CHAPTER **2**

Coding Standards, Block Elements, Text Elements, and Character References

CHAPTER OBJECTIVES

- Understand HTML5 coding conventions and learn where they can be found.
- Use HTML comments appropriately.
- Use the content model categories to determine which elements are allowed inside a given container.
- Define a block element.
- Understand `blockquote` elements.
- Know when whitespace collapsing occurs and how to combat it with the `pre` element.
- Understand phrasing elements, like `q`, `cite`, `dfn`, `abbr`, and `time`.
- Learn CSS basics.
- Learn how and when to use character references.

CHAPTER OUTLINE

2.1 Introduction

In the prior chapter, you were introduced to just enough about HTML elements so you could put together a rudimentary web page. In this chapter, we'll introduce you to additional HTML elements, so your web pages can be more expressive. In presenting the HTML elements, we make a point of using standard coding-style conventions, so your code will be acceptable to the web community as a whole.

Throughout this chapter and the rest of this book, you'll be exposed to lots of HTML elements. If you're like most people, learning lots of things that are all in a single category can be overwhelming. Defining subcategories can make learning easier. For example, rather than trying to memorize every animal species (an impossible task because thousands of new species are discovered each year), biologists remember categories of species, such as reptiles, mammals, and crustaceans, and assign each species to a particular category. Likewise, before you get overwhelmed with element overload, we describe the categorization scheme used by the World Wide Web Consortium (W3C) for organizing HTML elements. A bit confusing at first, but it'll pay off later.

In this chapter, after we present coding-style conventions and the W3C's element categorization scheme, we present elements that span the width of a web page (block elements) and then elements that can be embedded inside a paragraph (phrasing elements). At the end of the chapter, we take a break from HTML elements and introduce character references. Character references allow you to generate characters that are not on a standard keyboard. For example, if you need to display the half character (½) on a web page, you can do that with a character reference.

2.2 HTML Coding Conventions

Browsers are very lenient in terms of requiring web developers to write high-quality code. So even if a web page's code uses improper syntax or improper style, web browsers won't display an error message; instead, they'll try to render the code in a reasonable manner. You might think that's a good thing, but it's not. If a web page uses improper syntax, different browsers might render the web page differently. In a worst-case scenario, the web developer tests the web page on a browser where no errors are evident, mistakenly concludes that all is well, and publishes the web page on the Web. And then a user loads the web page using a different browser, and that browser renders the page in an inappropriate manner. So as a web developer, how do you deal with this problem? You should test with multiple browsers and check the syntax using the W3C's HTML validation service.

As you may recall, coding-style convention rules pertain to the format of code. For example, there are rules about when to use uppercase versus lowercase, when to insert blank lines, and when to indent. Those rules help programmers understand the code more easily, but the browsers don't care about such things. Consequently, for all those people who create web pages on their own, there's nothing to stop them from using horrible style. If they want to put the code for their entire web page on one line, browsers will treat that code the same as code with proper newlines and indentations. However, if you are taking a course in web programming, your teacher will (I hope) deduct points for poor style. More importantly, if you create web pages for a company, your company will require you to follow their coding-style conventions.

Companies like their programmers to follow standard coding conventions so the resulting programs are easier to maintain (*program maintenance* means debugging and enhancing a program after it has been released initially). This is particularly true for medium- and large-sized companies, where programs are debugged and enhanced by a larger number of people. With more people involved, there's a greater need to understand other people's code, and adhering to standard coding conventions helps with that.

In this book, we attempt to use coding-style conventions that are as widely agreed upon as possible. And how does one find such conventions? It's the same as for everything else in the world—by googling it. If you google "html style guide," you will get the coding-style conventions used by Google, the company. Because Google is ubiquitous, Google's style rules have gained huge support from the web developer community. Consequently, this book uses coding conventions that match Google's coding conventions. In this section, we'll go over some of the more important style rules, but for a more comprehensive description, see Appendix A, HTML5 and CSS Coding-Style Conventions. For now, it's OK to remember just the following style rules:

- For every container element, include both a start tag and an end tag. So even though it's legal to omit a `p` element's end tag, don't do it.
- Use lowercase for all tag names (e.g., `meta`) and attributes (e.g., `name`).
- Use lowercase for attribute values unless there's a reason for uppercase. For a `meta author` element, use title case for the author's name because that's how people's names are normally spelled (e.g., `name="Dan Connolly"`).
- For attribute-value assignments, surround the value with quotes, and omit spaces around the equals sign.

The capitalization rule for the doctype instruction is a gray area. Google's Style Guide says "All code has to be lowercase" except when it's appropriate for a value to use uppercase. Based on that, `<!DOCTYPE html>` should be `<!doctype html>`. However, the vast majority of examples on the W3C and WHATWG websites use uppercase for `DOCTYPE`, and the Google Style Guide uses uppercase for `DOCTYPE`, so that's what we recommend. If you prefer all lowercase for the doctype instruction, ask your boss or teacher if that's OK; if he or she says it is, go for it. Remember—HTML is case insensitive, and browsers will handle either `DOCTYPE` or `doctype` just fine.

The W3C provides a tool named Tidy, at http://services.w3.org/tidy/tidy, which can be used to apply style rules to a web page. Feel free to play around with Tidy, and make up your own mind whether you want to rely on it for formatting your code or rely on careful keyboarding. You can customize Tidy's style rules to match the rules required by your company or your teacher, but be aware that the customization process is nontrivial. Even if you end up using Tidy for all your formatting needs, you should still understand the style rules so you're comfortable reading other people's code.

2.3 Comments

As a programmer in the real world, you'll spend lots of time looking at and editing other people's code. And, other people will spend lots of time looking at and editing your code. Therefore, everyone's code needs to be understandable. One key to understanding is good comments. *Comments* are words that humans read but the computer skips. More specifically, for web programming, the browser engine skips HTML comments. The *browser engine* is the software inside a web browser that reads a web page's content (e.g., HTML code, image files) and formatting information (CSS), and then displays the formatted content on the screen.

Usually, HTML code is fairly easy to understand, so there is no need for extensive comments. However, sometimes comments are appropriate. The general rule is to include a comment whenever information is needed to clarify something about nearby HTML code. Here's an example:

```
<!-- The following image should be updated once a month. -->
<img src="januaryPicture.gif" width="400" height="250">
```

In this code fragment, which displays a picture on a web page, the first line is a comment. As you can see, to form a comment, surround commented text with `<!--` and `-->` markers. For comments that are short enough to fit on one line, like above, proper style suggests inserting a space immediately after `<!--` and immediately before `-->`. This is an appropriate comment because without it, it would be harder for the web developer to remember to update the picture.

For comments that are too long to fit on one line, proper style suggests putting the `<!--` and `-->` markers on lines by themselves and indenting the enclosed comment text. Here's an example:

```
<!--
  If the user clicks one of the color buttons, that will cause the
  following paragraph's font color to change to the button's color.
-->
```

If you're curious about the comment's subject matter—changing the color of a paragraph's text—be patient. You'll learn how to do that when we cover *JavaScript* later in the book. For now, all you need to know is that JavaScript is a programming language, but it's more powerful than the HTML programming language. For the most part, HTML just enables you (the programmer) to display stuff on your web page. JavaScript adds quite a bit of functionality by enabling you to read user input and update what the web page displays.

In the world of software development, *documentation* refers to a description of a program. That description can be in the form of a document completely separate from the source code (like a user guide), or it can be embedded in the source code itself. Comments are one form of embedded-code documentation. With HTML, `meta` elements provide another form of embedded-code documentation. As explained in the previous chapter, you should normally always include a `meta author` element, so other people in your company know whom to go to when questions arise (or whom to blame when the boss needs a target). The `meta description` and `meta keywords` elements are also popular, but not quite as popular as the `meta author` element.

2.4 HTML Elements Should Describe Web Page Content Accurately

An overarching goal in web programming is to use appropriate HTML elements so your web page's content is described accurately. For example, if you have text that forms what would normally be considered a paragraph, then surround the text with a `p` element, not some other element (like `div`). Likewise, if you want to display words as a heading, use a heading element (`h1`-`h6`), not some other element (like `strong`).

