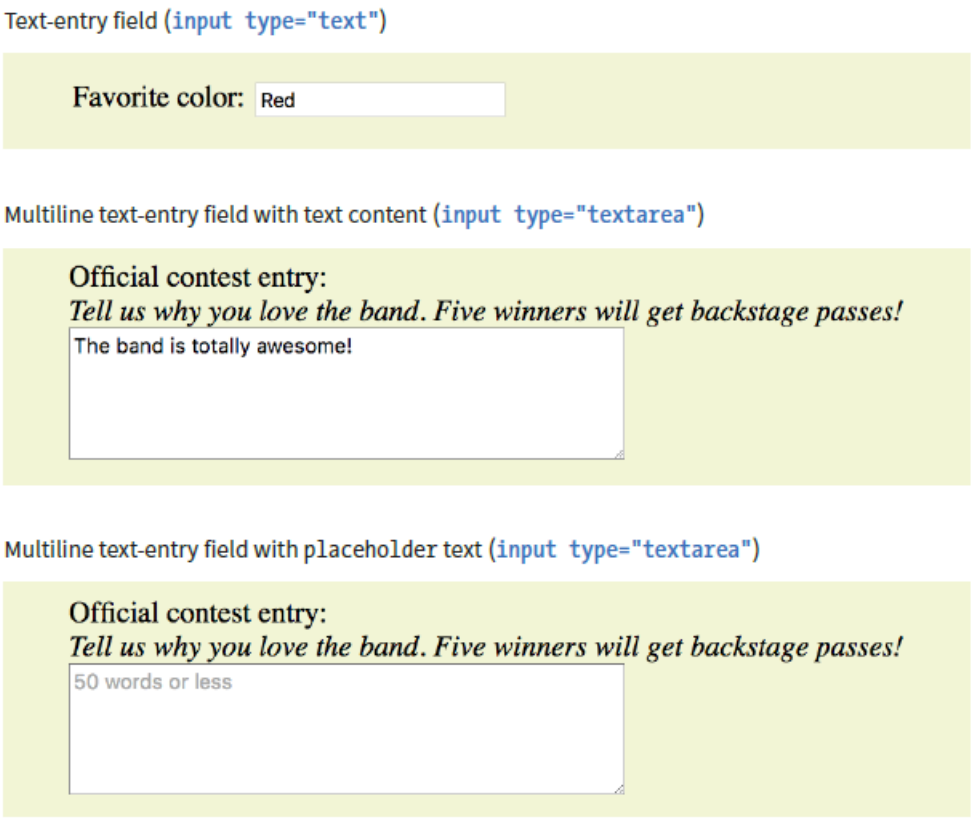


FIGURE 6-3. Examples of the text-entry control options for web forms.

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.

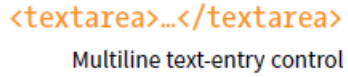
maxlength, minlength

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 exceedsthe 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.

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



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 6-3).

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 6-3.

```
<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>
```

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 6-4 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 6-4. 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 6-1 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 6-5). 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 6-6) 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.

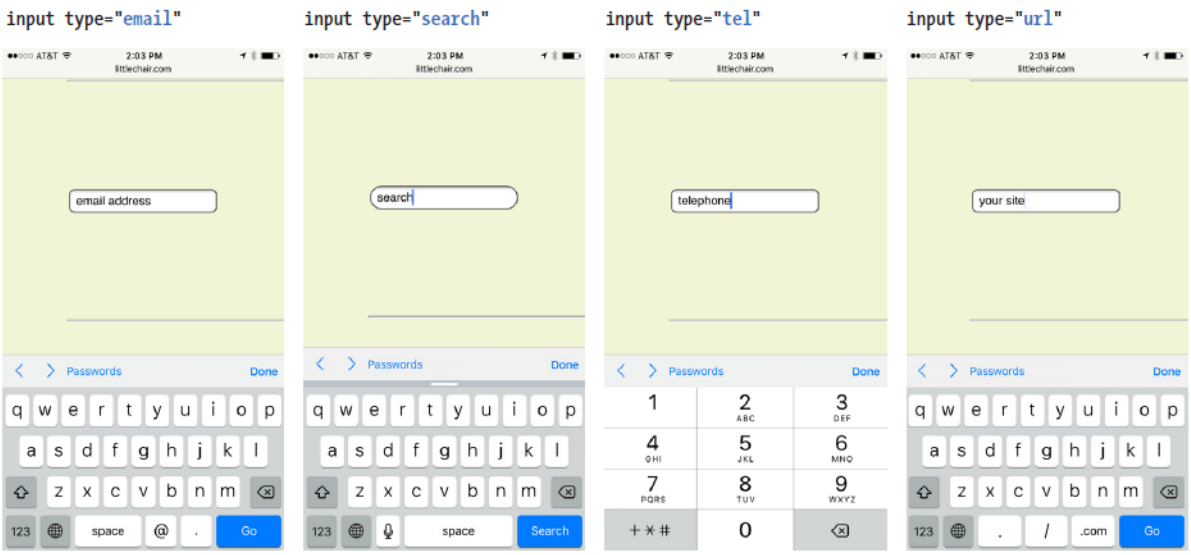


FIGURE 6-5. Safari on iOS provides custom keyboards based on the input type.

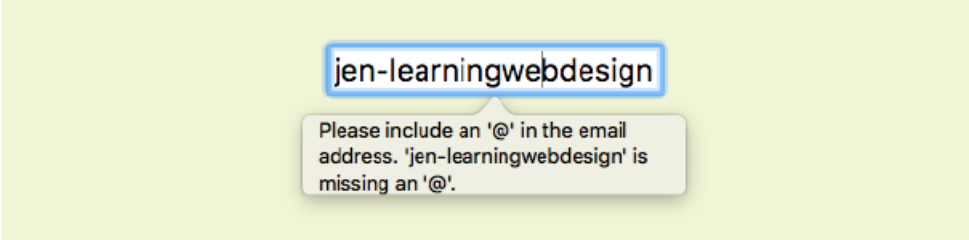


FIGURE 6-6. Opera displays a warning when input does not match the expected email format as part of its client-side validation support.

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 6-7), 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.

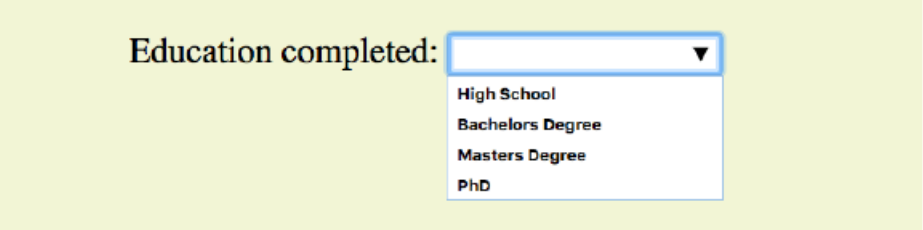


FIGURE 6-7. A datalist creates a pop-up menu of suggested values for a textentry field.

Submit and Reset Buttons

```
<input type="submit">
```
Submits the form data to the server

```
<input type="reset">
```
Resets the form controls to their default settings

There are several 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 6-8).

* <input type="submit"> <input type="reset" value="Start over"> *

FIGURE 6-8. 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.

EXERCISE 6-1 walks you through the first steps.

EXERCISE 6-1. 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 6-9) 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.

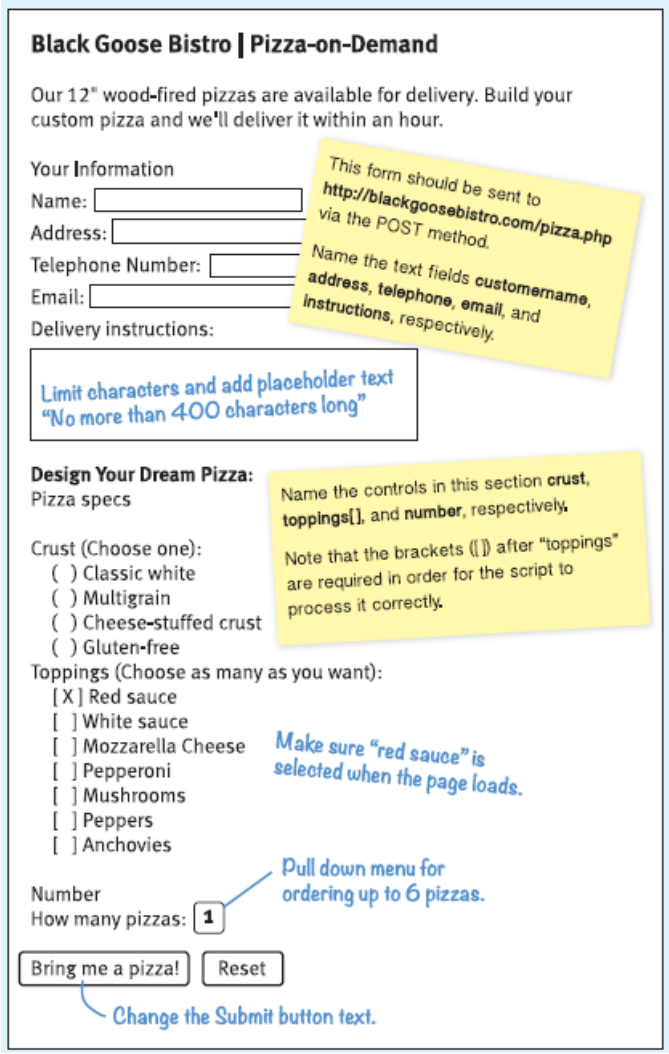


FIGURE 6-9. A sketch of the Black Goose Bistro pizza ordering form.

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">

   *
*

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 finetuned control):

   *<li>Delivery instructions:
*

   <textarea name="instructions" rows="4" cols="60"

   maxlength="400" placeholder="No more than 400

   *characters long"></textarea>
*

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.

   <input type="submit" value="Bring me a pizza!"><input type="reset">

6. Now, save the document and open it in a browser. The parts that are finished should generally match FIGURE 6-2. 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 6-10. Yes, pizza.php actually works, but sorry, no pizzas will be delivered.

FIGURE 6-10. 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

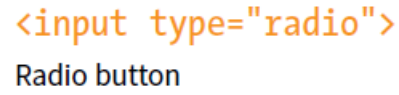
Both checkbox and radio buttons make it simple for your visitors to choose from a number of provided options. They are similar in that they function like little on/off switches that can be toggled by the user and are added with the input element. They serve distinct functions, however.

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.

Radio buttons



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.

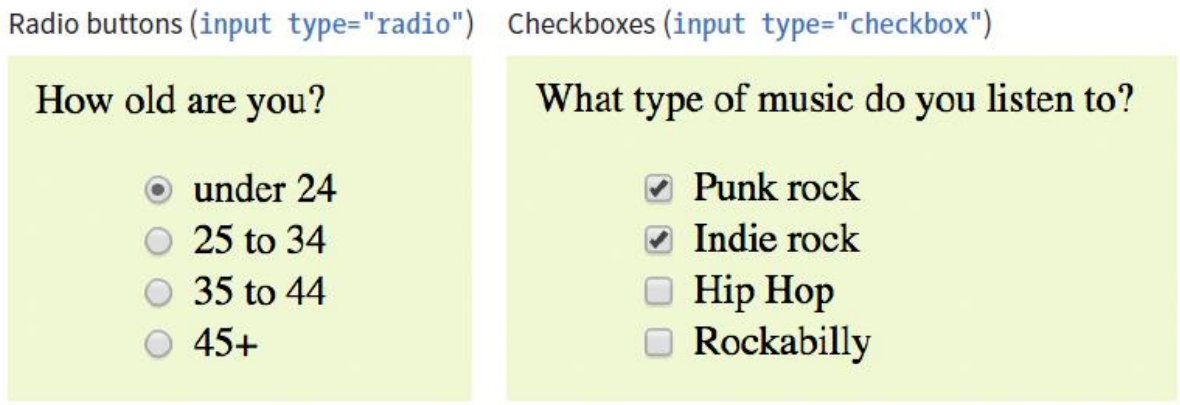
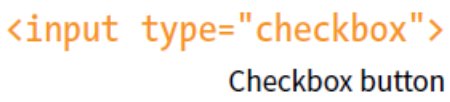


FIGURE 6-11. Radio buttons (left) are appropriate when only one selection is permitted. Checkboxes (right) are best when users may choose any number of choices, from none to all of them.

Checkbox buttons



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 6-11 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
*

*
*

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.

* <input type="checkbox" name="OptIn" value="yes"> Yes, send me news*

*and special promotions by email. *

Checkbox buttons also use the checked attribute to make them preselected when the form loads.

In EXERCISE 6-2, you'll get a chance to add both radio and checkbox buttons to the pizza ordering form.

EXERCISE 6-2. 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:

   *
<input type="radio" name="crust" value="white"> Classic white
*

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 6-12):

```
<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>
```

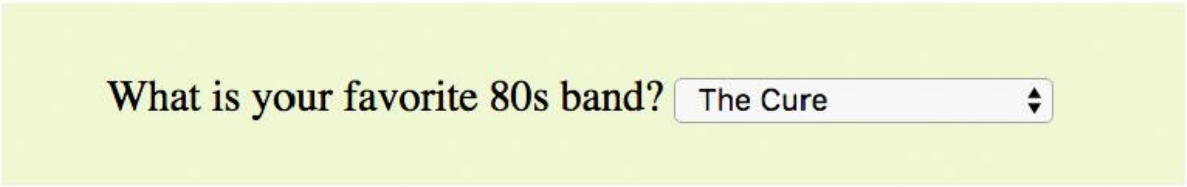


FIGURE 6-12. Pull-down menus pop open when the user clicks the arrow or bar.

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 6-13):

```
<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>
```

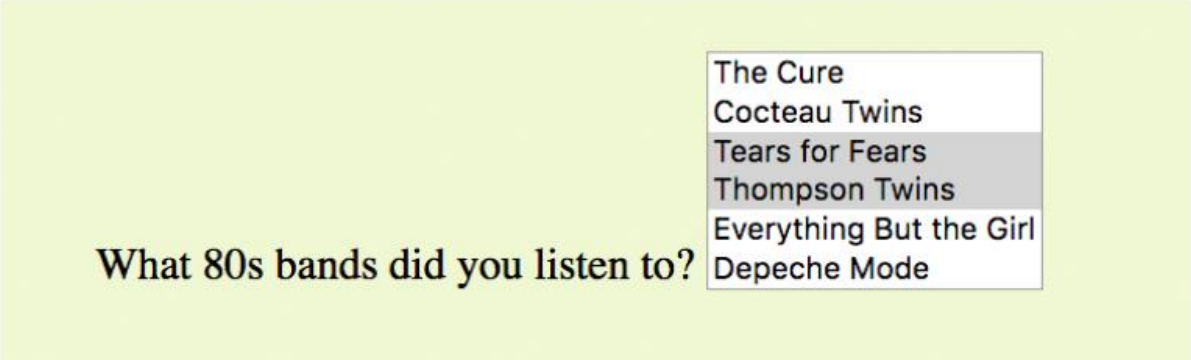


FIGURE 6-13. A scrolling menu with multiple options selected.

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 6-14 shows how option groups are rendered in modern browsers.

```
<select name="icecream" size="7" multiple>
    <optgroup label="traditional">
        <option>vanilla</option>
        <option>chocolate</option>
    </optgroup>
    <optgroup label="fancy">
        <option>Super praline</option>
        <option>Nut surprise</option>
        <option>Candy corn</option>
    </optgroup>
</select>
```

FIGURE 6-14. Option groups.

In EXERCISE 6-3, 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 6-15) 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 6-15).

File input (on Chrome browser)

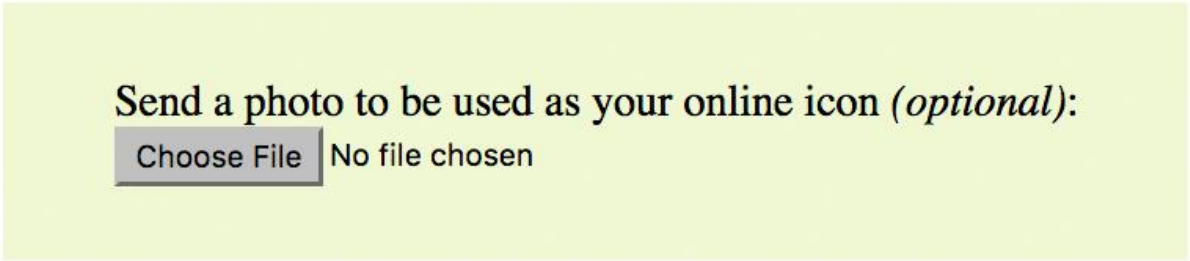


FIGURE 6-15. A file selection form field.

EXERCISE 6-3.

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 6-4, 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.

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 6-16 shows date and time widgets as rendered in Chrome on macOS.