A complementary overarching goal in web programming is to use HTML elements so your web page's content is described fully. For example, if you have a `title` for your web page, it would be legal to enter the title as plain text, and not have it be inside a container. But don't do that. Instead, put the title text inside a `title` element.

So, why is it good practice to describe web page content accurately and fully? It's a form of documentation, and documentation helps programmers understand and maintain the web page code more easily. Another benefit of describing web page content accurately and fully is that it enables you (the programmer) to manipulate the web page more effectively using CSS and JavaScript. For example, if you use `p` elements for all your paragraphs, you can use CSS to make all the paragraphs indented for their first lines. As another example, if you use heading elements (`h1`-`h6`) for all your headings, you can use JavaScript to make all the headings larger when a button is clicked.

So, how is this goal enforced whereby elements are used to accurately and fully describe the web page's content? Unfortunately, there's nothing in the HTML5 standard or in the W3C's HTML validation service that enforces this goal. Consequently, much of the enforcement is left up to programmers' due diligence. For example, the HTML validation service will allow you to surround a paragraph of text with `h1` tags or no tags at all. It's up to you not to do that; instead, you should surround the text with `p` tags.

2.5 Content Model Categories

What Content Is Allowed Within a Particular Container?

Despite the HTML validation service's shortcomings mentioned in the previous section, it's still a helpful tool, and you should use it. It's good at identifying syntax errors, like misspelling tag names. It's also good at containership rules. For example, the `head` container must contain a `title` element, and a `p` container must not contain a `div` container. With lots of elements (around 115), there are lots of containership rules (more than 11,000).[1] Rather than having you remember each of those rules, it's easier to assign elements to certain categories and have those categories be the basis for the containership rules. For a given web page, if an element X contains another element Y, all you have to do is look up Y's category and determine whether element X is allowed to contain elements from Y's category.

FIGURE 2.1 shows the W3C's diagram of the different element categories. The diagram becomes useful when you're writing the code for a container and you want to know which elements are allowed inside the container. For example, suppose you're writing the code for a `head` container. To determine which elements are allowed inside the `head` container, you can read about the `head` element in the W3C standard by going to https://www.w3.org/TR/html51. There, scroll through the table of contents to the `head` element entry (or do a ctrl+f "head element"), and click on its link. That should take you to a description of the `head` element. Read the "content model" section, which says:

> One or more elements of metadata content, of which exactly one is a `title` element and no more than one is a `base` element.[2]

So the `head` container is allowed to include elements that are in the metadata category. Now go to the content model categories diagram (using the URL from Figure 2.1) and hover your mouse

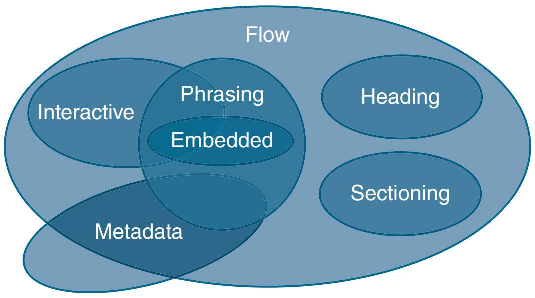


FIGURE 2.1 Content model categories

Reproduced from World Wide Web Consortium (W3C), "W3C HTML 5.1 Recommendation: Semantics, structure, and APIs of HTML documents," last modified November 1, 2016, https://www.w3.org/TR/html51/dom.html#kinds-of-content.

---

[1] "HTML Living Standard," *Web Hypertext Application Technology Working Group (WHATWG)*, last modified June 1, 2017, https://html.spec.whatwg.org/multipage/semantics.html. You can find a complete list of all the HTML elements on this page. The site shows there are 101 container elements and 115 total elements. That means the number of containership relationships is 101 × 115, which equals 11,615.

[2] World Wide Web Consortium (W3C), "W3C HTML 5.1 Recommendation: Semantics, structure, and APIs of HTML documents," last modified November 1, 2016, https://www.w3.org/TR/html51/dom.html#kinds-of-content.

over the metadata oval. That generates a list of all the elements in the metadata category—base, link, meta, noscript, script, style, and title.

For another example, suppose you're writing the code for a p element and you want to know what types of elements are allowed inside of it. How should you proceed? Try to do this on your own before reading the next paragraph.

To determine which elements are allowed inside the p container, look up the p element in the W3C standard and read the "content model" section, which says "phrasing content." The p container is allowed to include elements that are in the phrasing category. Now go to the content model categories diagram and hover your mouse over the phrasing oval. That generates a long list of all the content allowed in the phrasing category. That list includes elements that would be appropriate for describing text/phrases inside a paragraph (to remember what phrasing content is for, remember "phrase"). In addition to showing element names, the list also includes the word "text," which is for plain text devoid of markup tags.

As a third example, let's determine what's allowed inside the hr container. When you look up the hr element in the W3C standard and read the "content model" section, it says "empty." That means that the hr element is a void element, so it doesn't have an end tag or any enclosed content.

We'll describe Figure 2.1's categories in depth soon enough, but first note how the categories overlap. If two categories overlap, that means that the categories include some elements that are in both categories. For example, because the interactive and phrasing categories overlap, that means some of the elements in the interactive category are also in the phrasing category. If a category is completely inside another category, then all the enclosed category's elements are also in the surrounding category. For example, because the interactive, phrasing, embedded, heading, and sectioning categories are all inside the flow category, that means all the elements in the enclosed categories are also in the overarching flow category.

Content Model Category Descriptions

In this subsection, we describe each content model category shown in Figure 2.1. Let's start with the metadata category. The metadata category includes elements that provide information associated with the web page as a whole. That should sound familiar. That's the same description we used for the head container's contents. So an alternative definition of the metadata category is that it includes all the elements that are allowed in the head container.

The flow category includes plain text and all the elements that are allowed in a web page body container. As you can imagine, there are lots of elements in the flow category. We'll discuss quite a few of them later in this chapter, but here's a small sample for now—blockquote, div, hr, p, pre, script, sup. The blockquote, div, hr, p, and pre elements are flow content elements, and they are not in any other content model categories. The script element is for JavaScript. It's in the flow content category as well as the metadata content category (note the intersection of those two categories in the content model categories diagram). You'll normally use script in a head container, but it's legal to use it in a body container as well. The sup element is for superscripting. It's in the flow content category as well as the phrasing content category (note the intersection of those two categories in the content model categories diagram).

We introduced the phrasing category earlier. Here are the phrasing category elements we'll describe later in this chapter—`abbr`, `b`, `br`, `cite`, `code`, `del`, `dfn`, `em`, `i`, `ins`, `kbd`, `mark`, `q`, `s`, `samp`, `small`, `span`, `strong`, `sub`, `sup`, `time`, `u`, `var`, `wbr`.

The embedded category includes elements that refer to a resource that's separate from the current web page. For example, the `audio` element uses an audio file. Here are the embedded category elements we'll describe later in the book—`audio`, `canvas`, `iframe`, `img`, and `video`.

The interactive category includes elements that are intended for user interaction. For example, the `textarea` element displays a box in which the user can enter text. Here are the interactive category elements we'll describe later in the book—`a`, `button`, `input`, `select`, `textarea`.

The heading category includes elements that define a header for a group of related content. For example, the `h1` element displays a large header, which would normally go above content that is associated with the header. We already covered the following heading category elements in the previous chapter—`h1`, `h2`, `h3`, `h4`, `h5`, `h6`.

The sectioning category includes elements that define a group of related content. For example, the `aside` element is for content that's not part of the web page's main flow. Here are the sectioning category elements we'll describe later in the book—`article`, `aside`, `nav`, `section`.

Now that you've learned about the various content model categories and the content model category diagram, you might feel pretty good about being able to apply the containership rules correctly. But alas, we're human, and we make mistakes every now and then. Therefore, when coding a web page, you should always double-check your work by running the W3C's HTML validation service.

2.6 Block Elements

We'll now introduce an element category that is not part of the HTML5 standard. The category is for *block elements*. Even though "block element" is not an official term blessed by the W3C, we'll use it throughout the book because it will make certain explanations easier. A block element expands to fill the width of its container, so for a given container, there will be only one block element for each row in the container. For every example in the first part of this book, each block element's container is the `body` element, which spans the width of the browser window. So for those examples, the block element also spans the width of the entire browser window. That's different from a *phrasing element* in that (1) a phrasing element's width matches the width of the element's contents and (2) multiple phrasing elements can display in one row. If you're curious, there is a rather convoluted relationship between block elements and the W3C's content model categories: A block element corresponds to an element in the flow category that is not also an element in the phrasing category.

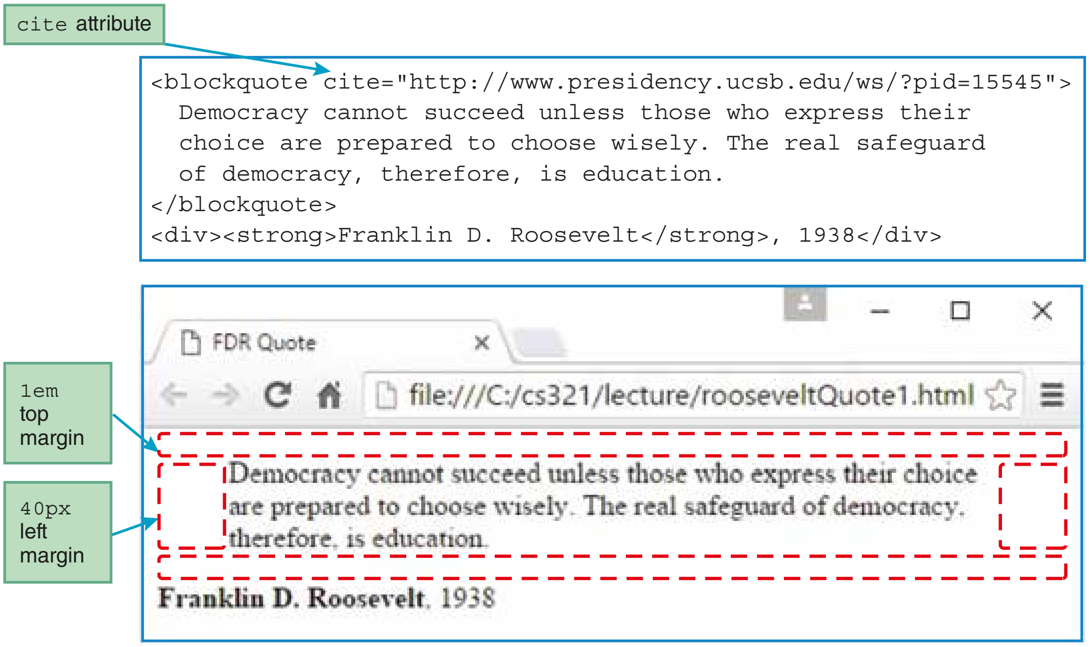
Be aware that a similar term, "block-level element," was part of the HTML4 standard, but it's been omitted from the HTML5 standard. Why? It's probably because the W3C feels that HTML should focus exclusively on content and let the browsers and CSS determine an element's formatting. With its focus on spanning the width of its container, the W3C deemed the block-level element category to be too format-oriented. For block-level element fans, it's disappointing that "block-level element" is no longer part of the official HTML lexicon. However, the term is not completely dead. The W3C's CSS standard uses a `block` value for the CSS `display` property (which we'll get to in a later chapter). Mozilla still uses "block" to describe

HTML concepts,[3] and in its coding-style guide, Google uses the similar term “block element” as a synonym for “block-level element.”[4] So as not to anger our Google overlords, we follow suit and use the term “block element” instead of “block-level element.”

2.7 `blockquote` Element

We’ve already talked about the `div` and `p` elements, which are block elements. Now let’s discuss another block element—the `blockquote` element. You should use a `blockquote` element when you have a quotation that is too long to embed within surrounding text. It’s a block element, so it spans the width of its container. More precisely, its content spans the width of the nonmargin part of its enclosing container.

For a `blockquote` element example, see **FIGURE 2.2**. In the figure’s browser window, note the margins on the four sides of the quote text. Most browsers render a `blockquote` element by displaying those margins. But as an alternative, a *browser vendor* (an organization that implements a browser) may render a `blockquote` element by displaying the text with italics and not with margins.



```
<blockquote cite="http://www.presidency.ucsb.edu/ws/?pid=15545">
  Democracy cannot succeed unless those who express their
  choice are prepared to choose wisely. The real safeguard
  of democracy, therefore, is education.
</blockquote>
<div><strong>Franklin D. Roosevelt</strong>, 1938</div>
```

Democracy cannot succeed unless those who express their choice are prepared to choose wisely. The real safeguard of democracy, therefore, is education.

Franklin D. Roosevelt, 1938

FIGURE 2.2 An example blockquote

[3] “Block-level elements,” *Mozilla Developer Network*, last modified April 21, 2017, https://developer.mozilla.org/en-US/docs/Web/HTML/Block-level_elements.
[4] “Google HTML/CSS Style Guide: 2.2 General Formatting Rules,” *Google.com*, http://google.github.io/styleguide/htmlcssguide.html#General_Formatting_Rules.

Typical Default Display Properties

For each element, the W3C's HTML5 standard provides a "typical default display properties" section that describes the typical format used by the major browsers in displaying the element. Browsers are not forced to follow those guidelines, but they usually do, and as a developer, you should pay attention to the guidelines. For example, **FIGURE 2.3** shows the typical default display properties for the `blockquote` element.

Do you recognize the format of Figure 2.3's code? It's CSS. The figure shows five CSS rules that are commonly used as defaults when a browser renders a `blockquote` element. The first CSS rule says to use a `block` value for the `display` property. That means that the element the rule applies to, `blockquote` in this case, will span the width of its container. Thus, the `display: block` property-value pair matches the characteristics of the block element described earlier.

The second and third CSS rules apply to the top and bottom margins. The `1em` values cause each of the two margins to be the height of one line of text. We'll discuss the CSS `em` unit in more depth in the next chapter, but for now, note the resulting blank lines above and below the `blockquote` text in Figure 2.2's browser window.

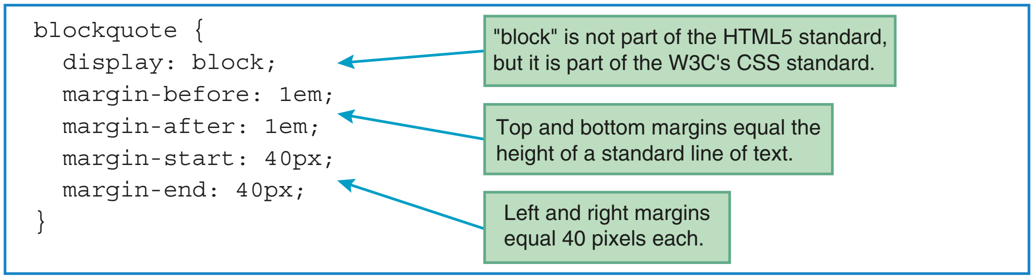
The fourth and fifth CSS rules apply to the left and right margins. The `40px` values cause each of the two margins to be 40 pixels wide, where 1 *pixel* is the size of an individually projected dot on a typical computer monitor. We'll discuss the CSS `px` unit in more depth in the next chapter, but for now, note the resulting margins at the left and right of the `blockquote` text in Figure 2.2's browser window.