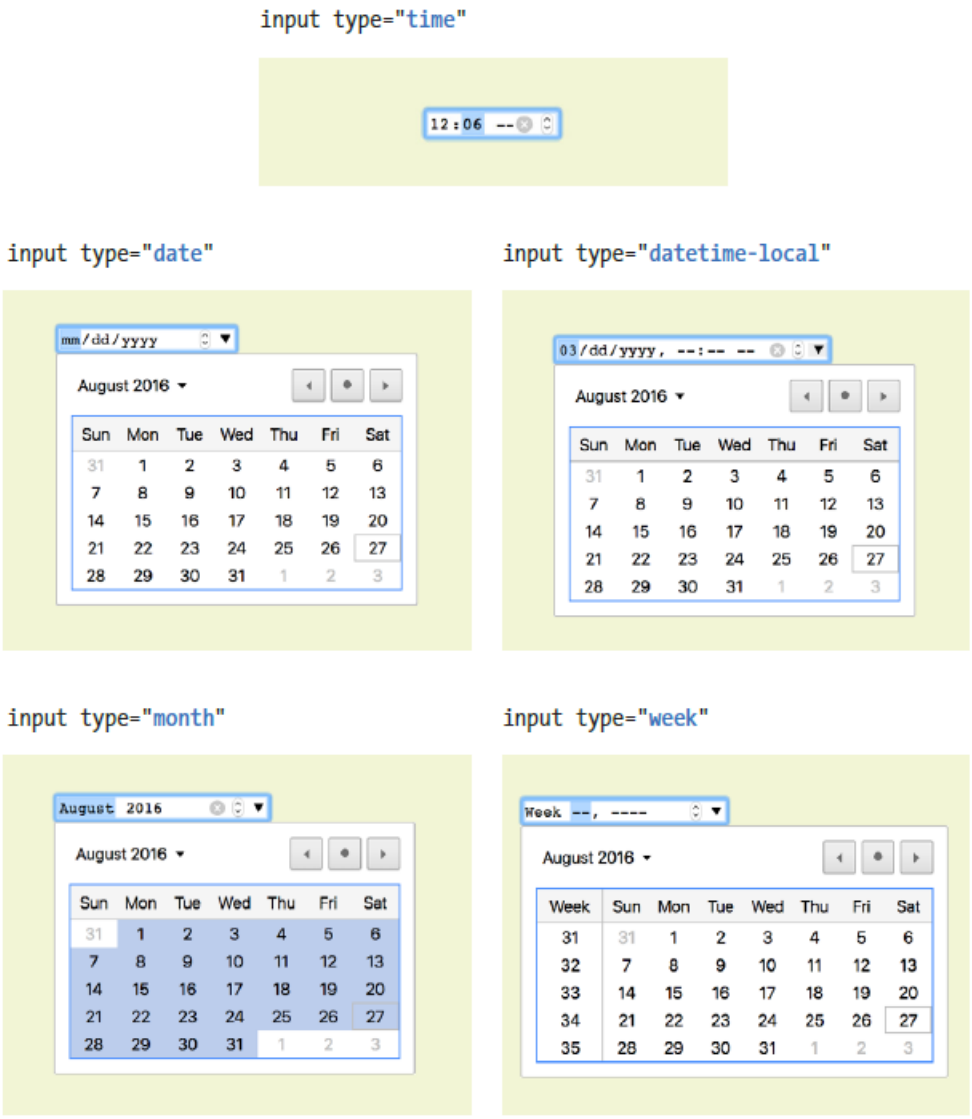


FIGURE 6-16. Date and time picker inputs (shown in Chrome on macOS).

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).

<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"

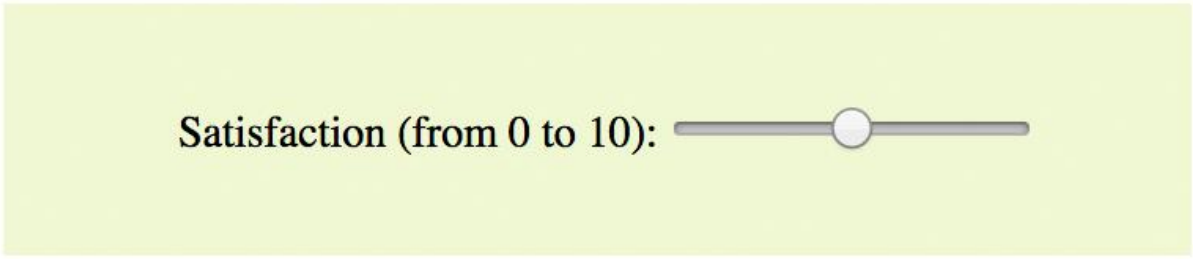


FIGURE 6-17. The number and range input types (shown in Chrome on macOS).

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 6-18 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>

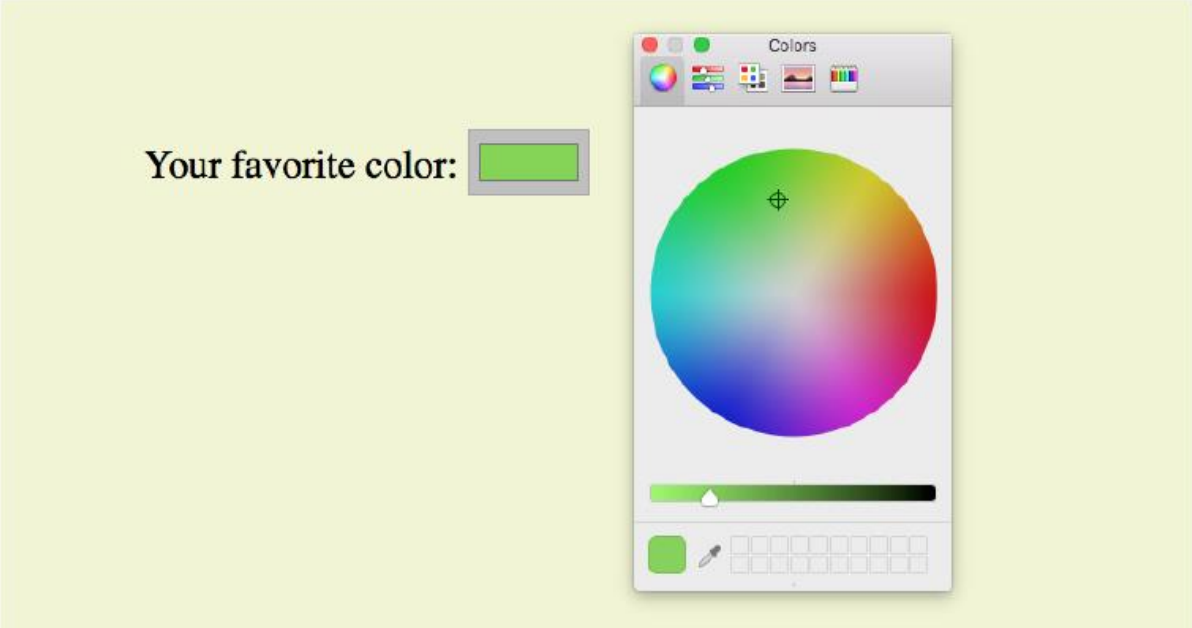


FIGURE 6-18. The color input type (shown in Chrome on macOS).

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.

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.

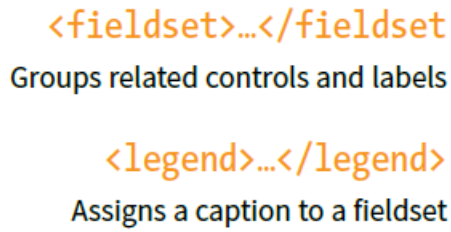
Each label element is associated with exactly one form control. There are two ways to use it. One method, called implicit association, nests the control and its description within a label element. In the following example, labels are assigned to individual checkboxes and their related text descriptions. (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">
```

fieldset and legend



The fieldset element indicates a logical group of form controls. A fieldset may also include a legend element that provides a caption for the enclosed fields.

FIGURE 6-19 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>
```

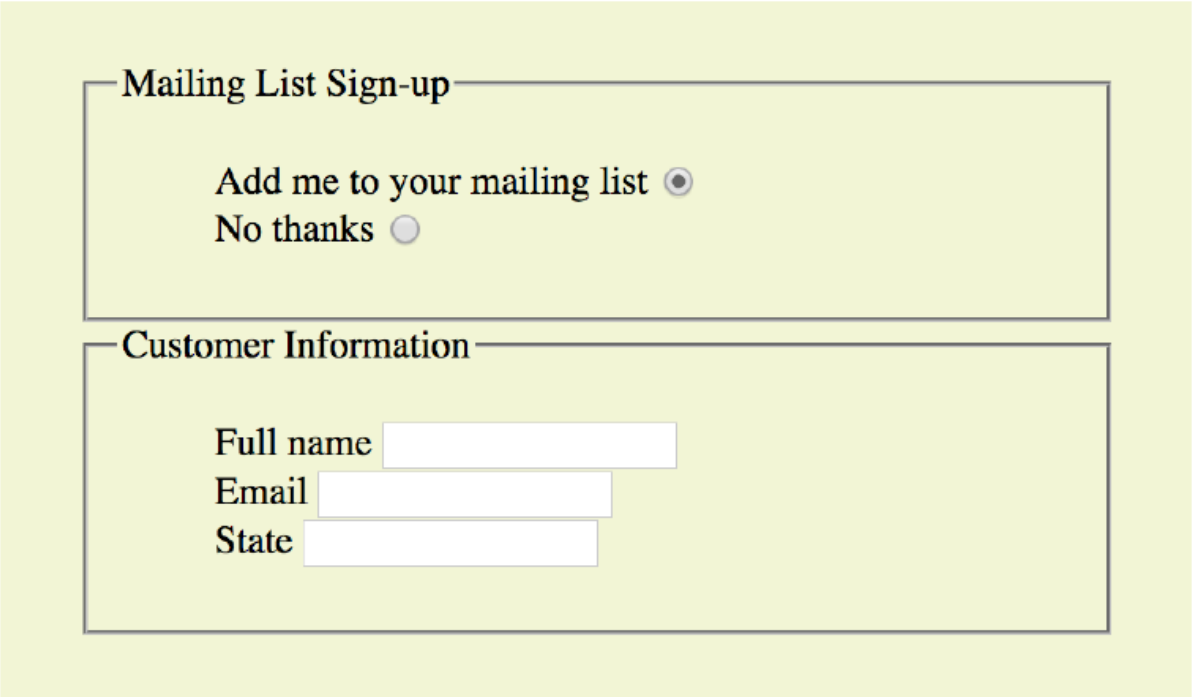


FIGURE 6-19. The default rendering of fieldsets and legends.

In EXERCISE 6-4, we'll wrap up the pizza order form by making it more accessible with labels and fieldsets.

EXERCISE 6-4. 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">

   *
*

   …

   *
*

</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 6-2, given the expected browser differences:

   <fieldset>

   <legend>Crust (Choose one):</legend>

   *
…
*

   </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:

   *
<label for="form-name">Name:</label> <input*

   *type="text" name="fullname" id="form-name">
*

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:

   *
<label><input type="radio" name="crust"*

   *value="white"> Classic White</label>
*

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.

1.7 Embedded media

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 7-1). 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>
```

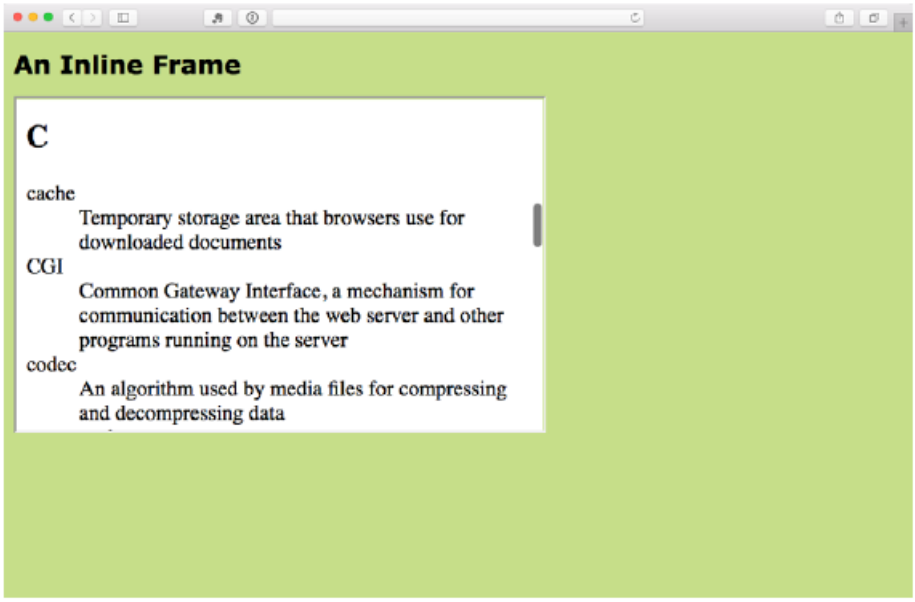


FIGURE 7-1. Inline frames (added with the iframe element) are like a browser window within the browser that displays external HTML documents and resources.

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 7-2.

FIGURE 7-2. 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.

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.

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 royaltyfree.

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, royaltyfree 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 7-1 lists which browsers support the various video options (see the “Server Setup” sidebar).

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 7-1. Video support in desktop and mobile browsers (as of 2017)

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 7-2).

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.

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+	–	–	–

TABLE 7-2. Audio support in current browsers (as of 2017)

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 7-3 shows a movie with the default player in the Chrome browser.

FIGURE 7-3. An embedded movie using the video element (shown in Chrome on a Mac).

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

</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 builtin 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>;
```

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 7-4 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 7-4. Audio player as rendered in Firefox.

EXERCISE 8-2. 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 8-1.
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.

2. CSS (Cascading Style Sheets)

You've heard style sheets mentioned quite a bit already, and now we'll finally put them to work and start giving our pages some much-needed style. Cascading Style Sheets (CSS) is the W3C standard for defining the presentation of documents written in HTML, and in fact, any XML language.

Presentation, again, refers to the way the document is delivered to the user, whether shown on a computer screen, displayed on a cell phone, printed on paper, or read aloud by a screen reader. With style sheets handling the presentation, HTML can handle the business of defining document structure and meaning, as intended.

CSS is a separate language with its own syntax. This chapter covers CSS terminology and fundamental concepts that will help you get your bearings for the upcoming chapters, where you'll learn how to change text and font styles, add colors and backgrounds, and even do basic page layout.

THE BENEFITS OF CSS

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

- Precise type and layout controls. You can achieve print-like precision using CSS. There is even a set of properties aimed specifically at the printed page (but we won't be covering them in this book).
- Less work. You can change the appearance of an entire site by editing one style sheet. This also ensures consistency of formatting throughout the site.
- More accessible sites. When all matters of presentation are handled by CSS, you can mark up your content meaningfully, making it more accessible for non-visual or mobile devices.

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

HOW STYLE SHEETS WORK

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

1. Start with a document that has been marked up in HTML.
2. Write style rules for how you ' d like certain elements to look.

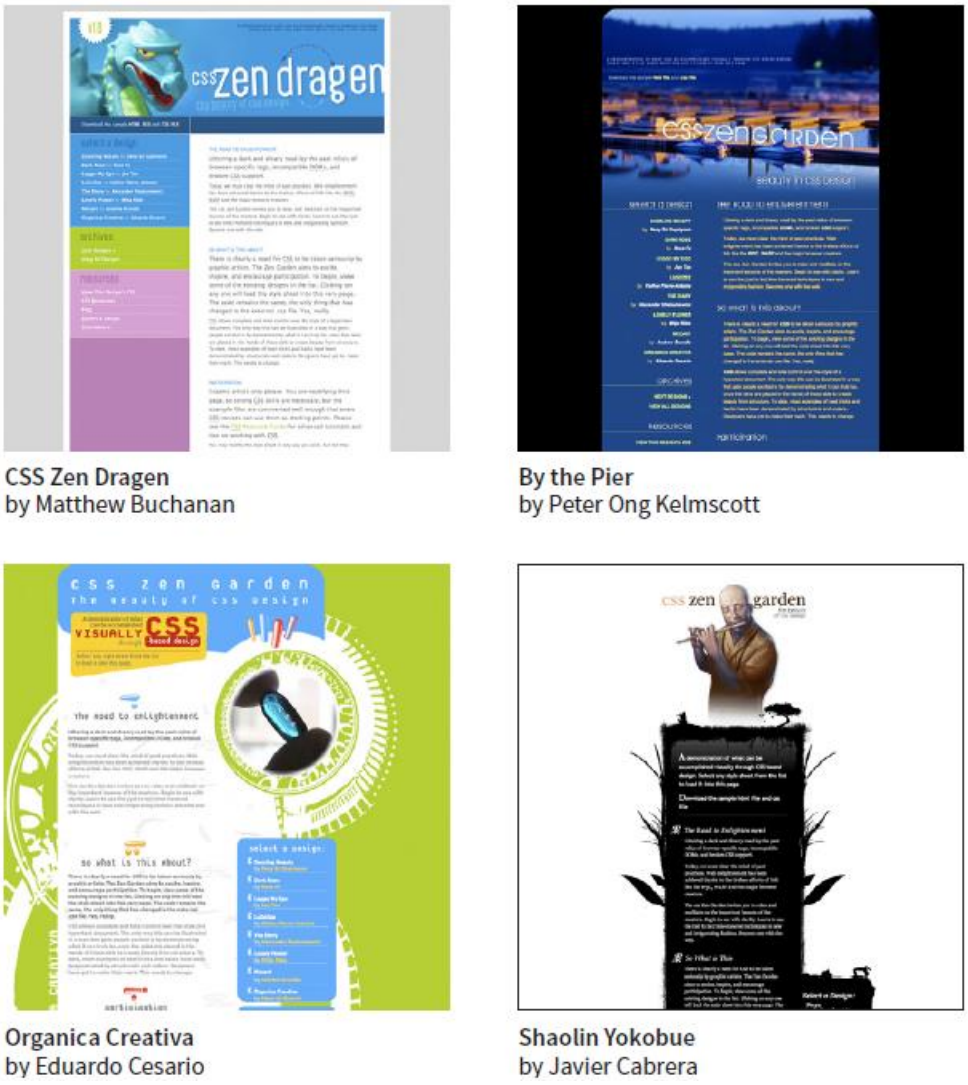


FIGURE 1-1. These pages from the CSS Zen Garden use the same HTML source document, but the design is changed with CSS alone (used with permission of CSS Zen Garden and the individual designers).

3. Attach the style rules to the document. When the browser displays the document, it follows your rules for rendering elements (unless the user has applied some mandatory styles, but we'll get to that later).

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

1. Marking Up the Document
   You know a lot about marking up content from the previous chapters. For example, you know that it is important to choose elements that accurately describe the meaning of the content. You also heard me say that the markup creates the structure of the document, sometimes called the structural layer, upon which the presentation layer can be applied.
   In this and the upcoming chapters, you'll see that having an understanding of your document's structure and the relationships between elements is central to your work as a style sheet author.