`cite` Attribute

In Figure 2.2's `blockquote` code, did you notice the `cite` attribute in the element's start tag? For your convenience, here's the start tag again:

```
<blockquote cite="http://www.presidency.ucsb.edu/ws/?pid=15545">
```

The purpose of the `cite` attribute is to document where the quote can be found on the Internet. The `cite` attribute's value must be in the form of a URL. Interestingly, browsers do not display the `cite` attribute's value. That's because the URL value is not for end users. Instead, it serves as documentation for the web developer(s) in charge of maintaining the web page. Presumably, the web developer would check the URL every now and then to make sure it's still active.



```
blockquote {
  display: block;
  margin-before: 1em;
  margin-after: 1em;
  margin-start: 40px;
  margin-end: 40px;
}
```

"block" is not part of the HTML5 standard, but it is part of the W3C's CSS standard.
Top and bottom margins equal the height of a standard line of text.
Left and right margins equal 40 pixels each.

FIGURE 2.3 Typical default display properties for the `blockquote` element

Besides providing documentation, another benefit of including the cite attribute is that it can be used as a "hook" for adding functionality to the blockquote element. Specifically, a web programmer could add JavaScript code that uses the cite attribute's URL value to perform some URL-related task (e.g., jumping to the URL's web page when the user hovers his or her mouse over the quote). That will make more sense when we talk about JavaScript later in the book.

By the way, if you think the user might be interested in visiting the web page where the quote came from, you can implement a link. We'll describe how to implement links, using <a> and </a> tags, in Chapter 4. If you use a link, you may or may not want to also include a cite attribute for your blockquote element.

Block Formatting

For a blockquote element with enclosed text that's greater than one line, you should use *block formatting*. Block formatting is a coding-style convention where the start and end tags go on their own lines and the enclosed text is indented. For an example, see Figure 2.2's blockquote element code, copied here for your convenience:

```
<blockquote cite="http://www.presidency.ucsb.edu/ws/?pid=15545">
  Democracy cannot succeed unless those who express their
  ...
</blockquote>
```

In the previous chapter, we covered the p and div elements. Like the blockquote element, they are block elements, so they span the width of their containers. For a p element example, see Figure 1.4's p element code, copied here for your convenience:

```
<p>
  It should be pleasant today with a high of 95 degrees.<br>
  With a humidity reading of 30%, it should feel like 102 degrees.
</p>
```

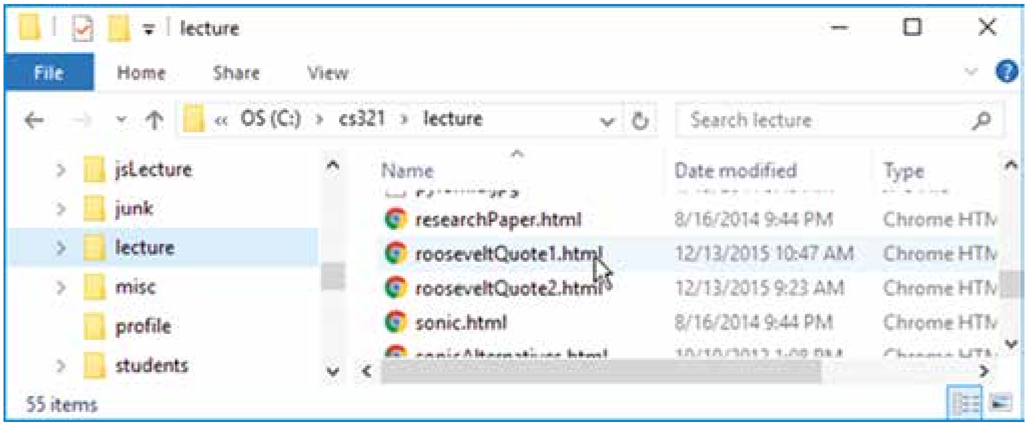
For a div element example, see Figure 2.2's div element code, copied here for your convenience:

```
<div><strong>Franklin D. Roosevelt</strong>, 1938</div>
```

The p and div elements are both block elements. So, in these code fragments, why does the p element use block formatting, but the div element does not? The block formatting style rule says to use block formatting for all block elements with content longer than one line. In the preceding p example, the content (plain text) is longer than one line, so block formatting is used. In the preceding div example, the content (a strong element plus plain text) is shorter than one line, so block formatting is not used.

Displaying a Web Page Without a Web Server

We'll get back to our discussion of block elements shortly, but for now we should point out something you might have noticed in the Franklin Roosevelt blockquote web page shown earlier. In Figure 2.2, note the URL value in the browser window's address bar—file:///C:/cs240/lecture/rooseveltQuote1 .html. The "file" at the beginning of the URL is the protocol. When you see a "file" protocol, that means the web page was generated by simply double clicking on its `.html` file from within Microsoft's File Explorer tool. For example, in the following File Explorer screenshot, imagine double clicking on the `rooseveltQuote1.html` file. That's how we generated Figure 2.2's web page.



Name	Date modified	Type
researchPaper.html	8/16/2014 9:44 PM	Chrome HTM
rooseveltQuote1.html	12/13/2015 10:47 AM	Chrome HTM
rooseveltQuote2.html	12/13/2015 9:23 AM	Chrome HTM
sonic.html	8/16/2014 9:44 PM	Chrome HTM

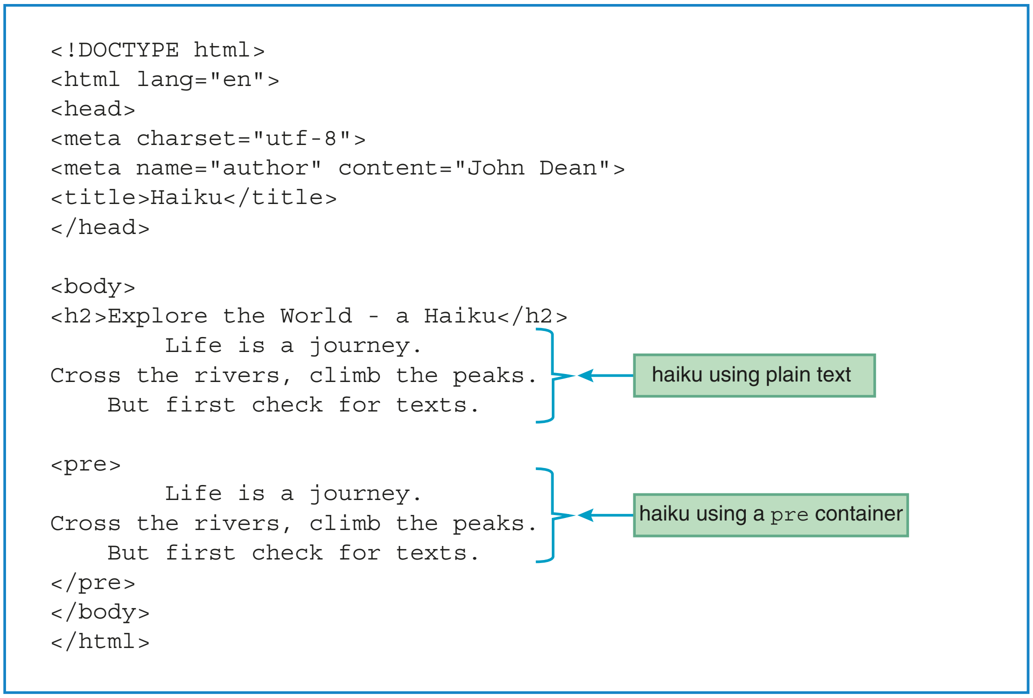
As explained in Chapter 1, if you want a web page to be accessible to everyone on the Web, you'll need to upload its `.html` file to a web server. But for a quick test, it's often easier to generate a web page by just double clicking on its file.

2.8 Whitespace Collapsing

The next block element we'll describe is the `pre` element. But for the `pre` element to make sense, we first need to explain whitespace collapsing. *Whitespace* refers to characters that are invisible when displayed on the browser window. The most common whitespace characters are the blank, newline, and tab characters. The web developer generates those characteristics by pressing the spacebar, enter, and tab keys, respectively. Normally, browsers collapse whitespace. In other words, if your HTML code contains consecutive blank spaces, newlines, or tabs, the browser will display the web page with only one whitespace character (usually a blank space).