In the exercises throughout this chapter you will get a feel for how simple it is to change the look of a document with style sheets. The good news is that I've whipped up a little HTML document for you to play with. You can get acquainted with the page we'll be working with in EXERCISE 1-1.

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

   The following example contains two rules. The first makes all the h1 elements in the document green; the second specifies that the paragraphs should be in a large, sans-serif font. Sans-serif fonts do not have a little slab (a serif) at the ends of strokes and tend to look more sleek and modern.

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

   In CSS terminology, the two main sections of a rule are the selector that identifies the element or elements to be affected, and the declaration that provides the rendering instructions. The declaration, in turn, is made up of a property (such as color) and its value (green), separated by a colon and a space. One or more declarations are placed inside curly brackets, as shown in FIGURE 1-3.

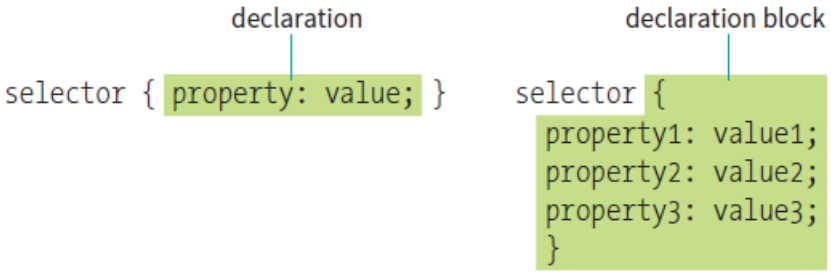


FIGURE 1-3. The parts of a style rule.

EXERCISE 1-1.

A first look, In this section, we'll add a few simple styles to a short article. The document, cooking.html, and its associated image, salads.jpg, are available at learningwebdesign.com/5e/materials/.

For now, just open the document in a browser to see how it looks by default (it should look something like FIGURE 1-2). You can also open the document in a text editor to get ready to follow along in the next two exercises.

FIGURE 1-2. This is what the article looks like without any style sheet instructions. Although we will not be making it beautiful, you will get a feel for how style sheets work.

Selectors

In the previous small style sheet example, the h1 and p elements are used as selectors. This is called an element type selector, and it is the most basic type of selector. The properties defined for each rule will apply to every h1 and p element in the document, respectively.

Another type of selector is an ID selector, which selects an element based on the value of an element's id attribute. It is indicated with the # symbol. For example, the selector #recipe targets an element with id="recipe".

In upcoming chapters, I'll introduce you to more sophisticated selectors that you can use to target elements, including ways to select groups of elements, and elements that appear in a particular context. See the "Selectors in this Book" sidebar for details.

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

Declarations

The declaration is made up of a property/value pair. There can be more than one declaration in a single rule; for example, the rule for the p element shown earlier in the code example has both the font-size and font-family properties.

Each declaration must end with a semicolon to keep it separate from the following declaration (see Note). If you omit the semicolon, the declaration and the one following it will be ignored. The curly brackets and the declarations they contain are often referred to as the declaration block (FIGURE 1-3).

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

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

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

Properties

The heart of style sheets lies in the collection of standard properties that can be applied to selected elements. The complete CSS specification defines dozens of properties for everything from text indents to how table headers should be read aloud. This book covers the most common and best-supported properties that you can begin using right away.

Values

Values are dependent on the property. Some properties take length measurements, some take color values, and others have a predefined list of keywords.

When you use a property, it is important to know which values it accepts; however, in many cases, simple common sense will serve you well. Authoring tools such as Dreamweaver or Visual Studio provide hints of suitable values to choose from. Before we move on, why not get a little practice writing style rules yourself in EXERCISE 1-2?

EXERCISE 1-2. Your first style sheet

Open cooking.html in a text editor. In the head of the document you will find that I have set up a style element for you to type the rules into. The style element is used to embed a style sheet in an HTML document. To begin, we'll simply add the small style sheet that we just looked at in this section. Type the following rules into the document, just as you see them here:

```
<style>
    h1 {
        color: green;
    }
    p {
        font-size: large;
        font-family: sans-serif;
    }
</style>
```

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

Now we'll edit the style sheet to see how easy it is to write rules and see the effects of the changes. Here are a few things to try.

IMPORTANT: Remember that you need to save the document after each change in order for the changes to be visible when you reload it in the browser.

- Make the h1 element "gray" and take a look at it in the browser.

  Then make it "blue". Finally, make it "orange".

- Add a new rule that makes the h2 elements orange as well.
- Add a 100-pixel left margin to paragraph (p) elements by using this declaration:

  margin-left: 100px;

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

- Add an orange, 1-pixel border to the bottom of the h1 element by using this declaration:

  border-bottom: 1px solid orange;

- Move the image to the right margin, and allow text to flow around it with the float property. The shorthand margin property shown in this rule adds zero pixels of space on the top and bottom of the image and 12 pixels of space on the left and right of the image:

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

When you are done, the document should look something like the one shown in FIGURE 1-4.

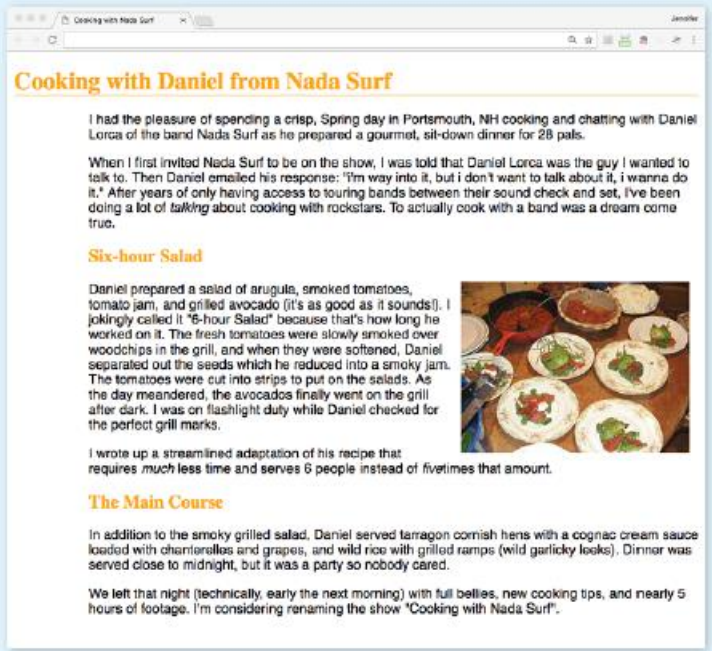


FIGURE 1-4. The article after we add a small style sheet. Not beautiful—just different.

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

   External style sheets

   An external style sheet is a separate, text-only document that contains a number of style rules. It must be named with the .css suffix. The .css document is then linked to (via the link element) or imported (via an @import rule in a style sheet) into one or more HTML documents. In this way, all the files in a website may share the same style sheet. This is the most powerful and preferred method for attaching style sheets to content.

   Embedded style sheets

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

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

Inline styles

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

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

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

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

Inline styles apply only to the particular element in which they appear.

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

EXERCISE 1-3 gives you an opportunity to write an inline style and see how it works. We won't be working with inline styles after this point for the reasons listed earlier, so here's your chance.

Comments in Style Sheets

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

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

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

```
body {
  font-size: small;
 /* change this later */
}
```

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

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

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

EXERCISE 1-3. Applying an inline style

Open the article cooking.html in whatever state you last left it in EXERCISE 1-2. If you worked to the end of the exercise, you will have a rule that makes the h2 elements orange.

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

<h2 style="color: gray">The

Main Course</h2>

Note that it must be gray-with-an-a (not grey-with-an-e) because that is the way the color is defined in the spec.

Save the file and open it in a browser.

Now the second heading is gray, overriding the orange color set in the embedded style sheet. The other h2 heading is unaffected.

THE BIG CONCEPTS

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

Inheritance

Are your eyes the same color as your parents'? Did you inherit their hair color? Well, just as parents pass down traits to their children, styled HTML elements pass down certain style properties to the elements they contain.

Notice in EXERCISE 1-1, when we styled the p elements in a large, sans-serif font, the em element in the second paragraph became large and sans-serif as well, even though we didn't write a rule for it specifically (FIGURE 1-5). That is because the em element inherited the styles from the paragraph it is in. Inheritance provides a mechanism for styling elements that don't have any explicit styles rules of their own.

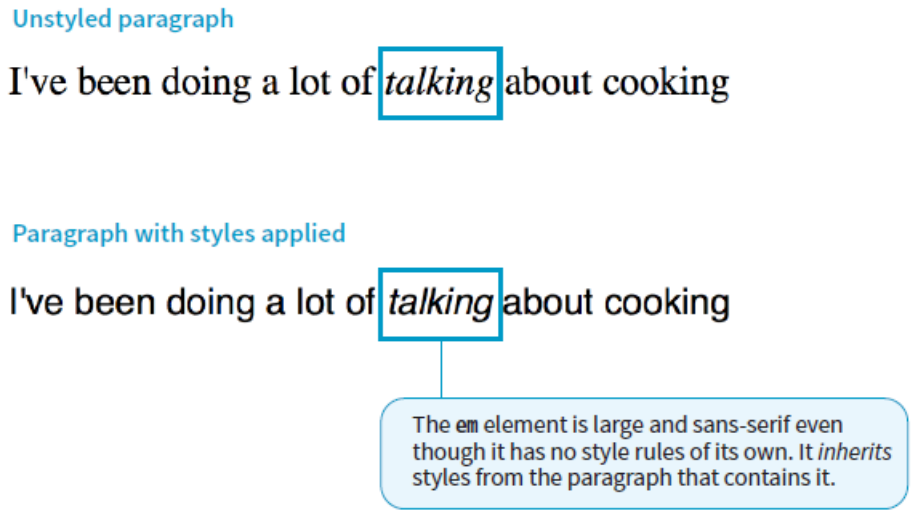


FIGURE 1-5. The em element inherits styles that were applied to the paragraph.

Document structure

This is where an understanding of your document's structure becomes important. As I've noted before, HTML documents have an implicit structure, or hierarchy. For example, the sample article we've been playing with has an html root element that contains a head and a body, and the body contains heading and paragraph elements. A few of the paragraphs, in turn, contain inline elements such as images (img) and emphasized text (em). You can visualize the structure as an upside-down tree, branching out from the root, as shown in FIGURE 1-6.

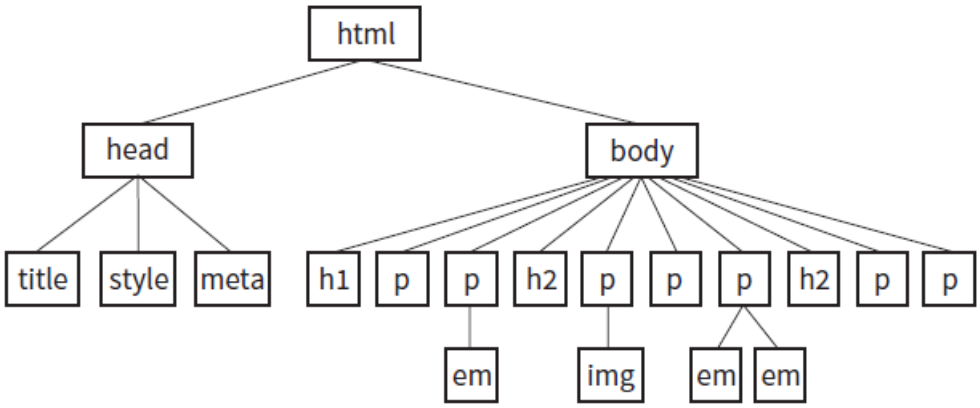


FIGURE 1-6. The document tree structure of the sample document, cooking.html.

Parents and children

The document tree becomes a family tree when it comes to referring to the relationship between elements. All the elements contained within a given element are said to be its descendants. For example, the h1, h2, p, em, and img elements in the document in FIGURE 1-6 are all descendants of the body element.

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

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

Pass it on

When you write a font-related style rule using the p element as a selector, the rule applies to all of the paragraphs in the document as well as the inline text elements they contain. We've seen the evidence of the em element inheriting the style properties applied to its parent (p) back in FIGURE 1-5. FIGURE 1-7 demonstrates what's happening in terms of the document structure diagram.

Note that the img element is excluded because font-related properties do not apply to images.

Notice that I've been saying "certain" properties are inherited. It's important to note that some style sheet properties inherit and others do not. In general, properties related to the styling of text—font size, color, style, and the like—are passed down. Properties such as borders, margins, backgrounds, and so on that affect the boxed area around the element tend not to be passed down.

This makes sense when you think about it. For example, if you put a border around a paragraph, you wouldn't want a border around every inline element (such as em, strong, or a) it contains as well.

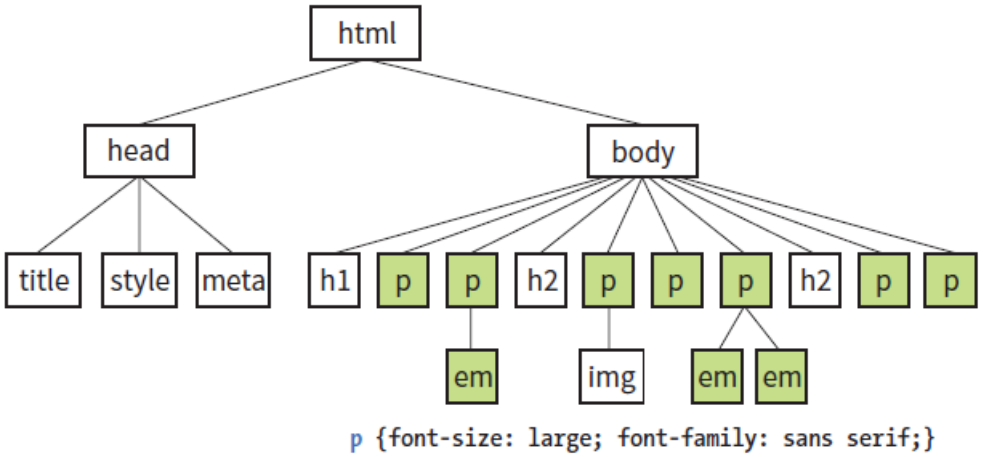


FIGURE 1-7. Certain properties applied to the p element are inherited by their children.

You can use inheritance to your advantage when writing style sheets. For example, if you want all text elements to be blue, you could write separate style rules for every element in the document and set the color to "blue". A better way would be to write a single style rule that applies the color property to the body element, and let all the elements contained in the body inherit that style (FIGURE 1-8).

Any property applied to a specific element overrides the inherited values for that property. Going back to the article example, if we specify that the em element should be orange, that would override the inherited blue setting.

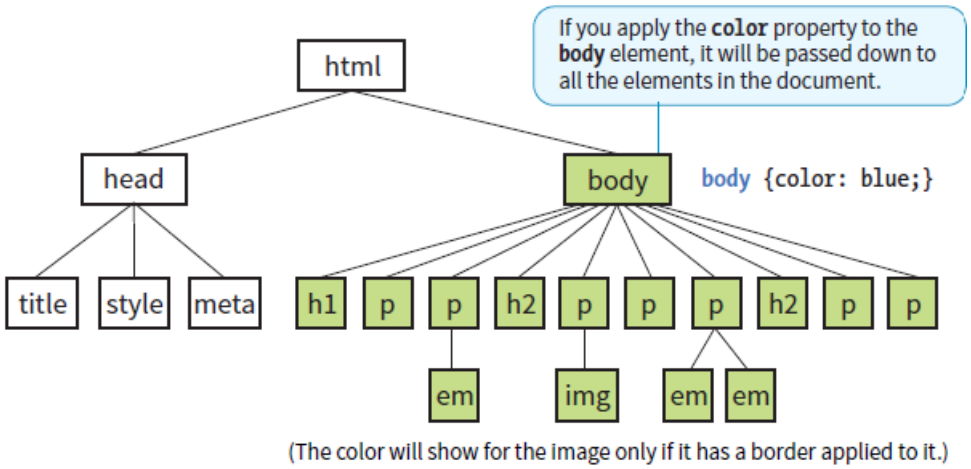


FIGURE 1-8. All the elements in the document inherit certain properties applied to the body element.

Conflicting Styles: The Cascade

Ever wonder why they are called “cascading” style sheets? CSS allows you to apply several style sheets to the same document, which means there are bound to be conflicts. For example, what should the browser do if a document’s imported style sheet says that h1 elements should be red, but its embedded style sheet has a rule that makes h1s purple? The two style rules with h1 selectors have equal weight, right?

The folks who wrote the style sheet specification anticipated this problem and devised a hierarchical system that assigns different weights to the various sources of style information. The cascade refers to what happens when several sources of style information vie for control of the elements on a page: style information is passed down (“cascades” down) until it is overridden by a style rule with more weight. Weight is considered based on the priority of the style rule source, the specificity of the selector, and rule order.

Priority

If you don’t apply any style information to a web page, it renders according to the browser’s internal style sheet. We’ve been calling this the default rendering; the W3C calls it the user agent style sheet. Individual users can apply their own styles as well (the user style sheet, also called the reader style sheet), which override the default styles in their browser. However, if the author of the web page has attached a style sheet (the author style sheet), that overrides both the user and the user agent styles. The sidebar “Style Rule Hierarchy” provides an overview of the cascading order from highest to lowest priority.

The only exception is if the user has identified a style as “important,” in which case that style will override all competing styles (see the “Assigning Importance” sidebar). This permits users to keep settings accommodating a disability such as extra large type for sight impairment.

Specificity

It is possible for conflicts to arise in which an element is getting style instructions from more than one rule. For example, there may be a rule that applies to paragraphs and another rule for a paragraph that has the ID “intro.” Which rule should the intro paragraph use?