For an example of whitespace collapsing, let's look at a haiku web page. A haiku is a form of Japanese poetry that consists of three lines—five syllables for the first line, seven syllables for the second line, and five syllables for the third line. In **FIGURE 2.4**, examine the text that comprises the plain text haiku. See how the three lines are centered horizontally? That's common for haikus. In Figure 2.5, note how the plain text haiku is displayed. In particular, note that the haiku's whitespace gets collapsed so that the resulting haiku is no longer centered or on three lines (a major faux pas for haiku fashionistas).

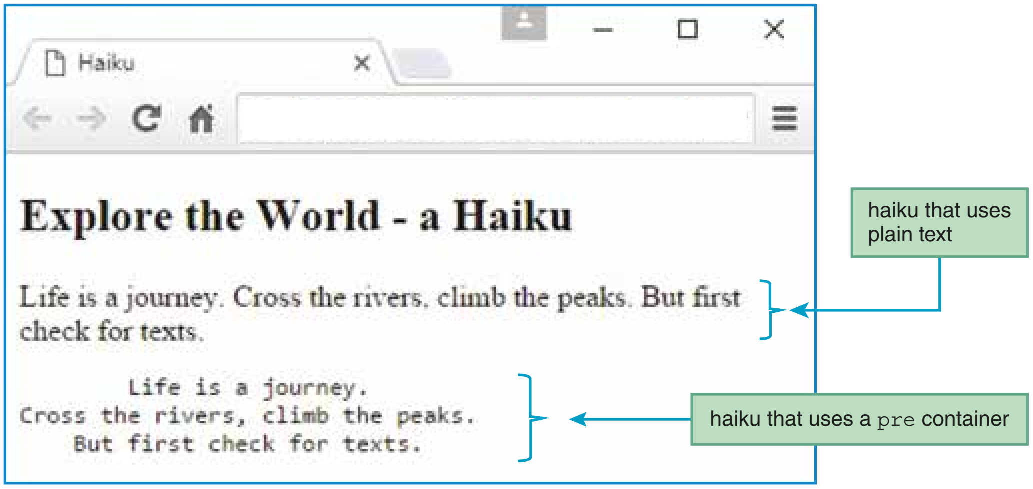
In **FIGURE 2.5**, do you see the newline after "But first"? That is not from collapsing whitespace. The only reason the browser inserts a newline after "But first" is because of line wrap. *Line wrap* is when a word bumps up against the right margin and is automatically moved to the next line.



```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="author" content="John Dean">
<title>Haiku</title>
</head>

<body>
<h2>Explore the World - a Haiku</h2>
        Life is a journey.
Cross the rivers, climb the peaks.
    But first check for texts.

<pre>
        Life is a journey.
Cross the rivers, climb the peaks.
    But first check for texts.
</pre>
</body>
</html>
```

haiku using plain text
haiku using a pre container

FIGURE 2.4 **Source code for Haiku web page**



Explore the World - a Haiku

Life is a journey. Cross the rivers, climb the peaks. But first check for texts.

```
        Life is a journey.
Cross the rivers, climb the peaks.
    But first check for texts.
```

FIGURE 2.5 **Haiku web page**

In Figure 2.5, the browser collapses whitespace within the plain text haiku, but the browser preserves whitespace for the rest of the web page. Why is that? Above the plain text haiku, there's a blank line. That's from the preceding `h2` element, which displays a noncollapsing blank line above and below its text. Below the plain text haiku, there's another blank line. That's from the `pre` element, introduced in the next section, which also displays a noncollapsing blank line above and below its text.

Whitespace collapsing can be helpful in many circumstances, but not all. For certain forms of literature, like haikus, line breaks and indentations need to be preserved. Later, we'll show another situation where whitespace needs to be preserved—displaying programming code. The `pre` element takes care of those situations.

2.9 `pre` Element

You should use the `pre` element for text that needs to have its whitespace preserved. Formally, `pre` stands for "preformatted text." However, we prefer to pretend that it stands for "preserved whitespace" because that makes more sense. In Figure 2.5, take a look at the bottom haiku (the one that uses a `pre` container). Note the blank spaces and newlines. Those are whitespace characters from the source code, and we can thank the `pre` container for preserving them.

Also in Figure 2.5, note the bottom haiku's monospace font. *Monospace font* is when each character's width is uniform. By default, browsers display `pre` element text with monospace font. If you don't like the default monospace font, you can use CSS to change the `pre` element text's font. That will make more sense when we introduce CSS's `font` property in the next chapter.

2.10 Phrasing Elements

Remember phrasing elements? If not, glance back at the content model categories diagram in Figure 2.1.

Phrasing elements are meant for text items that would be deemed acceptable within a typical paragraph. For example, the `strong` element, introduced in Chapter 1, is a phrasing element. That should make sense when you realize that its purpose is to place emphasis on a word or group of words within a paragraph.

Just because a phrasing element is defined as something that is "deemed acceptable within a typical paragraph" doesn't mean that phrasing elements can be found only within `p` containers. On the contrary, phrasing elements are allowed within many container elements besides the `p` container. In determining whether it's appropriate to use a phrasing element within a given container, think about whether it would be reasonable to put the phrasing element's text within that type of container. For example, in Figure 2.4, the second haiku uses a `pre` container. In that `pre` container, would it be reasonable to surround "check for texts" with a `strong` container? With texting found to be a "necessary component to

Cell phone enthusiasts extending their Snapchat streaks

sustain life,"[5] the answer is an unequivocal yes, and the HTML5 standard does indeed allow `strong` elements within `pre` containers.

As an aside, you can apply the "Would it be reasonable?" test to guess the containership rule for any two container elements. For example, would it be reasonable to embed a `p` container within another `p` container? No. So if you nest two `p` elements and then test your code with the HTML5 validation service, you'll get an error.

In talking about phrasing elements, you should be aware that the term *inline* is sometimes used to describe their nature. That should make sense: It's reasonable for a phrasing element to be contained in a paragraph, so it could be thought of as being "in" one of the paragraph's "lines." Thus the term "inline." Although the W3C appears to have abandoned its use of the term "inline" in its HTML5 standard, the W3C still uses `inline` in its CSS standard as a value for elements that are to be formatted as phrasing elements—where the element's width matches the width of its contents (not the width of its container, like a block element).

The remainder of this chapter focuses on some of the phrasing elements. We'll present about half of them in this chapter, and we'll present some of the other ones later on.

2.11 Editing Elements

We begin our discussion of individual phrasing elements with two elements that are used to indicate editing changes—the `ins` element (for insertions) and the `del` element (for deletions). First, let's verify that they are indeed phrasing elements. Go to the W3C's content model categories page (https://www.w3.org/TR/html51/dom.html#kinds-of-content) and hover your mouse over the diagram's phrasing area. That should cause a list of all the phrasing elements to appear. In the list, look for the `ins` and `del` elements.

The `ins` element is meant to indicate text that has been inserted. If you're an editor and you're reviewing someone else's written work, you'll probably have suggestions for inserted text every now and then. To make the suggested text stand out, you should format it differently from the original text. That way, the original writer can quickly identify what has been suggested. The `ins` element works the same way.

Typically, browsers display an `ins` container's text with underlining. However, it's up to each browser vendor to determine an element's presentation when it's rendered. *Presentation* refers to the appearance and format of a displayed element. You can assume that browsers will display the text with an appropriate appearance, but that appearance might be different for different browsers.

The `del` element is meant to indicate text that has been deleted. Typically, browsers display a `del` element with strikethrough text. To see an example of that, go to the W3Schools' HTML5 tag reference page at https://www.w3schools.com/tags, click the del tag's link, and then click the **Try it yourself** link. On the try-it-yourself page, enter this:

```
HTML is <del>boring.</del><ins>super exciting!</ins>
```

[5] Research findings from 2017 study, "You CANNOT Take Away My Phone!" Research participants: Jordan Dean, Caiden Dean.

After clicking **See Result**, you should see this:

HTML is ~~boring.~~super exciting!