When two rules in a style sheet conflict, the type of selector is used to determine the winner. The more specific the selector, the more weight it is given to override conflicting declarations. In our example, the selector that includes the ID name (#intro) is more specific than a general element selector (like p), so that rule would apply to the “intro” paragraph, overriding the rules set for all paragraphs.

It’s a little soon to be discussing specificity because we’ve looked at only two types of selectors. For now, put the term specificity and the concept that some selectors have more “weight,” and therefore override others, on your radar.

Assigning Importance

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

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

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

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

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

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

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

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

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

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

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

Beware that the **!important** indicator is not a get-out-of-jail-free card. Best practices dictate that it should be used sparingly, if at all, and certainly never just to get yourself out of a sticky situation with inheritance and the cascade.

Rule order

After all the style sheet sources have been sorted by priority, and after all the linked and imported style sheets have been shuffled into place, there are likely to be conflicts in rules with equal weights. When that is the case, the order in which the rules appear is important. The cascade follows a "last one wins" rule. Whichever rule appears last has the last word.

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

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

In this scenario, paragraph text will be green because the last rule in the style sheet—that is, the one closest to the content in the document—overrides the earlier ones. Procedurally, the paragraph is assigned a color, then assigned a new one, and finally a third one (green) that gets used. The same thing happens when conflicting styles occur within a single declaration stack:

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

The resulting color will be green because the last declaration overrides the previous two. It is easy to accidentally override previous declarations within a rule when you get into compound properties, so this is an important behavior to keep in mind. That is a very simple example. What happens when style sheet rules from different sources come into play?

Let's consider an HTML document that has an embedded style sheet (added with the style element) that starts with an @import rule for importing an external .css file. That same HTML document also has a few inline style attributes applied to particular h1 elements.

STYLE DOCUMENT (*external.css*):

```
…
h1 { color: red }
…
```

HTML DOCUMENT:

```
<!DOCTYPE html>
<html>
<head>
  <title>…</title>
  <style>
    @import url(external.css);  /* set to red first */
    h1 { color: purple;}  /* overridden by purple */
  </style>
</head>
<body>
  <h1 style="color: blue">Heading</h1>  /* blue comes last and wins */
  …
</body>
</html>
```

When the browser parses the file, it gets to the imported style sheet first, which sets h1s to red. Then it finds a rule with equal weight in the embedded style sheet that overrides the imported rule, so h1s are set to purple. As it continues, it encounters a style rule right in an h1 that sets its color to blue.

Because that rule came last, it's the winner, and that h1 will be blue. That's the effect we witnessed in EXERCISE 1-3. Note that other h1s in this document without inline style rules would be purple, because that was the last h1 color applied to the whole document.

The Box Model

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

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

h1 { border: 1px solid blue; }

h2 { border: 1px solid blue; }

p { border: 1px solid blue; }

em { border: 1px solid blue; }

img { border: 1px solid blue; }

FIGURE 1-9 shows the results. The borders reveal the shape of each block element box. There are boxes around the inline elements (em and img) as well.

If you look at the headings, you will see that block element boxes expand to fill the available width of the browser window, which is the nature of block elements in the normal document flow. Inline boxes encompass just the characters or image they contain.

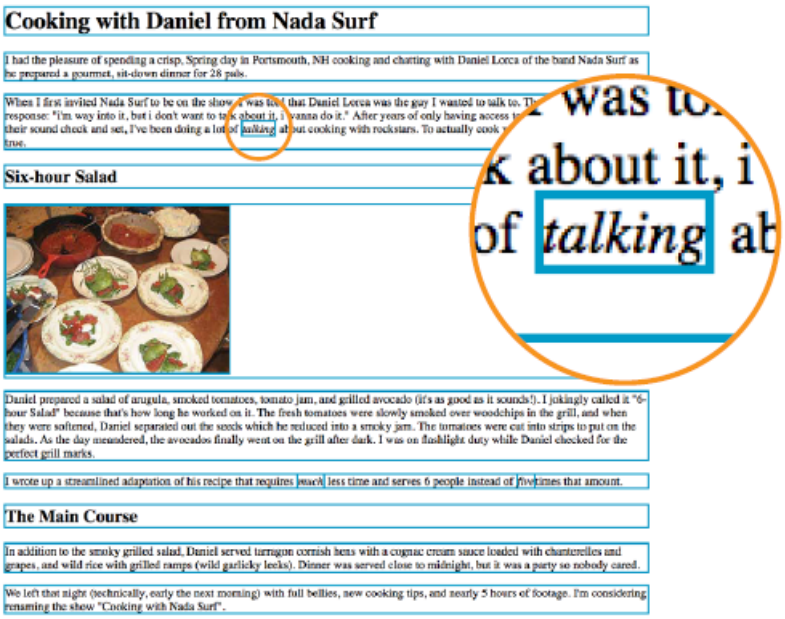


FIGURE 1-9. Rules around all the elements reveal their element boxes.

Grouped Selectors

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

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

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

CSS UNITS OF MEASUREMENT

This chapter lays the groundwork for upcoming lessons, so it's a good time to get familiar with the units of measurement used in CSS. You'll be using them to set font size, the width and height of elements, margins, indents, and so on.

The complete list is provided in the sidebar "CSS Units." Some will look familiar (like inches and millimeters), but there are some units that bear more explanation: absolute units, rem, em, and vw/vh. Knowing how to use CSS units effectively is another one of those core CSS skills.

CSS Units

CSS3 provides a variety of units of measurement. They fall into two broad categories: absolute and relative.

Absolute units

Absolute units have predefined meanings or real-world equivalents. With the exception of pixels, they are not appropriate for web pages that appear on screens.

- **px** pixel, defined as equal to 1/96 of an inch in CSS3.
- **in** inches.
- **mm** millimeters.
- **cm** centimeters.
- **q** ¼ millimeter.
- **pt** points (1/72 inch). Points are a unit commonly used in print design.
- **pc** picas (1 pica = 12 points or 1/6 inch). Points are a unit commonly used in print design.

Relative units

Relative units are based on the size of something else, such as the default text size or the size of the parent element.

- **em** a unit of measurement equal to the current font size.
- **ex** x-height, approximately the height of a lowercase "x" in the font.
- **rem** root em, equal to the em size of the root element (**html**).
- **ch** zero width, equal to the width of a zero (0) in the current font and size.
- **vw** viewport width unit, equal to 1/100 of the current viewport (browser window) width.
- **vh** viewport height unit, equal to 1/100 of the current viewport height.
- **vmin** viewport minimum unit, equal to the value of **vw** or **vh**, whichever is smaller.
- **vmax** viewport maximum unit, equal to the value of **vw** or **vh**, whichever is larger.

NOTES

- Although not a "unit," percentages are another common measurement value for web page elements. Percentages are calculated relative to another value, such as the value of a property applied to the current element or its parent or ancestor. The spec always says what a percentage value for a property is calculated on.

  When used for page layouts, percentage values ensure that page elements stay proportional.
- Child elements do not inherit the relative values of their parent, but rather the resulting *calculated* value.
- IE9 supports **vm** instead of **vmin**. IE and Edge (all versions as of 2017) do not support **vmax**.

Absolute Units

Absolute units have predefined meanings or real-world equivalents. They are always the same size, regardless of the context in which they appear.

The most popular absolute unit for web design is the pixel, which CSS3 defines as 1/96 inch. Pixels are right at home on a pixel-based screen and offer precise control over the size of the text and elements on the page. For a while there, pixels were all we used. Then we realized they are too rigid for pages that need to adapt to a wide variety of screen sizes and user preferences.

Relative measurements like rem, em, and % are more appropriate to the fluid nature of the medium.

As long as we are kicking px to the curb, all of the absolute units—such as pt, pc, in, mm, and cm—are out because they are irrelevant on screens, although they may be useful for print style sheets. That narrows down your unit choices a bit.

That said, pixels do still have their place in web design for elements that truly should stay the same size regardless of context. Border widths are appropriate in pixels, as are images that have inherent pixel dimensions.

Relative Units

As I just established, relative units are the way to go for most web measurements, and there are a few options: rem, em, and vw/vh.

The rem unit

CSS3 introduced a relative measurement called a rem (for root em) that is based on the font size of the root (html) element, whatever that happens to be. In modern browsers, the default root font size is 16 pixels; therefore, a rem is equivalent to a 16-pixel unit (unless you set it explicitly to another value).

An element sized to 10rem would measure 160 pixels.

For the most part, you can use rem units like an absolute measurement in style rules; however, because it is relative, if the base font size changes, so does the size of a rem. If a user changes the base font size to 24 pixels for easier reading from a distance, or if the page is displayed on a device that has a default font size of 24 pixels, that 10rem element becomes 240 pixels. That seems dodgy, but rest assured that it is a feature, not a bug. There are many instances in which you want a layout element to expand should the text size increase. It keeps the page proportional with the font size, which can help maintain optimum line lengths.

The em unit

An em is a relative unit of measurement that, in traditional typography, is based on the width of the capital letter M (thus the name “em”). In the CSS specification, an em is calculated as the distance between baselines when the font is set without any extra space between the lines (also known as leading).

For text with a font size of 16 pixels, an em measures 16 pixels; for 12-pixel text, an em equals 12 pixels; and so on, as shown in FIGURE 1-10.

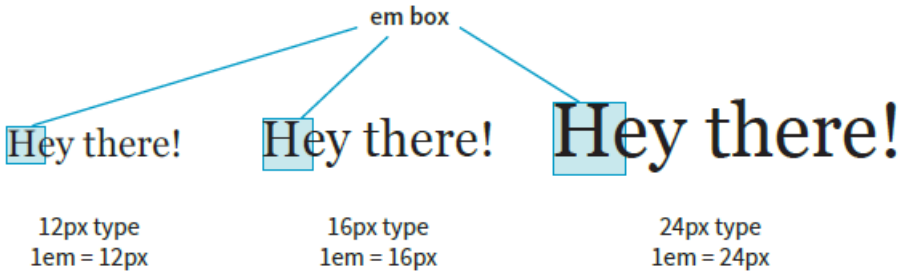


FIGURE 11-10. An em is based on the size of the text.

Once the dimension of an em for a text element is calculated by the browser, it can be used for all sorts of other measurements, such as indents, margins, the width of the element on the page, and so on. Basing measurements on text size helps keep everything in proportion should the text be resized.

The trick to working with ems is to remember they are always relevant to the current font size of the element. To borrow an example from Eric Meyer and Estelle Weyl's CSS: The Definitive Guide (O'Reilly), if you set a 2em left margin on an h1, h2, and p, those elements will not line up nicely because the em units are based on their respective element's sizes (FIGURE 1-11).

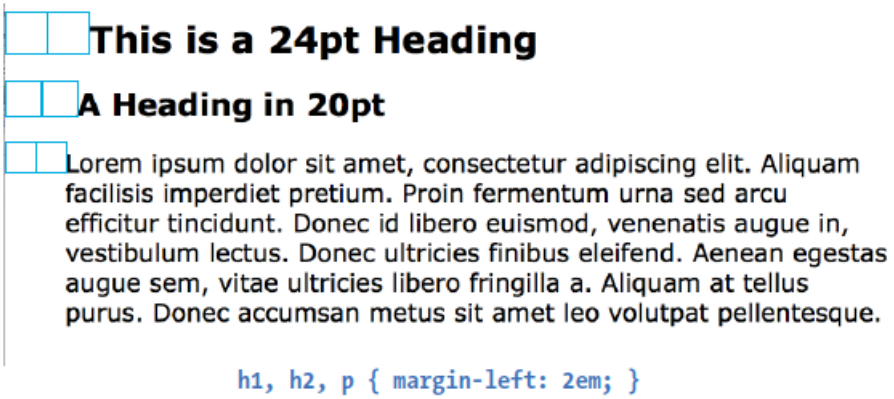


FIGURE 1-11. Em measurements are always relevant to the element's font size. An em for one element may not be the same for another.

Viewport percentage lengths (vw/vh)

The viewport width (vw) and viewport height (vh) units are relative to the size of the viewport (browser window). A vw is equal to 1/100 the width of the viewport. Similarly, a vh is equal to 1/100 the height of the viewport.

Viewport-based units are useful for making images and text elements stay the full width or height of the viewport:

header {

width: 100vw;

height: 100vh; }

It's also easy to specify a unit to be a specific percentage of the window size,

such as 50%:

img {

width: 50vw;

height: 50vh; }

Related are the vmin unit (equal to the value of vw or vh, whichever is smaller) and vmax (equal to the value of vw or vh, whichever is larger).

That should give you a good introduction to the units you'll be using in your style sheets.

DEVELOPER TOOLS RIGHT IN YOUR BROWSER

Because of the cascade, a single page element may have styles applied from a number of sources. This can make it tricky to debug a page when styles aren't displaying the way you think they should. Fortunately, every major browser comes with developer tools that can help you sort things out.

I've opened the simple cooking.html document that we've been working on in the Chrome browser, then selected View → Developer → Developer Tools from the menu. The Developer Tools panel opens at the bottom of the document, as you can see in FIGURE 1-12. You can also make it its own separate window by clicking the windows icon in the top left.

In the Elements tab on the left, I can see the HTML source for the document.

The content is initially hidden so you can see the structure of the document more clearly, but clicking the arrows opens each section. When I click the element in the source (like the second p element shown in the figure), that element is also highlighted in the browser window view.

In the Styles tab on the right, I can see all of the styles that are being applied to the selected element. In the example, I see the font-size, font-family, and margin-left properties from the style element in the document. If there were external CSS documents, they'd be listed too. I can also see the "User Agent Style Sheet," which is the browser's default styles. In this case, the browser style sheet adds the margin space around the paragraph. Chrome also provides a box model diagram for the selected element that shows the content dimensions, padding, border, and margins that are applied. This is a great tool for troubleshooting unexpected spacing in layouts.

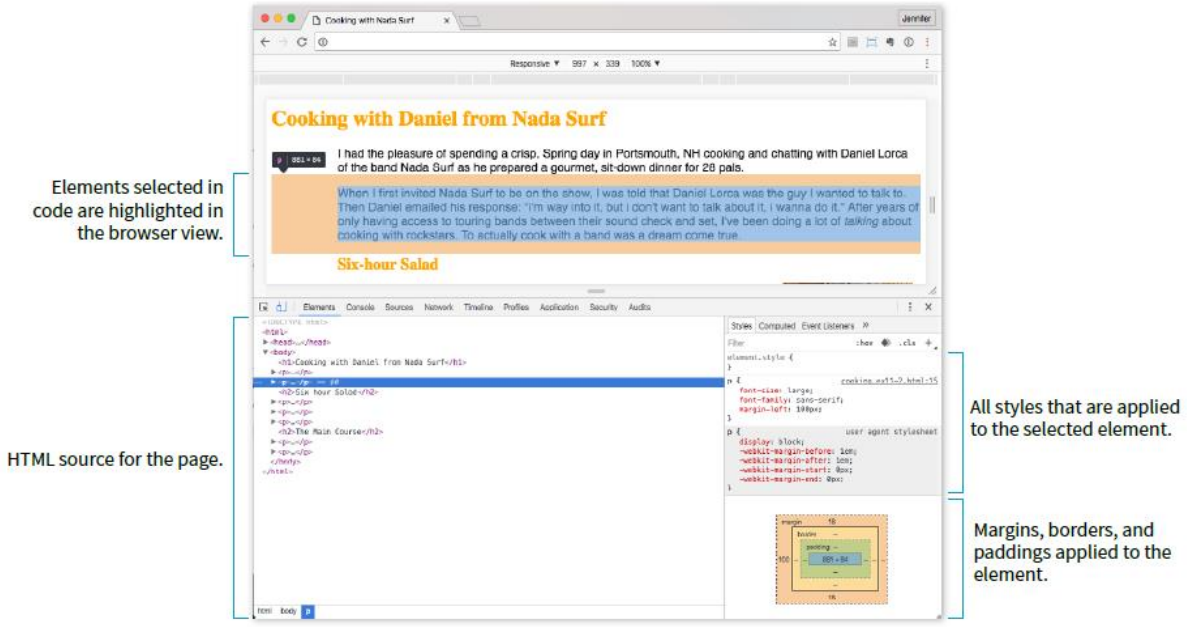


FIGURE 1-12. The Chrome browser with the Developer Tools panel open.

The cool thing is that when you edit the style rules in the panel, the changes are reflected in the browser view of the page in real time! If I select the h1 element and change the color from orange to green, it turns green in the window.

It's a great way to experiment with or troubleshoot a design; however, the changes are not being made to the document itself. It's just a preview, so you'll have to duplicate the changes in your source.

You can inspect any page on the web in this way, play around with turning styles off and on, and even add some of your own. Nothing you do has any effect on the actual site, so it is just for your education and amusement.

The element and style inspectors are just the tip of the iceberg of what browser developer tools can do. You can also tweak and debug JavaScript, check performance, view the document in various device simulations, and much more. The good news is that all major browsers now have built-in tools with similar features. As a web developer, you'll find they are your best friend.

- Chrome DevTools (View → Developer → Developer Tools)

  developer.chrome.com/devtools

- Firefox (Tools → Web Developer)

  developer.mozilla.org/en-US/docs/Tools

- Microsoft Edge (open with F12 key)

  developer.microsoft.com/en-us/microsoft-edge/platform/documentation/f12-devtools-guide/

- Safari (Develop → Show Web Inspector)

  developer.apple.com/library/content/documentation/AppleApplications/Conceptual/Safari_Developer_Guide/Introduction/Introduction.html)

- Opera (View → Developer Tools → Opera Dragonfly)

  www.opera.com/dragonfly/

- Internet Explorer 9+ (open with F12 key)

  msdn.microsoft.com/en-us/library/gg589512(v=vs.85).aspx

MOVING FORWARD WITH CSS

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

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

2.1 Formatting text

BASIC FONT PROPERTIES

When I design a text document (for print or the web), one of the first things I do is specify a font. In CSS, fonts are specified using a set of font-related properties for typeface, size, weight, font style, and special characters. There are also shortcut properties that let you specify multiple font attributes in a single rule.

A Word About Property Listings

Each CSS property listing in this book is accompanied by information on how it behaves and how to use it. Property listings include:

Values:
These are the accepted values for the property. Predefined keyword values appear in code font (for example, **`small`**, **`italic`**, or **`small-caps`**) and must be typed in exactly as shown.

Default:
This is the value that will be used for the property by default (its initial value)—that is, if no other value is specified. Note that the default browser style sheet values may vary from the defaults defined in CSS.

Applies to:
Some properties apply only to certain types of elements.

Inherits:
This indicates whether the property is passed down to the element's descendants.

CSS-wide keywords

All CSS properties accept the three CSS-wide keywords: **`initial`**, **`inherit`**, and **`unset`**. Because they are shared by all properties, they are not listed with the values for individual property listings.

- The **`initial`** keyword explicitly sets the property to its default (initial) value.
- The **`inherit`** keyword allows you to explicitly force an element to inherit a style property from its parent. This may come in handy to override other styles applied to that element and to guarantee that the element always matches its parent.
- Finally, **`unset`** erases declared values occurring earlier in the cascade, setting the property to either **`inherit`** or **`initial`**, depending on whether it inherits or not.

Specifying the Font Name

Choosing a typeface, or font family as it is called in CSS, for your text is a good place to start. Let's begin with the font-family property and its values.

font-family

Values: one or more font or generic font family names, separated by commas

Default: depends on the browser

Applies to: all elements

Inherits: yes

Use the font-family property to specify a font or list of fonts (known as a font stack) by name, as shown in these examples:

```
body { font-family: Arial; }

var { font-family: Courier, monospace; }

p { font-family: "Duru Sans", Verdana, sans-serif; }
```

Here are some important syntax requirements:

- All font names, with the exception of generic font families, must be capitalized. For example, use Arial instead of arial.
- Use commas to separate multiple font names, as shown in the second and third examples.
- Notice that font names that contain a character space (such as Duru Sans in the third example) must appear within quotation marks.

You might be asking, "Why specify more than one font?" That's a good question, and it brings us to one of the challenges of specifying fonts for the web.

Font limitations

Browsers are limited to displaying fonts they have access to. Traditionally, that meant the fonts that were already installed on the user's hard drive. In 2010, however, there was a boom in browser support for embedded web fonts using the CSS @font-face rule, so it became possible for designers to provide their own fonts. See the sidebar "Say Hello to Web Fonts" for more information.

But back to our font-family rule. Even when you specify that the font should be Futura in a style rule, if the browser can't find it (for example, if that font is not installed on the user's computer or the provided web font fails to load), the browser uses its default font instead.

Fortunately, CSS allows us to provide a list of back-up fonts (that font stack we saw earlier) should our first choice not be available. If the first specified font is not found, the browser tries the next one, and down through the list until it finds one that works. In the third font-family rule shown in the previous code example, if the browser does

not find Duru Sans, it will use Verdana, and if Verdana is not available, it will substitute some other sans-serif font.

AT A GLANCE

Font Properties

The CSS2.1 font-related properties are universally supported:

```
font-family
font-size
font-weight
font-style
font-variant
font
```

The CSS Font Module Level 3 adds these properties for more sophisticated font handling, although browser support is inconsistent as of this writing:

```
font-stretch
font-variant-ligatures
font-variant-position
font-variant-caps
font-variant-numeric
font-variant-alternates
font-variant-east-asian
font-size-adjust
font-kerning
font-feature-settings
font-language-override
```

Generic font families

That last option, “some other sans-serif font,” bears more discussion. “Sansserif” is just one of five generic font families that you can specify with the font-family property. When you specify a generic font family, the browser chooses an available font from that stylistic category. FIGURE 2-2 shows examples from each family.

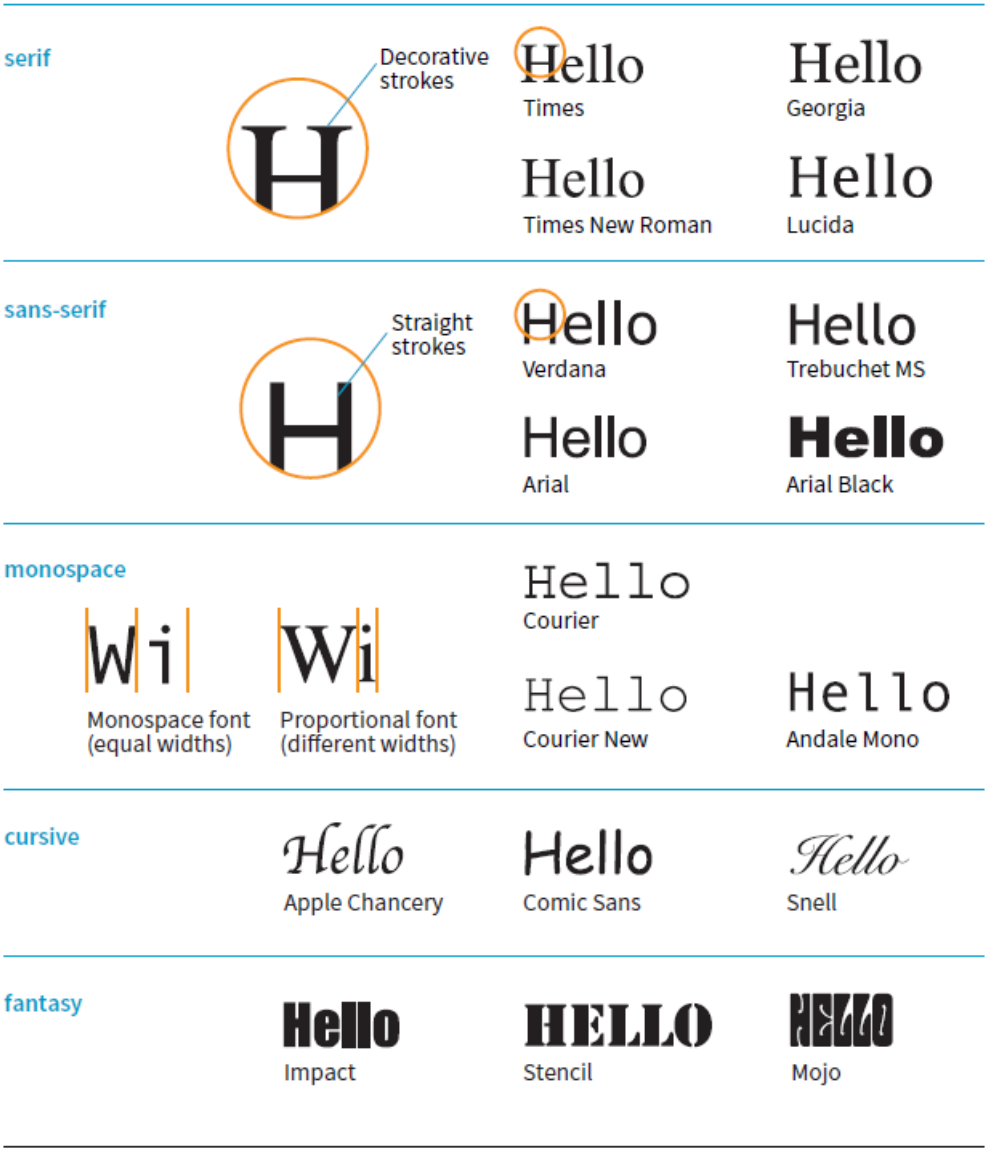


FIGURE 2-2. Examples of the five generic font families.

serif

Examples: Times, Times New Roman, Georgia Serif typefaces have decorative slab-like appendages (serifs) on the ends of certain letter strokes.

sans-serif

Examples: Arial, Arial Black, Verdana, Trebuchet MS, Helvetica, Geneva Sans-serif typefaces have straight letter strokes that do not end in serifs.

monospace

Examples: Courier, Courier New, and Andale Mono

In monospace (also called constant width) typefaces, all characters take up the same amount of space on a line. For example, a capital W will be no wider than a lowercase i. Compare this to proportional typefaces (such as the one you're reading now) that allot different widths to different characters.

cursive

Examples: Apple Chancery, Zapf-Chancery, and Comic Sans Cursive fonts emulate a script or handwritten appearance.

fantasy

Examples: Impact, Western, or other decorative font

Fantasy fonts are purely decorative and would be appropriate for headlines and other display type.

Font stack strategies

The best practice for specifying fonts for web pages is to start with your first choice, provide some similar alternatives, and then end with a generic font family that at least gets users in the right stylistic ballpark. For example, if you want an upright, sans-serif font, you might start with a web font if you are providing one (Oswald), list a few that are more common (Univers, Tahoma, Geneva), and finish with the generic sans-serif. There is no limit to the number of fonts you can include, but many designers strive to keep it under 10.

font-family: Oswald, Univers, Tahoma, Geneva, sans-serif;

A good font stack should include stylistically related fonts that are known to be installed on most computers. Sticking with fonts that come with the Windows, macOS, and Linux operating systems, as well as fonts that get installed with popular software packages such as Microsoft Office and Adobe Creative Suite, gives you a solid list of "web-safe" fonts to choose from. A good place to look for stylistically related web-safe fonts is CSS Font Stack (www.cssfontstack.com). There are many articles on font stack strategies that are just a Google search away. I recommend Michael Tuck's "8 Definitive Font Stacks" (www.sitepoint.com/eight-definitive-font-stacks), which is an oldie but goodie.

So, as you see, specifying fonts for the web is more like merely suggesting them. You don't have absolute control over which font your users will see.

You might get your first choice; you might get the generic fallback. It's one of those web design quirks you learn to live with.

Now seems like a good time to get started formatting the Black Goose Bistro menu. We'll add new style rules one at a time as we learn new properties, staring with EXERCISE 2-1.

EXERCISE 2-1. Formatting a menu

In this exercise, we'll change the fonts for the body and main heading of the Black Goose Bistro menu document, menu.html, which is available at.learningwebdesign.com/5e/materials. Open the document in a text editor.

Hang on to this document, because this exercise will continue as we pick up additional font properties.

I've included an embedded font in this exercise to show you how easy it is to do with a service like Google Web Fonts.

1. Use an embedded style sheet for this exercise. Start by adding a

   style element in the head of the document, like this:

   ```
   <head>
   <title>Black Goose Bistro</title>
   <style>
   </style>
   </head>
   ```

2. I would like the main text to appear in Verdana or some other sans-serif font. Instead of writing a rule for every element in the document, we will write one rule for the body element that will be inherited by all the elements it contains. Add this rule to the embedded style sheet:

   ```
   <style>
       body {font-family: Verdana, sans-serif;}
   </style>
   ```

3. I want a fancy font for the "Black Goose Bistro, Summer Menu" headline, so I chose a free display font called Marko One from Google Web Fonts (www.google.com/webfonts). Google gave me the code for linking the font file on their server to my HTML file (it's actually a link to an external style sheet). It must be placed in the head of the document, so copy it exactly as it appears, but keep it on one line. Put it after the title and before the style element.

   ```
   <head>
   <title>Black Goose Bistro</title>
   <link href="http://fonts.googleapis.com/
   css?family=Marko+One" rel="stylesheet">
   <style>
   …
   ```

4. Now write a rule that applies it to the h1 element. Notice I've specified Georgia or another serif font as fallbacks:

   <style>

   body {font-family: Verdana, sans-serif;}

   h1 {font-family: "Marko One", Georgia, serif;}

   </style>

5. Save the document and reload the page in the browser. It should look like FIGURE 12-3. Note that you'll need to have an internet connection and a current browser to view the Marko One headline font. We'll work on the text size in the next exercise.

Black Goose Bistro • Summer Menu

Baker's Corner, Seekonk, Massachusetts
Hours: Monday through Thursday: 11 to 9, Friday and Saturday; 11 to midnight

Appetizers

This season, we explore the spicy flavors of the southwest in our appetizer collection.

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

Big, bold flavors are the name of the game this summer. Allow us to assist you with finding the perfect wine.

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. **Very spicy.** $12.95
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
Grilled skirt steak with mushroom fricasee
Flavorful skirt steak marinated in asian flavors grilled as you like it*. Served over a blend of sauteed wild mushrooms with a side of blue cheese mashed potatoes. $16.95

* We are required to warn you that undercooked food is a health risk.

FIGURE 2-3. The menu after we change only the font family.

Specifying Font Size

Use the aptly named font-size property to specify the size of the text.

font-size

Values: length unit | percentage | xx-small | x-small | small | medium | large |

x-large | xx-large | smaller | larger

Default: medium

Applies to: all elements

Inherits: yes

You can specify text size in several ways:

- Using one of the CSS length units, as shown here:

    h1 { font-size: 1.5em; }

When specifying a number of units, be sure the unit abbreviation immediately follows the number, with no extra character space in between (see the sidebar “Providing Measurement Values”).

CSS length units are discussed in Chapter 11, Introducing Cascading Style Sheets. See also the “CSS Units Cheat Sheet” sidebar.

- As a percentage value, sized up or down from the element’s inherited font size:

    h1 { font-size: 150%; }

- Using one of the absolute keywords (xx-small, x-small, small, medium, large, x-large, xx-large). On most current browsers, medium corresponds to the default font size.

    h1 { font-size: x-large; }

- Using a relative keyword (larger or smaller) to nudge the text larger or smaller than the surrounding text:

    strong { font-size: larger; }

I’m going to cut to the chase and tell you that, despite all these options, the preferred values for font-size in contemporary web design are the relative length units em and rem, as well as percentage values. You can specify font size in pixels (px), but in general, they do not provide the flexibility required in web page design. All of the other absolute units (pt, pc, in, etc.) are out too, unless you are creating a style sheet specifically for print.

I’ll explain the keyword-based font-size values in a moment, but let’s start our discussion with the best practice using relative values.

Providing Measurement Values

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

```
margin: 2em;
```

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

```
INCORRECT: margin: 2 em;
```

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

```
margin: 0;
```

AT A GLANCE

CSS Units Cheat Sheet

As a quick reference, here are the CSS length units again:

Relative units

em	ex	rem	ch
vw	vh	vmin	vmax

Absolute units

px	in	mm	cm
q	pt	pc	

Sizing text with relative values

The best practice for setting the font size of web page elements is to do it in a way that respects the user's preference. Relative sizing values %, rem, and em allow you to use the default font size as the basis for proportional sizing of other text elements. It's usually not important that the headlines are exactly 24 pixels; it is important that they are 1.5 times larger than the main text so they stand out. If the user changes their preferences to make their default font size larger, the headlines appear larger, too.

To maintain the browser's default size, set the font-size of the root element to 100% (see Note):

```
html {
    font-size: 100%;
}
```

That sets the basis for relative sizing. Because the default font size for all modern browsers is 16 pixels, we'll assume our base size is 16 pixels going forward (we'll also keep in mind that it could be different).

Rem values

The rem unit, which stands for "root em," is always relative to the size of the root (html) element. If the root size is 16 pixels, then a rem equals 16 pixels.

What's nice about rem units is, because they are always relative to the same element, they are the same size wherever you use them throughout the document.

In that way, they work like an absolute unit. However, should the root size be something other than 16 pixels, elements specified in rem values will resize accordingly and proportionally. It's the best of both worlds.

Here is that same heading sized with rem values:

```
h1 { font-size: 1.5rem; } /* 1.5 x 16 = 24 */
```

Em measurements

Em units are based on the font size of the current element. When you specify font-size in ems, it will be relative to the inherited size for that element.

Once the em is calculated for an element, it can be used for other measurements as well, such as margins, padding, element widths, and any other setting you want to always be relative to the size of the font.

Here I've used em units to specify the size of an h1 that has inherited the default 16-pixel font size from the root:

```
h1 { font-size: 1.5em; } /* 1.5 x 16 = 24 */
```

There are a few snags to working with ems. One is that because of rounding errors, there is some inconsistency in how browsers and platforms render text set in ems.

The other tricky aspect to using ems is that they are based on the inherited size of the element, which means that their size is based on the context in which they are applied.

The h1 in the previous example was based on an inherited size of 16 pixels.

But if this h1 had appeared in an article element that had its font size set to 14 pixels, it would inherit the 14-pixel size, and its resulting size would be just 21 pixels (1.5 × 14 = 21). FIGURE 2-4 shows the results.

THE MARKUP

```
<h1>Headline in Body</h1>
<p>Pellentesque ligula leo,…</p>
<article>
    <h1>Headline in Article</h1>
    <p>Vivamus …</p>
</article>
```

THE STYLES

```
h1 {
    font-size: 1.5em; /* sets all h1s to 1.5em */
}
    article {
    font-size: .875em /* 14 pixels based on 16px default */
}
```

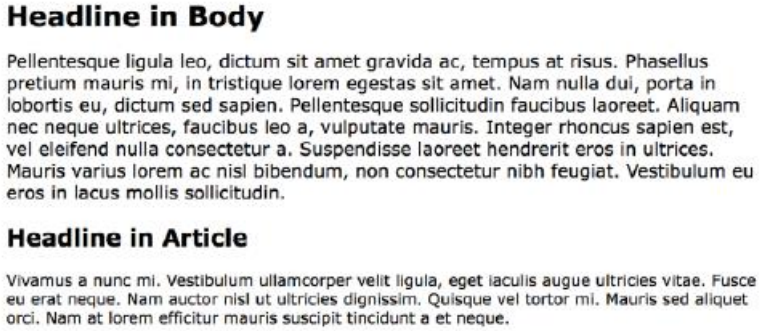


FIGURE 2-4. All h1 elements are sized at 1.5em, but they are different sizes because of the context in which they appear.

From this example, you can see that an element set in ems might appear at different sizes in different parts of the document. If you wanted the h1 in the article to be 24 pixels as well, you could calculate the em value by dividing the target size by its context: 24 / 14 = 1.71428571 em. (No need to round that figure down…the browser knows what to do with it.)