2.12 `q` and `cite` Elements

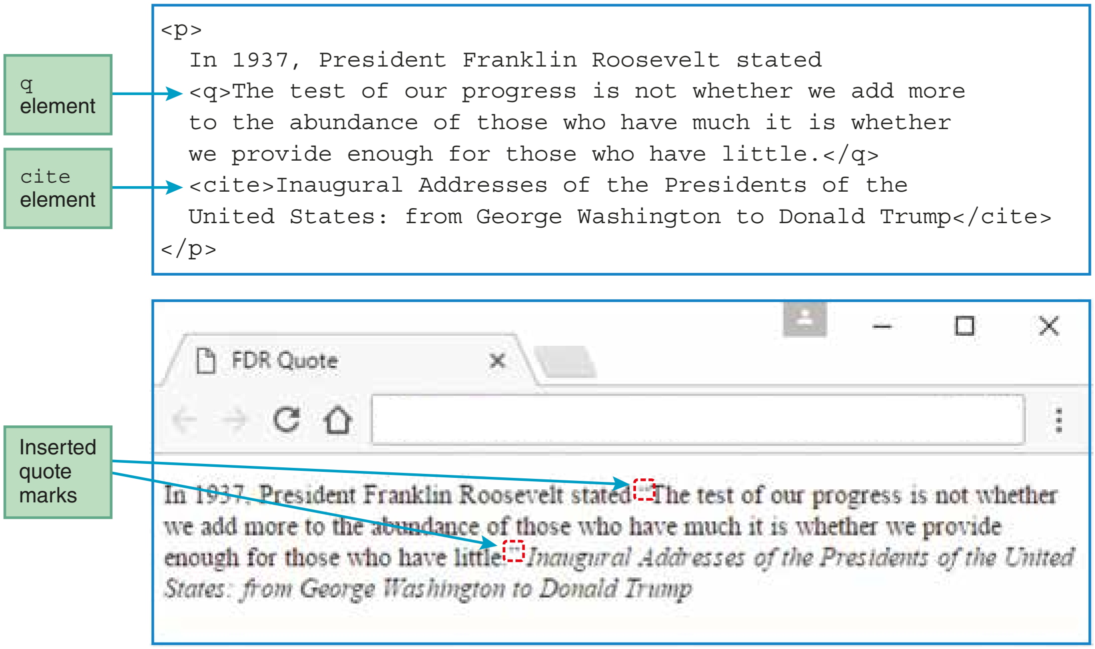
`q` Element

Now for another phrasing element—the `q` element. It's for quoted text that is to be rendered within the flow of surrounding text. That's different from the `blockquote` element, which spans the width of its container. Normally, browsers display a `q` element by surrounding its text with quotes. In **FIGURE 2.6**, note that there are no quote marks in the code fragment, and note the inserted quote marks in the resulting web browser.

For normal writing, if you have quoted text, the opening quote mark should be right next to the first character in the quoted text, and the closing quote mark should be right next to the last character in the quoted text. When implementing a web page with quoted text, it's a nonissue most of the time because the code that supports that format tends to be intuitive. However, suppose Figure 2.6's code fragment was written as follows, with the `<q>` start tag moved up to the end of the previous line:

```
<p>
  In 1937, President Franklin Roosevelt stated <q>
  The test of our progress is not whether we add more
```

This has been moved up.



```
<p>
  In 1937, President Franklin Roosevelt stated
  <q>The test of our progress is not whether we add more
  to the abundance of those who have much it is whether
  we provide enough for those who have little.</q>
  <cite>Inaugural Addresses of the Presidents of the
  United States: from George Washington to Donald Trump</cite>
</p>
```

In 1937, President Franklin Roosevelt stated “The test of our progress is not whether we add more to the abundance of those who have much it is whether we provide enough for those who have little” *Inaugural Addresses of the Presidents of the United States: from George Washington to Donald Trump*

FIGURE 2.6 **Code fragment with `q` and `cite` elements, plus resulting browser window**

Such a move seems reasonable because there is plenty of room at the end of the previous line. But what do you think happens when the browser renders the code? First, the browser renders the `<q>` start tag as an opening quote mark. So far, so good. Next, the browser sees the newline and because of whitespace collapsing, it displays a single space before "The test…". Not so good. The moral of the story is to make sure you put the `<q>` start tag right next to the first character in the quoted text, and you put the `</q>` end tag right next to the last character in the quoted text.

By the way, this same issue does not apply to the `p` element. In the preceding code, even though the `<p>` start tag is on a different line from the paragraph's first character, the browser displays no whitespace at the left of the paragraph's first character. That's because `p` is a block element, and testing shows that browsers remove whitespace at the left and right of a block element's text.

cite Element

When you display quoted text, if the text comes from a "work," you should cite the work's title using the `cite` element. In defining the `cite` element at https://www.w3.org/TR/html51/textlevel-semantics.html#the-cite-element, the W3C states that a "cite element represents the cited title of a work; for example, the title of a book, paper, essay, poem, score, song, script, film, TV show, game, sculpture, painting, theater production, play, opera, musical, exhibition, legal case report, or other such work." Typically, browsers display a `cite` container's text with italics.

It's common for a `cite` element to follow a `q` element, which is what the example in Figure 2.6 shows. As an alternative, you can have a `cite` element follow a `blockquote` element. As another alternative, you can have a `cite` element appear within a `blockquote` element, after the `blockquote` element's text.

Previously, we talked about the `cite` attribute as part of the `blockquote` element. What is the difference between the `cite` element and the `cite` attribute? The `cite` element is for a cited work, like a book title, whereas the `cite` attribute is strictly for a URL value. Another difference is that browsers display the content in a `cite` element, whereas browsers do not display the content in a `cite` attribute.

2.13 dfn, abbr, and time Elements

In this section, we introduce three elements that aren't used all that often, but they are helpful if you want to manipulate definitions, abbreviations, dates, and times with CSS or JavaScript.

dfn Element

The `dfn` element is for a word or expression that is to be defined. It's not for the definition, but rather the thing that is being defined. So, in the following example, `dfn` tags surround "tooltip," the word being defined, and not "is a pop-up …," which is the definition.

```
<p>
  A <dfn>tooltip</dfn> is a pop-up box that provides information
  about the item that the mouse is hovering over.
</p>
```

You might think that dfn stands for "definition." It's certainly possible that dfn stood for "definition" when HTML was invented. However, the HTML5 standard states that dfn stands for "<u>d</u>e<u>f</u>i<u>n</u>ing instance," not "definition." "Defining instance" is a mouthful, but it matches the dfn element's functionality pretty well.

Typically, browsers display a dfn element's text with italics. That coincides with standard writing practice, which says to italicize a word if it's being defined.

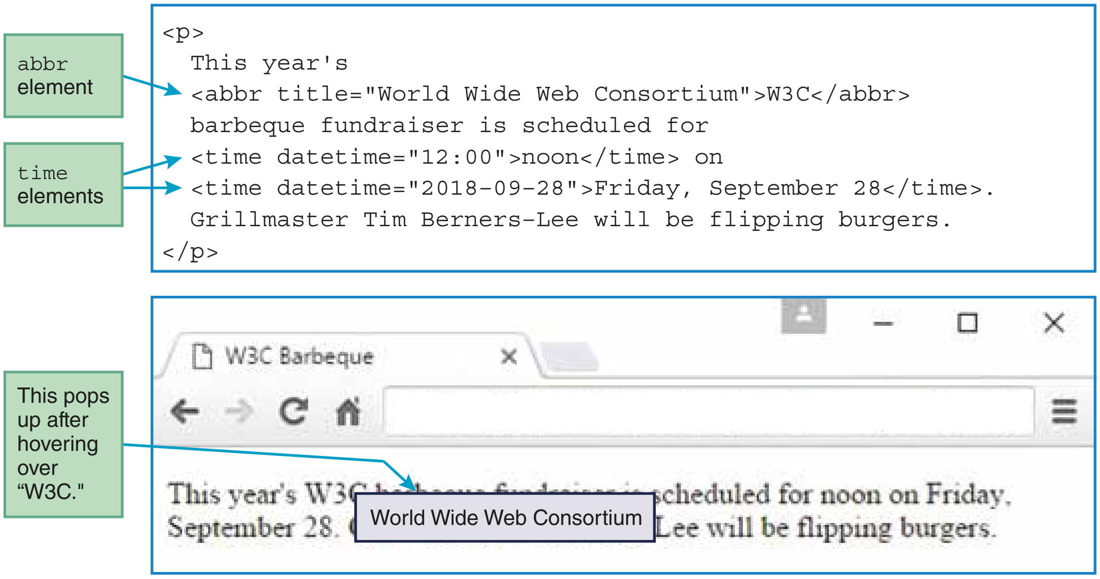
abbr Element

The abbr element is for an abbreviation or acronym. For an example, see **FIGURE 2.7**, which surrounds the W3C acronym with abbr tags. In the abbr element's start tag, note the title attribute. It provides the expanded text that the abbreviation represents. So, for the W3C abbr element, its title attribute has the value "World Wide Web Consortium."