If you have elements nested several layers deep, the size increase or decrease compounds, which can create problems. With many layers of nesting, text may end up being way too small. When working with ems, pay close attention and write style rules in a way that takes the context into account.

This compounding nature of the em is what has driven the popularity of the predictable rem unit.

Percentage values

We saw a percentage value (100%) used to preserve the default font size, but you can use percentage values for any element. They are pretty straightforward.

In this example, the h1 inherits the default 16px size from the html element, and applying the 150% value multiplies that inherited value, resulting in an h1 that is 24 pixels:

```
h1 { font-size: 150%; } /* 150% of 16 = 24 */
```

Working with keywords

An alternative way to specify font-size is by using one of the predefined absolute keywords: xx-small, x-small, small, medium, large, x-large, and xx-large. The keywords do not correspond to particular measurements, but rather are scaled consistently in relation to one another. The default size is medium in current browsers. FIGURE 2-5 shows how each of the absolute keywords renders in a browser when the default text is set at 16 pixels. I've included samples in Verdana and Times to show that, even with the same base size, there is a big difference in legibility at sizes small and below.

Verdana was designed to be legible on screens at small font sizes; Times was designed for print so is less legible in that context.

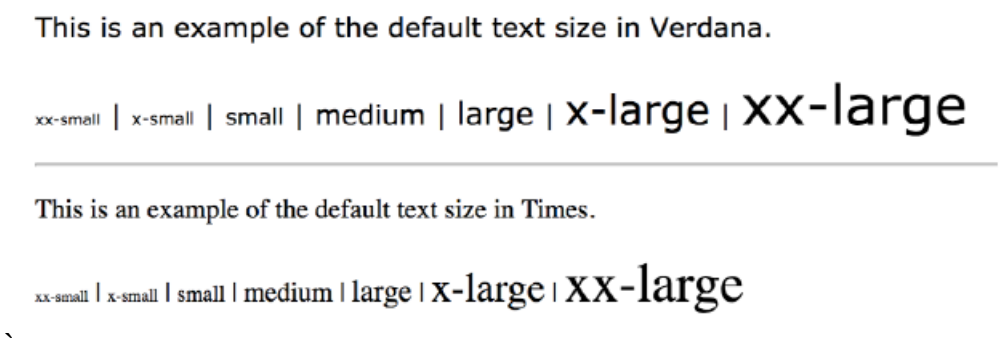


FIGURE 2-5. Text sized with absolute keywords.

The relative keywords, larger and smaller, are used to shift the size of text relative to the size of the parent element text. The exact amount of the size change is determined by each browser and is out of your control. Despite that limitation, it is an easy way to nudge type a bit larger or smaller if the exact proportions are not critical.

You can apply your new CSS font knowledge in EXERCISE 2-2.

EXERCISE 2-2. Setting font size

Let's refine the size of some of the text elements to give the online menu a more sophisticated appearance. Open menu.html in a text editor and follow the steps. You can save the document at any point and take a peek in the browser to see the results of your work. You should also feel free to try out other size values along the way.

1. There are many approaches to sizing text on web pages. In this example, start by putting a stake in the ground and setting the font-size of the body element to 100%, thus clearing the way for em measurements thereafter:

   body {

   font-family: Verdana, sans-serif;

   font-size: 100%;

   }

2. The browser default of 16 pixels is a fine size for the main page text, but I would like to improve the appearance of the heading levels. I'd like the main heading to be 24 pixels, or one and a half times larger than the body text [target (24) ÷ context (16) = 1.5]. I'll add a new rule that sets the size of the h1 to 1.5em. I could have used 150% to achieve the same thing.

   h1 {

   font-size: 1.5em;

   }

3. Now make the h2s the same size as the body text so they blend in with the page better:

   h2 {

   font-size: 1em;

   }

FIGURE 2-6 shows the result of our font-sizing efforts.

FIGURE 2-6. The online menu after a few minor font-size changes to the headings.

Font Weight (Boldness)

After font families and size, the remaining font properties are straightforward.

For example, if you want a text element to appear in bold, use the font-weight property to adjust the boldness of type.

> *font-weight*
>
> Values: normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900
>
> Default: normal
>
> Applies to: all elements
>
> Inherits: yes

As you can see, the font-weight property has many predefined values, including descriptive terms (normal, bold, bolder, and lighter) and nine numeric values (100 to 900) for targeting various weights of a font if they are available.

Because most fonts commonly used on the web have only two weights, normal (or Roman) and bold, the only font weight value you will use in most cases is bold. You may also use normal to make text that would otherwise appear in bold (such as strong text or headlines) appear at a normal weight.

The numeric chart may come in handy when using web fonts with a large range of weights (I've seen a few Google web fonts that require numeric size values). If multiple weights are not available, numeric settings of 600 and higher generally result in bold text, as shown in FIGURE 2-7 (although even that can vary by browser).

If a separate bold face is not available, the browser may "synthesize" a bold font by beefing up the available normal face (see Note).

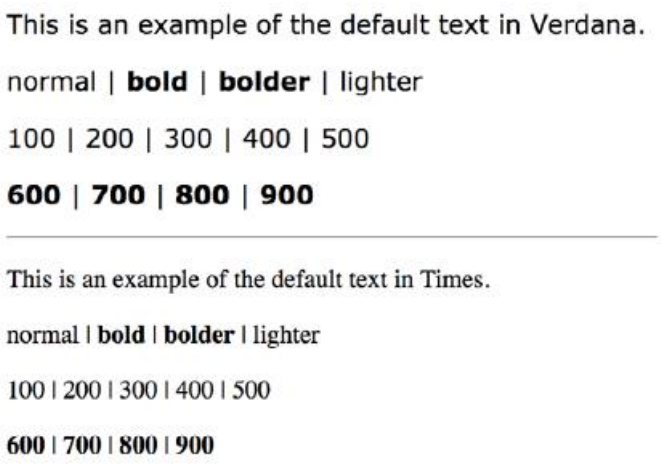


FIGURE 2-7. The effect (and lack thereof!) of font-weight values.

Font Style (Italics)

The font-style property affects the posture of the text—that is, whether the letter shapes are vertical (normal) or slanted (italic and oblique).

font-style

Values: normal | italic | oblique

Default: normal

Applies to: all elements

Inherits: yes

Use the font-style property to make text italic. Another common use is to make text that is italicized in the browser's default styles (such as emphasized text) display as normal. There is an oblique value that specifies a slanted version of the font; however, browsers generally display oblique exactly the same as italic.

Try out weight and style in EXERCISE 2-3.

EXERCISE 2-3. Making text bold and italic

Back to the menu. I've decided that I'd like all of the menu item names to be in bold text. What I'm not going to do is wrap each one in <b> tags…that would be so 1996! I'm also not going to mark them up as strong elements…that is not semantically accurate. Instead, the right thing to do is simply apply a style to the semantically correct dt (definition term) elements to make them all bold at once. Add this rule to the end of the style sheet, save the file, and try it out in the browser:

dt { font-weight: bold; }

Now that all the menu item names are bold, some of the text I've marked as strong isn't standing out very well, so I think I'll make them italic for further emphasis. To do this, simply apply the fontstyle property to the strong element:

strong { font-style: italic;}

Once again, save and reload. It should look like the detail shown in FIGURE 2-8.

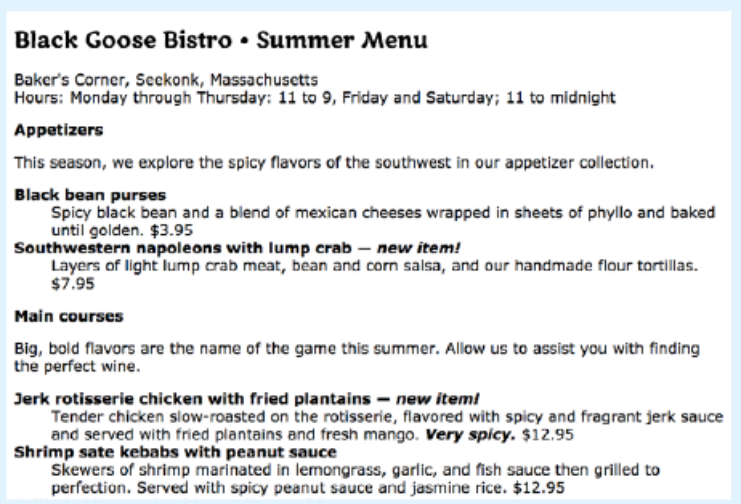


FIGURE 2-8. Applying the font-weight and font-style properties.

Font Variant in CSS2.1 (Small Caps)

font-variant

Values: normal | small-caps

Default: normal

Applies to: all elements

Inherits: yes

Some typefaces come in a “small caps” variant. This is a separate font design that uses small uppercase-style letters in place of lowercase letters. Small caps characters are designed to match the size and density of lowercase text so they blend in.

Small caps should be used for strings of three or more capital letters appearing in the flow of text, such as acronyms and abbreviations, that may look jarring as full-sized capitals. Compare NASA and USA in the standard font to nasa and usa in small caps. Small caps are also recommended for times, like 1am or 2017ad.

When the font-variant property was introduced in CSS2.1, it was a one-trick pony that allowed designers to specify a small-caps font for text elements.

CSS3 has greatly expanded the role of font-variant, as I will cover in the upcoming section “Advanced Typography with CSS3.” For now, we’ll look at only the CSS2.1 version of font-variant.

In most cases, browsers simulate small caps by scaling down uppercase letters in the current font. To typography sticklers, this is less than ideal and results in inconsistent stroke weights, but you may find it an acceptable option for adding variety to small amounts of text. You will see an example of small caps when we use the font-variant property in EXERCISE 2-5.

Font Stretch (Condensed and Extended)

font-stretch

Values: normal | ultra-condensed | extra-condensed | condensed |

semi-condensed | semi-expanded | expanded | extra-expanded |

ultra-expanded

Default: normal

Applies to: all elements

Inherits: yes

The CSS3 font-stretch property tells the browser to select a normal, condensed, or extended font in the font family (FIGURE 2-9). If the browser cannot find a matching font, it will not try to synthesize the width by stretching or squeezing text; it may just substitute a font of a different width. Browser support is just beginning to kick in for this property. As of this writing, it works on IE11+, Edge, Firefox, Chrome 48+, Opera, and Android 52+, but it is not yet supported on Safari or iOS Safari; however, that may change.

The Shortcut font Property

Specifying multiple font properties for each text element can get repetitive and lengthy, so the creators of CSS provided the shorthand font property, which compiles all the font-related properties into one rule.

font

Values: font-style font-weight font-variant font-stretch font-size/line-height font-family | caption | icon | menu | message-box | small-caption | status-bar

Default: depends on default value for each property listed

Applies to: all elements

Inherits: yes

The value of the font property is a list of values for all the font properties we just looked at, separated by character spaces. It is important to note that only the CSS2.1 version of font-variant (small-caps) can be used in the font shortcut (which is one reason I kept it separate). In this property, the order of the values is important:

{ font: style weight stretch variant size/line-height font-family; }

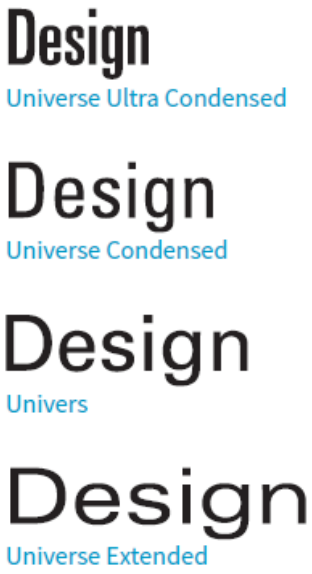


FIGURE 2-9. Examples of condensed, normal, and extended versions of the Universe typeface.

At minimum, the font property must include a font-size value and a fontfamily value, in that order. Omitting one or putting them in the wrong order causes the entire rule to be invalid. This is an example of a minimal font property value:

p { font: 1em sans-serif; }

Once you've met the size and family requirements, the other values are optional and may appear in any order prior to the font-size. When style, weight, stretch, or variant is omitted, its value is set to normal. That makes it easy to accidentally override a previous setting with the shorthand property, so be careful when you use it.

There is one value in there, line-height, that we have not seen yet. As it sounds, it adjusts the height of the text line and is used to add space between lines of text. It appears just after font-size, separated by a slash, as shown in these examples. The line-height property is covered in more detail later in this chapter.

h3 { font: oblique bold small-caps 1.5em/1.8em Verdana, sans-serif; }

h2 { font: bold 1.75em/2 sans-serif; }

In EXERCISE 2-4, we'll use the shorthand font property to make some changes to the h1 headings in the bistro menu.

System font keywords

The font property also has a number of keyword values (caption, icon, menu, message-box, small-caption, and status-bar) that represent system fonts, the fonts used by operating systems for things like labels for icons and menu items. These may be useful when you're designing a web application so that it matches the environment the user is working on. These are considered shorthand values because they encapsulate the font, size, style, and weight of the font used for each purpose with only one keyword.

Like the shorthand font property, EXERCISE 2-4 is short and sweet.

CHANGING TEXT COLOR

You change the color of text with the color property.

color

Values: color value (name or numeric)

Default: depends on the browser and user's preferences

Applies to: all elements

Inherits: yes

Using the color property is very straightforward. The value of the color property can be a predefined color name (see the "Color Names" sidebar) or a numeric value describing a specific RGB color. Here are a few examples, all of which make the h1 elements in a document gray:

```
h1 { color: gray; }

h1 { color: #666666; }

h1 { color: #666; }

h1 { color: rgb(102,102,102); }
```

Don't worry about the numeric values for now; I just wanted you to see what they look like.

Color is inherited, so you can change the color of all the text in a document by applying the color property to the body element, as shown here:

```
body { color: fuchsia; }
```

OK, so you probably wouldn't want all your text to be fuchsia, but you get the idea.

For the sake of accuracy, I want to point out that the color property is not strictly a text-related property. In fact, according to the CSS specification, it is used to change the foreground (as opposed to the background) color of an element. The foreground of an element consists of both the text it contains as well as its border. So, when you apply a color to an element (including image elements), know that color will be used for the border as well, unless there is a specific border-color property that overrides it. Before we add color to the online menu, I want to take a little side trip and introduce you to a few more types of selectors that will give us more flexibility in targeting elements in the document for styling.

AT A GLANCE

Color Names

CSS2.1 defines 17 standard color names:

black	white	purple
lime	navy	aqua
silver	maroon	fuchsia
olive	blue	orange
gray	red	green
yellow	teal	

The updated CSS Color Module Level 3 allows names from a larger set of 140 color names to be specified in style sheets. You can see samples of each in FIGURE 13-2 and at *learningwebdesign.com/colornames.html*.

EXERCISE 2-5. Using selectors

This time, we'll add a few more style rules using descendant, ID, and class selectors combined with the font and color properties we've learned about so far.

1. I'd like to add some attention-getting color to the "new item!" elements next to certain menu item names. They are marked up as strong, so we can apply the color property to the strong element. Add this rule to the embedded style sheet, save the file, and reload it in the browser:

```
strong {
font-style: italic;
color: tomato;
}
```

That worked, but now the strong element "Very spicy" in the description is "tomato" red too, and that's not what I want.

The solution is to use a contextual selector that targets only the strong elements that appear in dt elements. Remove the color declaration you just wrote from the strong rule, and create a new rule that targets only the strong elements within definition list terms:

```
dt strong { color: tomato; }
```

2. Look at the document source, and you will see that the content has been divided into three unique divs: info, appetizers, and entrees. We can use these to our advantage when it comes to styling. For now, let's do something simple and apply a teal color to the text in the div with the ID "info". Because color inherits,

we need to apply the property only to the div and it will be passed down to the h1 and p:

```
#info { color: teal; }
```

3. Now let's get a little fancier and make the paragraph inside the "info" section italic in a way that doesn't affect the other paragraphs on the page. Again, a contextual selector is the answer. This rule selects only paragraphs contained within the info section of the document:

```
#info p { font-style: italic; }
```

4. I want to give special treatment to all of the prices on the menu. Fortunately, they have all been marked up with span elements:

```
<span class="price">$3.95</span>
```

So now all we have to do is write a rule using a class selector to change the font to Georgia or some serif font, make the prices italic, and gray them back:

```
.price {
    font-family: Georgia, serif;
    font-style: italic;
    color: gray;
}
```

5. Similarly, in the "info" div, I can change the appearance of the spans that have been marked up as belonging to the "label" class to make the labels stand out:

```
.label {
    font-weight: bold;
    font-variant: small-caps;
    font-style: normal;
}
```

6. Finally, there is a warning at the bottom of the page that I want to make small and red. It has been given the class "warning," so I can use that as a selector to target just that paragraph for styling. While I'm at it, I'm going to apply the same style to the sup element (the footnote asterisk) earlier on the page so they match. Note that I've used a grouped selector, so I don't need to write a *separate rule.*

```
p.warning, sup {
font-size: small;
color: red;
}
```

FIGURE 2-11 shows the results of all these changes. We now have some touches of color and special typography treatments.

FIGURE 2-11. The current state of the bistro menu.

EXERCISE 2-6. Finishing touches

Let's add a few finishing touches to the online menu, menu.html.

It might be useful to save the file and look at it in the browser after each step to see the effect of your edits and to make sure you're on track. The finished style sheet is provided in the materials folder for this chapter.

1. First, I have a few global changes to the body element in mind. I've had a change of heart about the font-family. I think that a serif font such as Georgia would be more sophisticated and appropriate for a bistro menu. Let's also use the line-height property to open up the text lines and make them easier to read. Make these updates to the body style rule, as shown:

```
body {
    font-family: Georgia, serif;
    font-size: small;
    line-height: 1.75em;
}
```

2. I also want to redesign the "info" section of the document. Remove the teal color setting by deleting that whole rule. Once that is done, make the h1 olive green and the paragraph in the header gray. Add color declarations to the existing rules:

```
#info { color: teal; } /* delete */
h1 {
```

font: bold 1.5em "Marko One", Georgia, serif;

color: olive;}

#info p {

font-style: italic;

color: gray;}

3. Next, to imitate a fancy restaurant menu, I'm going to center a few key elements on the page with the text-align property. Write a rule with a grouped selector to center the headings and the "info" section:

   h1, h2, #info {

   text-align: center;}

4. I want to make the "Appetizer" and "Main Courses" h2 headings more eye-catching. Instead of large, bold type, I'm going to use all uppercase letters, extra letter spacing, and color to call attention to the headings. Here's the new rule for h2 elements that includes all of these changes:

   h2 {

   font-size: 1em;

   text-transform: uppercase;

   letter-spacing: .5em;

   color: olive;}

5. We're really close now; just a few more tweaks to those paragraphs right after the h2 headings. Let's center those too and make them italic:

   h2 + p {

   text-align: center;

   font-style: italic;}

   Note that I've used a next-sibling selector (h2 + p) to select any paragraph that follows an h2.

6. Next, add a softer color to the menu item names (in dt elements). I've chosen "sienna," one of the names from the CSS3 color module. Note that the strong elements in those dt elements stay "tomato" red because the color applied to the strong elements overrides the color inherited by their parents.

   dt {

   font-weight: bold;

   color: sienna;}

7. Finally, for kicks, add a drop shadow under the h1 heading. You can play around with the values a little to see how it works. I find it to look a little clunky against a white background, but when you have a patterned background image, sometimes a drop shadow provides the little punch you need to make the text stand out. Notice how small the shadow values are—a little goes a long way!

```
h1 {
    font: bold 1.5em "Marko One", Georgia, serif;
    color: olive;
    text-shadow: .05em .05em .1em lightslategray;}
```

And we're done! FIGURE 2-20 shows how the menu looks now—an improvement over the unstyled version, and we used only text and color properties to do it. Notice that we didn't touch a single character of the document markup in the process. That's the beauty of keeping style separate from structure.

Black Goose Bistro • Summer Menu

Baker's Corner, Seekonk, Massachusetts

HOURS: MONDAY THROUGH THURSDAYS *11 to 9*, FRIDAY AND SATURDAYS *11 to midnight*

APPETIZERS

This season, we explore the spicy flavors of the southwest in our appetizer collection.

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

Big, bold flavors are the name of the game this summer. Allow us to assist you with finding the perfect wine.

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. ***Very spicy.*** *$12.95*

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*

Grilled skirt steak with mushroom fricasee

Flavorful skirt steak marinated in asian flavors grilled as you like it*. Served over a blend of sauteed wild mushrooms with a side of blue cheese mashed potatoes. *$16.95*

* We are required to warn you that undercooked food is a health risk.

FIGURE 2-20. The formatted Black Goose Bistro menu.

CHANGING LIST BULLETS AND NUMBERS

Before we close out this chapter on text properties, I want to show you a few tweaks you can make to bulleted and numbered lists. As you know, browsers automatically insert bullets before unordered list items, and numbers before items in ordered lists (the list markers). For the most part, the rendering of these markers is determined by the browser. However, CSS provides a few properties that allow authors to choose the type and position of the marker, or turn them off entirely.

Choosing a Marker

Apply the list-style-type property to the ul, ol, or li element select the type of marker that appears before each list item (see Note).

list-style-type

Values: none | disc | circle | square | decimal | decimal-leading-zero | lower-alpha | upper-alpha | lower-latin | upper-latin | lower-roman | upper-roman | lower-greek

Default: disc

Applies to: ul, ol, and li (or elements whose display value is list-item)

Inherits: yes

More often than not, developers use the list-style-type property with its value set to none to remove bullets or numbers altogether. This is handy when you're using list markup as the foundation for a horizontal navigation menu or the entries in a web form. You can keep the semantics but get rid of the pesky markers.

The disc, circle, and square values generate bullet shapes just as browsers have been doing since the beginning of the web itself (FIGURE 2-21).

Unfortunately, there is no way to change the appearance (size, color, etc.) of generated bullets, so you're stuck with the browser's default rendering.

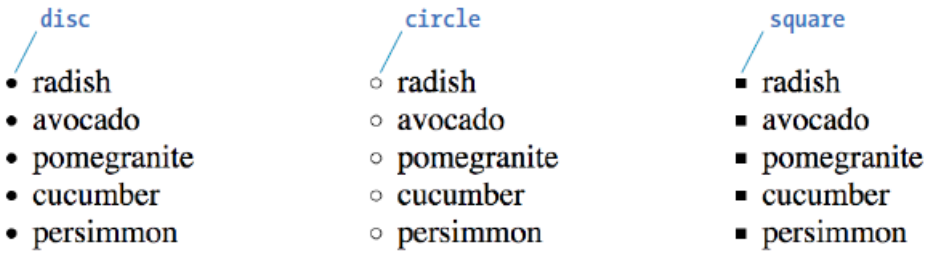


FIGURE 2-21. The list-style-type values disc, circle, and square.

The remaining keywords (TABLE 2-1) specify various numbering and lettering styles for use with ordered lists.

Keyword	System
decimal	1, 2, 3, 4, 5...
decimal-leading-zero	01, 02, 03, 04, 05...
lower-alpha	a, b, c, d, e...
upper-alpha	A, B, C, D, E...
lower-latin	a, b, c, d, e... (same as lower-alpha)
upper-latin	A, B, C, D, E... (same as upper-alpha)
lower-roman	i, ii, iii, iv, v...
upper-roman	I, II, III, IV, V...
lower-greek	α, β, γ, δ, ε...

TABLE 2-1. Lettering and numbering system (CSS2.1)

Marker Position

By default, the marker hangs outside the content area for the list item, displaying as a hanging indent. The list-style-position property allows you to pull the bullet inside the content area so it runs into the list content.

list-style-position

Values: inside | outside | hanging

Default: outside

Applies to: ul, ol, and li (or elements whose display value is list-item)

Inherits: yes

I've applied a light green background color to the list items in FIGURE 2-22 to reveal the boundaries of their content area boxes.

You can see that when the position is set to outside (top), the markers fall outside the content area.When it is set to inside (bottom), the markers are tucked into the content area.

li {background-color: #F99;}

ul#outside {list-style-position: outside;}

ul#inside {list-style-position: inside;}

CSS3 adds the hanging value for list-styleposition.

it is similar to inside, but the markers appear outside and abutting the left edge of the shaded area.

outside

- **Radish.** Praesent in lacinia risus. Morbi urna ipsum, efficitur id erat pellentesque, tincidunt commodo sem. Phasellus est velit, porttitor vel dignissim vitae, commodo ut urna.
- **Avocado.** Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Curabitur lacinia accumsan est, ut malesuada lorem consectetur eu.
- **Pomegranite.** Nam euismod a ligula ac bibendum. Aenean ac justo eget lorem dapibus aliquet. Vestibulum vitae luctus orci, id tincidunt nunc. In a mauris odio. Duis convallis enim nunc.

inside

- **Radish.** Praesent in lacinia risus. Morbi urna ipsum, efficitur id erat pellentesque, tincidunt commodo sem. Phasellus est velit, porttitor vel dignissim vitae, commodo ut urna.
- **Avocado.** Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Curabitur lacinia accumsan est, ut malesuada lorem consectetur eu.
- **Pomegranite.** Nam euismod a ligula ac bibendum. Aenean ac justo eget lorem dapibus aliquet. Vestibulum vitae luctus orci, id tincidunt nunc. In a mauris odio. Duis convallis enim nunc.

FIGURE 2-22. The list-style-position property.

3. JavaScript

JavaScript (JS) is the most popular lightweight, interpreted compiled programming language. It can be used for both Client-side as well as Server-side developments. JavaScript also known as a scripting language for web pages.

Reason to Learn JavaScript

JavaScript is used by many developers (65% of the total development community), and the number is increasing day by day. JavaScript is one such programming language that has more than 1444231 libraries and increasing rapidly. It is preferred over any other programming language by most developers. Also, major tech companies like Microsoft, Uber, Google, Netflix, and Meta use JavaScript in their projects.

JavaScript can be added to your HTML file in two ways:

1. Internal JavaScript
2. External JavaScript

Internal JavaScript

We can add JS code directly to our HTML file by writing the code inside the <script> & </script>. The <script> tag can either be placed inside the <head> or the <body> tag according to the requirement.

Example: It is the basic example of using JavaScript code inside of HTML code, that script enclosing section can be placed in the body or head of the HTML document.

HTML

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>
        Basic Example to Describe JavaScript
    </title>
</head>

<body>

    <!-- JavaScript Code -->
    <script>
        console.log("Welcome to GeeksforGeeks");
    </script>
</body>

</html>
```

Run on IDE

External JavaScript: We can create the file with a .js extension and paste the JS code inside of it. After creating the file, add this file in <script src="file_name.js"> tag, and this <sctipt> can import inside <head> or <body> tag of the HTML file.

Example: It is the basic example of using JavaScript javascript code which is written in a different file. By importing that .js file in the head section.

HTML Javascript

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>
        Basic Example to Describe JavaScript
    </title>
    <script src="main.js"></script>
</head>

<body>
</body>

</html>
```

3.1 Using JavaScript (and the Document Object Mdel)

MEET THE DOM

> The DOM gives us a way to access and manipulate the contents of a document.

You've seen references to the Document Object Model (DOM for short) several times throughout this book, but now is the time to give it the attention it deserves. The DOM gives us a way to access and manipulate the contents of a document. We commonly use it for HTML, but the DOM can be used with any XML language as well. And although we're focusing on its relationship with JavaScript, it's worth noting that the DOM can be accessed by other languages too, such as PHP, Ruby, C++, and more. Although DOM Level 1 was released by the W3C in 1998, it was nearly five years later that DOM scripting began to gain steam.

The DOM is a programming interface (an API) for HTML and XML pages.

It provides a structured map of the document, as well as a set of methods to interface with the elements contained therein. Effectively, it translates our markup into a format that JavaScript (and other languages) can understand.

It sounds pretty dry, I know, but the basic gist is that the DOM serves as a map to all the elements on a page and lets us do things with them. We can use it to find elements by their names or attributes, and then add, modify, or delete elements and their content.

Without the DOM, JavaScript wouldn't have any sense of a document's contents—and by that, I mean the entirety of the document's contents.

Everything from the page's doctype to each individual letter in the text can be accessed via the DOM and manipulated with JavaScript.

The Node Tree

A simple way to think of the DOM is in terms of the document tree as diagrammed in FIGURE 3-1. You saw documents diagrammed in this way when you were learning about CSS selectors.

```
<!DOCTYPE html>
<html>
<head>
    <title>Document title</title>
    <meta charset="utf-8">
</head>
<body>
    <div>
    <h1>Heading</h1>
    <p>Paragraph text with a <a href="foo.html">link</a> here.</p>
    </div>
    <div>
    <p>More text here.</p>
    </div>
</body>
</html>
```

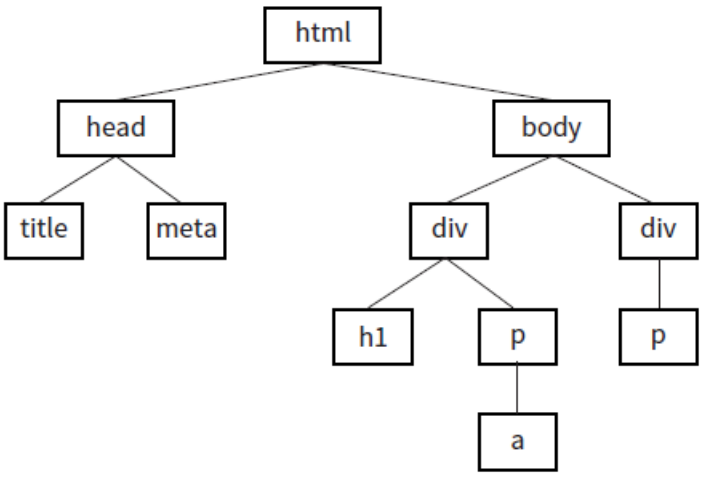


FIGURE 1-1. A simple document.

Each element within the page is referred to as a node. If you think of the DOM as a tree, each node is an individual branch that can contain further branches. But the

DOM allows deeper access to the content than CSS because it treats the actual content as a node as well. FIGURE 1-2 shows the structure of the first p element. The element, its attributes, and its contents are all nodes in the DOM's node tree.

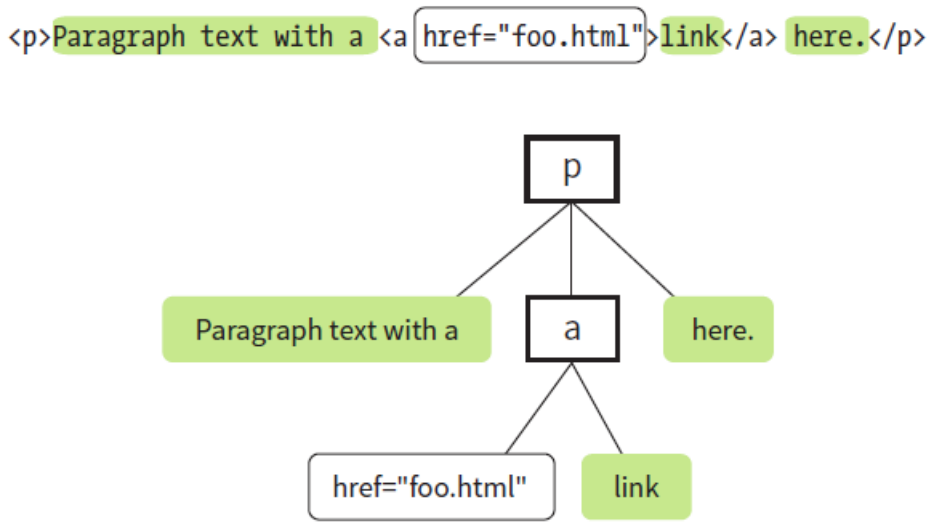


FIGURE 1-2. The nodes within the first p element in our sample document.

The DOM also provides a standardized set of methods and functions through which JavaScript can interact with the elements on our page. Most DOM scripting involves reading from and writing to the document.

There are several ways to use the DOM to find what you want in a document.

Let's go over some of the specific methods we can use for accessing objects defined by the DOM (we JS folks call this "crawling the DOM" or "traversing the DOM"), as well as some of the methods for manipulating those elements.

Accessing DOM Nodes

The document object in the DOM identifies the page itself, and more often than not will serve as the starting point for our DOM crawling. The document object comes with a number of standard properties and methods for accessing collections of elements. Just as length is a standard property of all arrays, the document object comes with a number of built-in properties containing information about the document. We then wind our way to the element we're after by chaining those properties and methods together, separated by periods, to form a sort of route through the document.

To give you a general idea of what I mean, the statement in this example says to look on the page (document), find the element that has the id value "beginner", find the HTML content within that element (innerHTML), and save those contents to a variable (foo):

var foo = document.getElementById("beginner").innerHTML;

Because the chains tend to get long, it is also common to see each property or method broken onto its own line to make it easier to read at a glance.

Remember, whitespace in JavaScript is ignored, so this has no effect on how the statement is parsed.

```
var foo = document
    .getElementById("beginner")
        .innerHTML;
```

There are several methods for accessing nodes in the document.

By element name

getElementsByTagName()

We can access individual elements by the tags themselves, using document.

getElementsByTagName(). This method retrieves any element or elements you specify as an argument.

For example, document.getElementsByTagName("p") returns every paragraph on the page, wrapped in something called a collection or nodeList, in the order they appear in the document from top to bottom. nodeLists behave much like arrays. To access specific paragraphs in the nodeList, we reference them by their index, just like an array.

```
var paragraphs = document.getElementsByTagName("p");
```

Based on this variable statement, paragraphs[0] is a reference to the first paragraph in the document, paragraphs[1] refers to the second, and so on. If we had to access each element in the nodeList separately, one at a time…well, it's a good thing we learned about looping through arrays earlier. Loops work the exact same way with a nodeList.

```
var paragraphs = document.getElementsByTagName("p");
for( var i = 0; i < paragraphs.length; i++ ) {
    // do something
}
```

Now we can access each paragraph on the page individually by referencing paragraphs[i] inside the loop, just as with an array, but with elements on the page instead of values.

By id attribute value

getElementById()

This method returns a single element based on that element's ID (the value of its id attribute), which we provide to the method as an argument. For example, to access this particular image

<img src="photo.jpg" alt="" id="lead-photo">

we include the id value as an argument for the getElementById() method:

var photo = document.getElementById("lead-photo");

By class attribute value

getElementsByClassName()

Just as it says on the tin, this allows you to access nodes in the document based on the value of a class attribute. This statement assigns any element with a class value of "column-a" to the variable firstColumn so it can be accessed easily from within a script:

var firstColumn = document.getElementsByClassName("column-a");

Like getElementsByTagName(), this returns a nodeList that we can reference by index or loop through one at a time.

By selector

querySelectorAll()

querySelectorAll() allows you to access nodes of the DOM based on a CSSstyle selector. The syntax of the arguments in the following examples should look familiar to you. It can be as simple as accessing the child elements of a specific element:

var sidebarPara = document.querySelectorAll(".sidebar p");

or as complex as selecting an element based on an attribute:

var textInput = document.querySelectorAll("input[type='text']");

querySelectorAll() returns a nodeList, like getElementsByTagName() and getElementsByClassName(), even if the selector matches only a single element.