Typically, browsers display the abbr element's title value as a *tooltip*. That means that when the mouse hovers over the abbr element's displayed text, the title attribute's value pops up. In Figure 2.7, you can see the "World Wide Web Consortium" tooltip pop-up after the user has hovered the mouse over the abbr element's displayed text, "W3C."

For the W3C Barbeque web page, suppose you want to show not only the expanded form of "W3C" (with the abbr element's title attribute), but you also want to provide a definition of the W3C organization. To do that, you should surround the abbr element with a dfn element, like this:

```
The <dfn><abbr title="World Wide Web Consortium">W3C</abbr></dfn>
is the governing body for various technologies on the Web. It
hosts an annual barbeque fundraiser….
```


```
<p>
  This year's
  <abbr title="World Wide Web Consortium">W3C</abbr>
  barbeque fundraiser is scheduled for
  <time datetime="12:00">noon</time> on
  <time datetime="2018-09-28">Friday, September 28</time>.
  Grillmaster Tim Berners-Lee will be flipping burgers.
</p>
```

This year's W3C ... scheduled for noon on Friday, September 28. (World Wide Web Consortium Lee will be flipping burgers.

FIGURE 2.7 **Code fragment with abbr and time elements, plus resulting browser window**

time Element

The time element is used to indicate that its text represents a date or time. By using a time element instead of just plain text, you enable the browser engine to recognize the text as a time or date value. That recognition enables CSS and JavaScript to read and/or manipulate the date/time value if there is a need to do so.

The time element's datetime attribute provides a date/time value in a format that the browser engine can understand. The most common format for a date is yyyy-mm-dd. Figure 2.7 shows an example that uses that format. The relevant code is copied here for your convenience:

```
<time datetime="2018-09-28">Friday, September 28</time>
```

The most common format for a time is hh:mm. Figure 2.7 shows an example that uses that format, and here's the relevant code:

```
<time datetime="12:00">noon</time>
```

Here are two additional examples:

```
<time datetime="2019-10-01">Tuesday, October 1</time>
<time datetime="04:00">4 AM</time>
```

Note how the examples follow the preceding formats—4 digits for the year, and 2 digits each for the month, day, hour, and minutes values. If you provide date and time values with fewer digits than 4, 2, 2, 2, and 2, you can't be sure that the browser engine will interpret the values correctly.

Date and time values are not mutually exclusive. It's legal to specify a date and time with one datetime attribute by separating the date and time values with a T. For example:

```
Join us for a rockin' <time datetime="2019-01-01T00:00">New Year's
Eve</time> "Bingo Bash" at the Seniors Center.
All you can drink spiced apple cider!
```

It's legal to omit the datetime attribute, but if you do so, the enclosed content between the time element's tags must use one of the formats prescribed in the HTML5 standard. The preceding formats (yyyy-mm-dd and hh:mm) are in the HTML5 standard, and here are examples that use those formats for the time element's enclosed content:

```
<time>1967-07-15</time>
<time>04:00</time>
```

Browsers do not display the datetime attribute's value, so normally you should include a date and/or time value between the start and end tags, as shown in all the preceding examples. Typically, browsers display a time element's enclosed text with default formatting, which means that the format comes from the element that surrounds the time element.

2.14 Code-Related Elements

As a programmer, you'll sometimes want to look up coding syntax and coding examples on the Web. It's amazing how much code is out there. To accommodate web developers who want to make web pages that show code, the HTML5 standard provides several coding-related elements—`code`, `kbd`, `samp`, and `var`.

The `code` element indicates that its enclosed text is programming code. The `kbd` element indicates that its enclosed text represents input for a program. The "kbd" stands for "keyboard." The `samp` element indicates that its enclosed text represents output for a program. The "samp" stands for "sample." The `var` element indicates that its enclosed text represents a programming variable or a mathematical variable. The "var" stands for "variable."

For an example with all four code-related elements, see **FIGURE 2.8**. It shows a JavaScript code fragment with sample input and output. Specifically, the web page says that if you run the program with "1992" for input, you'll get "The earliest you can receive social security benefits is 2054." If you study the JavaScript code (we'll explain JavaScript syntax later in the book), you'll see that the program adds 62 to the user's entered birth year, and the result is the first year that the user is eligible to receive social security benefits.

Typically, browsers display `code` element text with monospace font. In **FIGURE 2.9**, note the monospace font used for the `code` element. Monospace font is appropriate for `code` element text because when you look at code within a text editor or an IDE, the code uses monospace font. So, on a web page, if you display programming code with monospace font, your code will look more realistic.

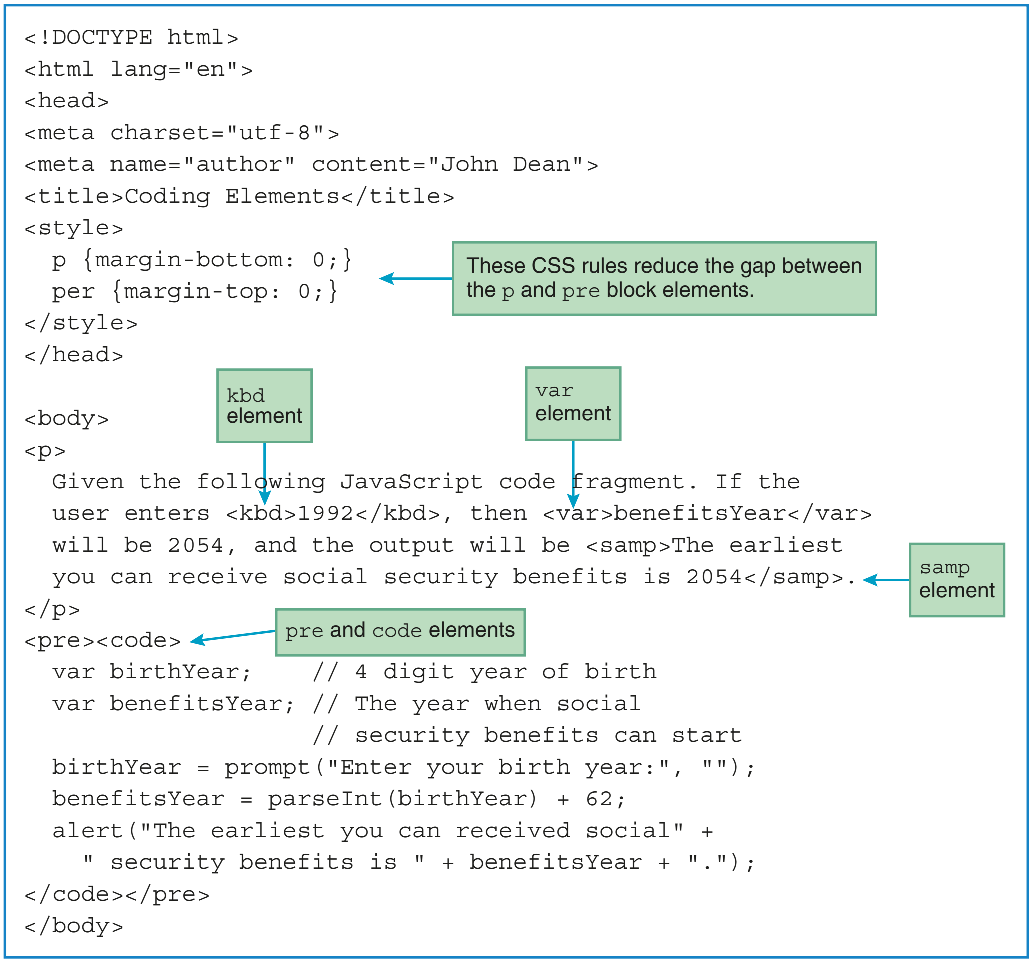
Another reason why monospace is appropriate for displaying code is that it enables you to align similar types of things within your program. For example, in Figure 2.9, note the first three lines in the code fragment, copied here for your convenience:

```
var birthYear;    // 4 digit year of birth
var benefitsYear; // The year when social
                  // security benefits can start
```