Accessing an attribute value

getAttribute()

As I mentioned earlier, elements aren't the only thing you can access with the DOM. To get the value of an attribute attached to an element node, we call getAttribute() with a single argument: the attribute name. Let's assume we have an image, stratocaster.jpg, marked up like this:

<img src="stratocaster.jpg" alt="electric guitar" id="lead-image">

In the following example, we access that specific image (getElementbyId()) and save a reference to it in a variable ("bigImage"). At that point, we could access any of the element's attributes (alt, src, or id) by specifying it as an argument in the getAttribute() method. In the example, we get the value of the src attribute and use it

as the content in an alert message. (I'm not sure why we would ever do that, but it does demonstrate the method.)

var bigImage = document.getElementById("lead-image");

alert(bigImage.getAttribute("src")); // Alerts "stratocaster.jpg".

Manipulating Nodes

Once we've accessed a node by using one of the methods discussed previously, the DOM gives us several built-in methods for manipulating those elements, their attributes, and their contents.

setAttribute()

To continue with the previous example, we saw how we get the attribute value, but what if we wanted to set the value of that src attribute to a new pathname altogether? Use setAttribute()! This method requires two arguments: the attribute to be changed and the new value for that attribute.

In this example, we use a bit of JavaScript to swap out the image by changing the value of the src attribute:

var bigImage = document.getElementById("lead-image");

bigImage.setAttribute("src", "lespaul.jpg");

Just think of all the things you could do with a document by changing the values of attributes. Here we swapped out an image, but we could use this same method to make a number of changes throughout our document:

- Update the checked attributes of checkboxes and radio buttons based on user interaction elsewhere on the page.
- Find the link element for our .css file and point the href value to a different style sheet, changing all the page's styles.
- Update a title attribute with information on an element's state ("this element is currently selected," for example).

innerHTML

innerHTML gives us a simple method for accessing and changing the text and markup inside an element. It behaves differently from the methods we've covered so far. Let's say we need a quick way of adding a paragraph of text to the first element on our page with a class of intro:

var introDiv = document.getElementsByClassName("intro");

introDiv[0].innerHTML = " This is our intro text ";

The second statement here adds the content of the string to introDiv (an element with the class value "intro") as a real live element because innerHTML tells JavaScript to parse the strings " " and " " as markup.

style

The DOM also allows you to add, modify, or remove a CSS style from an element by using the style property. It works similarly to applying a style with the inline style attribute. The individual CSS properties are available as properties of the style property. I bet you can figure out what these statements are doing by using your new CSS and DOM know-how:

```
document.getElementById("intro").style.color = "#fff";

document.getElementById("intro").style.backgroundColor = "#f58220";

    //orange
```

In JavaScript and the DOM, property names that are hyphenated in CSS (such as background-color and border-top-width) become camel case (backgroundColor and borderTopWidth, respectively) so the "–" character isn't mistaken for an operator.

In the examples you've just seen, the style property is used to set the styles for the node. It can also be used to get a style value for use elsewhere in the script. This statement gets the background color of the #intro element and assigns it to the brandColor variable:

```
var brandColor = document.getElementById("intro").style.backgroundColor;
```

Adding and Removing Elements

So far, we've seen examples of getting and setting nodes in the existing document.

The DOM also allows developers to change the document structure itself by adding and removing nodes on the fly. We'll start out by creating new nodes, which is fairly straightforward, and then we'll see how we add the nodes we've created to the page. The methods shown here are more surgical and precise than adding content with innerHTML. While we're at it, we'll remove nodes, too.

createElement()

To create a new element, use the aptly named createElement() method. This function accepts a single argument: the element to be created. Using this method is a little counterintuitive at first because the new element doesn't appear on the page right away. Once we create an element in this way, that new element remains floating in the JavaScript ether until we add it to the document. Think of it as creating a reference to a new element that lives purely in memory—something that we can manipulate in JavaScript as we see fit, and then add to the page once we're ready:

```
var newDiv = document.createElement("div");
```

createTextNode()

If we want to enter text into either an element we've created or an existing element on the page, we can call the createTextNode() method. To use it, provide a string of text as an argument, and the method creates a DOM-friendly version of that text, ready for inclusion on the page. Like createElement(), this creates a reference to the new text node that we can store in a variable and add to the page when the time comes:

var ourText = document.createTextNode("This is our text.");

appendChild()

So we've created a new element and a new string of text, but how do we make them part of the document? Enter the appendChild() method. This method takes a single argument: the node you want to add to the DOM. You call it on the existing element that will be its parent in the document structure. Time for an example.

Here we have a simple div on the page with the id "our-div":

* *

Let's say we want to add a paragraph to #our-div that contains the text "Hello, world!" We start by creating the p element (document.createElement()) as well as a text node for the content that will go inside it (createTextNode()):

var ourDiv = document.getElementById("our-div");

var newParagraph = document.createElement("p");

var copy = document.createTextNode("Hello, world!");

Now we have our element and some text, and we can use appendChild() to put the pieces together:

newParagraph.appendChild(copy);

ourDiv.appendChild(newParagraph);

The first statement appends copy (that's our "Hello, world!" text node) to the new paragraph we created (newParagraph), so now that element has some content. The second line appends the newParagraph to the original div (ourDiv). Now ourDiv isn't sitting there all empty in the DOM, and it will display on the page with the content "Hello, world!"

You should be getting the idea of how it works. How about a couple more?

insertBefore()

The insertBefore() method, as you might guess, inserts an element before another element. It takes two arguments: the first is the node that gets inserted, and the second is the element it gets inserted in front of. You also need to know the parent to which the element will be added.

So, for example, to insert a new heading before the paragraph in this markup

* *

* Our paragraph text *

* *

we start by assigning variable names to the div and the p it contains, and then create the h1 element and its text node and put them together, just as we saw in the last example:

```
var ourDiv = document.getElementById("our-div");

var para = document.getElementById("our-paragraph");

var newHeading = document.createElement("h1");

var headingText = document.createTextNode("A new heading");

newHeading.appendChild( headingText );

// Add our new text node to the new heading
```

Finally, in the last statement shown here, the insertBefore() method places the newHeading h1 element before the para element inside ourDiv.

```
ourDiv.insertBefore( newHeading, para );
```

replaceChild()

The replaceChild() method replaces one node with another and takes two arguments. The first argument is the new child (i.e., the node you want to end up with). The second is the node that gets replaced by the first. As with insertBefore(), you also need to identify the parent element in which the swap happens. For the sake of simplicity, let's say we start with the following markup:

```
<div id="our-div">

<div id="swap-me"></div>

</div>
```

And we want to replace the div with the id "swap-me" with an image. We start by creating a new img element and setting the src attribute to the pathname to the image file. In the final statement, we use replaceChild() to put newImg in place of swapMe.

```
var ourDiv = document.getElementById("our-div");

var swapMe = document.getElementById("swap-me");

var newImg = document.createElement("img");

// Create a new image element

newImg.setAttribute( "src", "path/to/image.jpg" );

// Give the new image a "src" attribute

ourDiv.replaceChild( newImg, swapMe );
```

removeChild()

To paraphrase my mother, “We brought these elements into this world, and we can take them out again.” You remove a node or an entire branch from the document tree with the removeChild() method. The method takes one argument, which is the node you want to remove. Remember that the DOM thinks in terms of nodes, not just elements, so the child of an element may be the text (node) it contains, not just other elements.

Like appendChild(), the removeChild() method is always called on the parent element of the element to be removed (hence, “remove child”). That means we’ll need a reference to both the parent node and the node we’re looking to remove. Let’s assume the following markup pattern:

```
<div id="parent">
    <div id="remove-me">
        <p>Pssh, I never liked it here anyway.</p>
    </div>
</div>
```

Our script would look something like this:

```
var parentDiv = document.getElementById("parent");
var removeMe = document.getElementById("remove-me");
parentDiv.removeChild( removeMe );
// Removes the div with the id "remove-me" from the page.
```

3.2 Tic-Tac-Toe

A fun way to experience JavaScript is with the aid of a fun game “Tic-Tac-Toe”

The big question then… Where do we start?

Well, usually the best way to start would be to break down the application into smaller, easily digestible pieces.

First, let’s break down the user interface:

- Title
- 3x3 grid
    - the grid should be clickable
    - the grid cells should have the correct player sign displayed an information display
- should display a message informing the current player it’s their turn
    - should show us who won the game
    - should show us if the game ended in a draw
- restart button
    - will restart the entire game

Next, let’s break down the game flow for a cell click:

- needs to track any clicks that happen on our cells
- needs to check if a valid move has been made
    - needs to make sure nothing happens if an already played cell has been clicked
- we should update our game state
- we should validate the game state
    - check if a player has won
    - check if the game ended in a draw
- either stop the game or change the active player, depending on the above checks
- reflect the updates made on the UI
- rinse and repeat

That’s it, nothing special or overly complicated but still an excellent opportunity to practice and improve.

Let’s get to the fun part and build something!

Folder Structure

We’ll start by building the user interface so we have something to look at while building the game logic.

As I mentioned this is a simple game so there is no need for complicated folder structures.

You should have three files in total:

1. index.html (will hold our UI structure and import the other files we need)
2. style.css (to make our game look halfway decent)
3. script.js (will hold our game logic, and handle everything else we need)

HTML

```
<!doctype html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport"

        content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0">

    <meta http-equiv="X-UA-Compatible" content="ie=edge">

    <title>Tic Tac Toe</title>

    <link rel="stylesheet" href="style.css">
```

```
</head>
<body>
    <section>
        <h1 class="game--title">Tic Tac Toe</h1>
        <div class="game--container">
            <div data-cell-index="0" class="cell"></div>
            <div data-cell-index="1" class="cell"></div>
            <div data-cell-index="2" class="cell"></div>
            <div data-cell-index="3" class="cell"></div>
            <div data-cell-index="4" class="cell"></div>
            <div data-cell-index="5" class="cell"></div>
            <div data-cell-index="6" class="cell"></div>
            <div data-cell-index="7" class="cell"></div>
            <div data-cell-index="8" class="cell"></div>
        </div>
        <h2 class="game--status"></h2>
        <button class="game--restart">Restart Game</button>
    </section>
<script src="script.js"></script>
</body>
</html>
```

Aside from the usual boilerplate, we have included our style sheet in the <head> element, we do this to make sure the style sheet is always loaded before the actual HTML.

We have also included our script.js file just above the closing </body> tag to make sure that the JavaScript is always loaded after the HTML.

We will hold the actual game cells in a wrapping div to allow us to make use of the CSS grid. Also, each cell has a “data-cell-index” attribute to allow us to easily track with cell has been clicked.

We also have an <h2> element where we will display the before mentioned game information and a restart button.

CSS

```
body {
    font-family: "Arial", sans-serif;
}
section {
    text-align: center;
}
.game--container {
    display: grid;
    grid-template-columns: repeat(3, auto);
    width: 306px;
    margin: 50px auto;
}
.cell {
    font-family: "Permanent Marker", cursive;
    width: 100px;
    height: 100px;
    box-shadow: 0 0 0 1px #333333;
    border: 1px solid #333333;
    cursor: pointer;
line-height: 100px;
    font-size: 60px;
}
```

I wanted to keep the CSS for the application down to a minimum so the only thing I'd draw your attention to are the styles for the ".game — container" since this is where we implement our CSS grid.

Since we want to have a 3x3 grid we make use of the "grid-template-columns" property setting it to repeat(3, auto);

In a nutshell, this splits the contained divs (cells) into three columns and letting the cells automatically decide their width.

JavaScript

Now we get to the fun part!

Let’s kick our JS off by structuring some pseudo-code to break it down into smaller pieces using our before written game logic template

```
/*
We store our game status element here to allow us to more easily
use it later on
*/
const statusDisplay = document.querySelector('.game--status');
/*
Here we declare some variables that we will use to track the
game state throught the game.
*/
/*
We will use gameActive to pause the game in case of an end scenario
*/
let gameActive = true;
/*
We will store our current player here, so we know whos turn
*/
let currentPlayer = "X";
/*
We will store our current game state here, the form of empty strings in an array
 will allow us to easily track played cells and validate the game state later on
*/
let gameState = ["", "", "", "", "", "", "", "", ""];
/*
```

Here we have declared some messages we will display to the user during the game. Since we have some dynamic factors in those messages, namely the current player, we have declared them as functions, so that the actual message gets created with current data every time we need it.

```
*/
const winningMessage = () => `Player ${currentPlayer} has won!`;
const drawMessage = () => `Game ended in a draw!`;
const currentPlayerTurn = () => `It's ${currentPlayer}'s turn`;
/*
We set the inital message to let the players know whose turn it is
*/
statusDisplay.innerHTML = currentPlayerTurn();
function handleCellPlayed() {
}
function handlePlayerChange() {
}
function handleResultValidation() {
}
function handleCellClick() {
}
function handleRestartGame() {
}
/*
And finally we add our event listeners to the actual game cells, as well as our
restart button
*/
document.querySelectorAll('.cell').forEach(cell => cell.addEventListener('click', handleCellClick));
document.querySelector('.game--restart').addEventListener('click', handleRestartGame);
```

We have also outlined all the functionalities we will need to handle our game logic, so let's go and write our logic!

handleCellClick

In our cell click handler, we'll handle two things.

First off we need to check if the clicked cell has already been clicked and if it hasn't we need to continue our game flow from there.

Let’s see how this looks in action:

```
function handleCellClick(clickedCellEvent) {
/*
We will save the clicked html element in a variable for easier further use
*/
    const clickedCell = clickedCellEvent.target;
/*
Here we will grab the 'data-cell-index' attribute from the clicked cell to identify where that cell is in our grid.
Please note that the getAttribute will return a string value. Since we need an actual number we will parse it to an
integer(number)
*/
    const clickedCellIndex = parseInt(
      clickedCell.getAttribute('data-cell-index')
    );
/*
Next up we need to check whether the call has already been played,
or if the game is paused. If either of those is true we will simply ignore the click.
*/
    if (gameState[clickedCellIndex] !== "" || !gameActive) {
        return;
    }
/*
If everything if in order we will proceed with the game flow
*/
    handleCellPlayed(clickedCell, clickedCellIndex);
    handleResultValidation();
}
```

We will accept a ClickEvent from our cell event listener. That will allow us to track which cell has been clicked and get its’ index attribute more easily.

handleCellPlayed

In this handler, we'll need to handle two things. We'll update our internal game state, and update our UI.

```
function handleCellPlayed(clickedCell, clickedCellIndex) {
/*
We update our internal game state to reflect the played move,
as well as update the user interface to reflect the played move
*/
    gameState[clickedCellIndex] = currentPlayer;
    clickedCell.innerHTML = currentPlayer;
}
```

We accept the currently clicked cell (the .target of our click event), and the index of the cell that has been clicked.

handleResultValidation

Here comes the core of our Tic Tac Toe game, the result validation. Here we will check whether the game ended in a win, draw, or if there are still moves to be played.

Let's start by checking if the current player won the game.

```
const winningConditions = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8],
    [0, 3, 6],
    [1, 4, 7],
    [2, 5, 8],
    [0, 4, 8],
    [2, 4, 6]
];
function handleResultValidation() {
    let roundWon = false;
    for (let i = 0; i <= 7; i++) {
        const winCondition = winningConditions[i];
```

```
        let a = gameState[winCondition[0]];
        let b = gameState[winCondition[1]];
        let c = gameState[winCondition[2]];
        if (a === '' || b === '' || c === '') {
            continue;
        }
        if (a === b && b === c) {
            roundWon = true;
            break
        }
    }
if (roundWon) {
        statusDisplay.innerHTML = winningMessage();
        gameActive = false;
        return;
    }
}
```

Take a minute to break this down before continuing as an exercise.

Values in arrays for our winningConditions are indexes for cells that need to be populated by the same player for them to be considered a victor.

In our for-loop, we go through each one and check whether the elements of our game state array under those indexes match. If they do match we move on to declare the current player as victorious and ending the game.

Of course, we need to handle the other two cases as well. Let's first check if the game has ended in a draw. The only way the game can end in a draw would be if all the fields have been filled in.

```
const winningConditions = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8],
    [0, 3, 6],
    [1, 4, 7],
```

```
    [2, 5, 8],
    [0, 4, 8],
    [2, 4, 6]
];
function handleResultValidation() {
    let roundWon = false;
    for (let i = 0; i <= 7; i++) {
        const winCondition = winningConditions[i];
        let a = gameState[winCondition[0]];
        let b = gameState[winCondition[1]];
        let c = gameState[winCondition[2]];
        if (a === '' || b === '' || c === '') {
            continue;
        }
        if (a === b && b === c) {
            roundWon = true;
            break
        }
    }
if (roundWon) {
        statusDisplay.innerHTML = winningMessage();
        gameActive = false;
        return;
    }
/*
We will check weather there are any values in our game state array
that are still not populated with a player sign
*/
    let roundDraw = !gameState.includes("");
    if (roundDraw) {
```

```
        statusDisplay.innerHTML = drawMessage();
        gameActive = false;
        return;
    }
/*
If we get to here we know that the no one won the game yet,
and that there are still moves to be played, so we continue by changing the current player.
*/
    handlePlayerChange();
}
```

Since we have a return statement in our roundWon check we know that, if a player has won that round, our script will stop there. This allows us to avoid using else conditions and to keep our code nice and compact.

handlePlayerChange

Here we will simply change the current player and update the game status message to reflect the change.

```
function handlePlayerChange() {
    currentPlayer = currentPlayer === "X" ? "O" : "X";
    statusDisplay.innerHTML = currentPlayerTurn();
}
```

We are using a ternary operator here to assign a new player, you can learn more about it here. It really is awesome!

The only thing left to do would be to connect our game restart functionality.

handleRestartGame

Here we will set all our game tracking variables back to their defaults, clear the game board by removing all the signs, as well as updating the game status back to the current player message.

```
function handleRestartGame() {
    gameActive = true;
    currentPlayer = "X";
    gameState = ["", "", "", "", "", "", "", "", ""];
    statusDisplay.innerHTML = currentPlayerTurn();
```

```
    document.querySelectorAll('.cell')
            .forEach(cell => cell.innerHTML = "");
}
```

Conclusion

Basically, that’s it!

You have a functioning playable Tic-Tac-Toe game.