The preceding code matches what you'd see in an IDE. Because of the monospace font, it was easy to align the `//`'s. Just press the spacebar until the `//`'s line up. On the other hand, with the exact same keystrokes as before, if you use a non-monospace font (like Times New Roman), here's what the code looks like:

```
var birthYear;    // 4 digit year of birth
var benefitsYear; // The year when social
                  // security benefits can start
```

The reason the `//`'s don't align like they used to is because the space characters are narrower than the letter characters. If you attempt to fix the alignment in the first and third lines by inserting more space characters in front of the `//`'s, you probably won't be able to align the `//`'s perfectly. Even if inserting spaces does generate perfect alignment, the resulting code



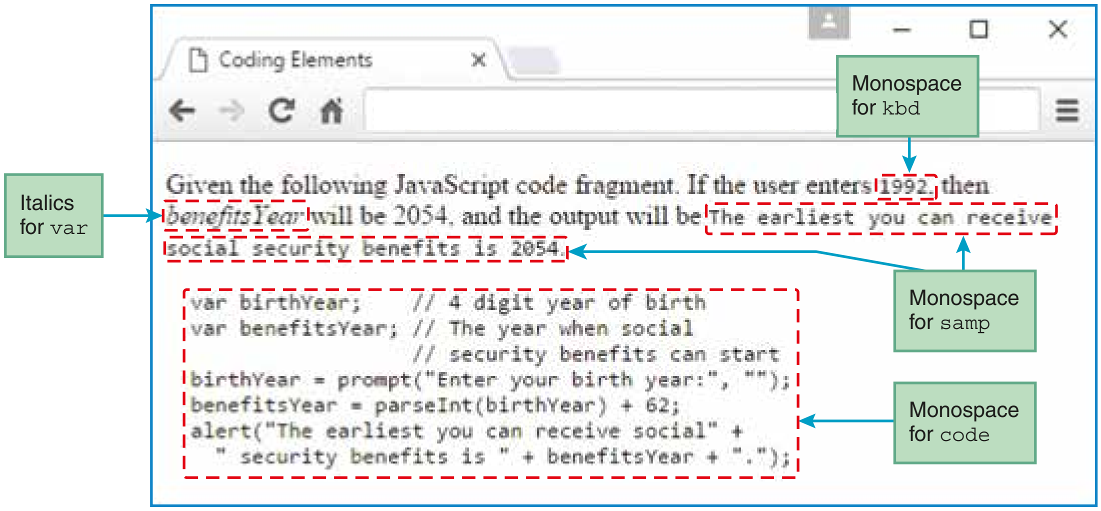
```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="author" content="John Dean">
<title>Coding Elements</title>
<style>
  p {margin-bottom: 0;}
  per {margin-top: 0;}
</style>
</head>

<body>
<p>
  Given the following JavaScript code fragment. If the
  user enters <kbd>1992</kbd>, then <var>benefitsYear</var>
  will be 2054, and the output will be <samp>The earliest
  you can receive social security benefits is 2054</samp>.
</p>
<pre><code>
  var birthYear;    // 4 digit year of birth
  var benefitsYear; // The year when social
                    // security benefits can start
  birthYear = prompt("Enter your birth year:", "");
  benefitsYear = parseInt(birthYear) + 62;
  alert("The earliest you can received social" +
    " security benefits is " + benefitsYear + ".");
</code></pre>
</body>
```

These CSS rules reduce the gap between the p and pre block elements.
kbd element
var element
samp element
pre and code elements

FIGURE 2.8 Source code for Coding Elements web page

will be harder to maintain. For example, assume you have perfect alignment and then you change “benefitsYear” to “benefitYear.” To fix the newly introduced misalignment, you might try deleting one or two spaces from each of the other two lines, but that would probably not work perfectly.

As with the `code` element, browsers typically display `kbd` and `samp` text with monospace font. In Figure 2.9, note the monospace font used for the `kbd` and `samp` elements. The rationale for monospace font for `kbd` and `samp` text is a bit tenuous. Today, most programs display input and output text with fonts that are customizable, and those fonts are usually not in the monospace font category. However, in the old days, programs were rather boring; they were limited to monospace font when displaying input and output. The `kbd` and `samp` elements follow that tradition, and that’s why they use monospace font by default.



Given the following JavaScript code fragment. If the user enters 1992, then *benefitsYear* will be 2054, and the output will be The earliest you can receive social security benefits is 2054.

```
var birthYear;    // 4 digit year of birth
var benefitsYear; // The year when social
                  // security benefits can start
birthYear = prompt("Enter your birth year:", "");
benefitsYear = parseInt(birthYear) + 62;
alert("The earliest you can receive social" +
  " security benefits is " + benefitsYear + ".");
```

FIGURE 2.9 **Coding Elements web page**

As mentioned earlier, the `var` element is for a programming variable or a mathematical variable. In math books and in equation editors,[6] variables are written with italics. Following suit, browsers typically display `var` elements with italics. For an example, see the variable *benefitsYear* in Figure 2.9.

In this section, we've been examining the source code for the Coding Elements web page. In that source code, there's one last noteworthy item—the CSS rules shown in Figure 2.8 and copied here for your convenience:

```
p {margin-bottom: 0;}
pre {margin-top: 0;}
```

These rules have nothing to do with code-related elements; they help solve a problem. Note the single-line gap between the opening paragraph and the subsequent JavaScript code. Without the CSS rules, that gap would be too large. The opening paragraph and subsequent JavaScript code segments are implemented with `p` and `pre` elements, respectively. Remember, browsers normally display the `p` and `pre` elements with blank lines above and below them. To get rid of those blank lines, the CSS rules assign zero-height margins below the `p` element and above the `pre` element, respectively. Even with those rules in place, you can see in Figure 2.9 that there's still a single blank line between the `p` element and the `pre` element. That blank line is a result of the newline after `<pre><code>`. Remember that the `pre` element preserves whitespace, so that newline is preserved, thus creating a single blank line.

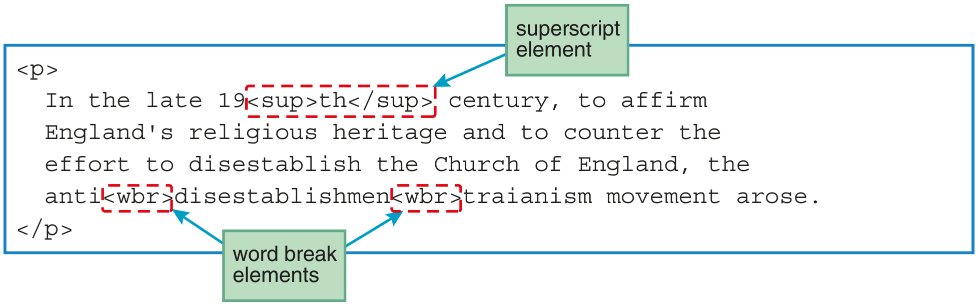
[6] An equation editor is software that helps users write equations within an electronic document.

2.15 `br` and `wbr` Elements

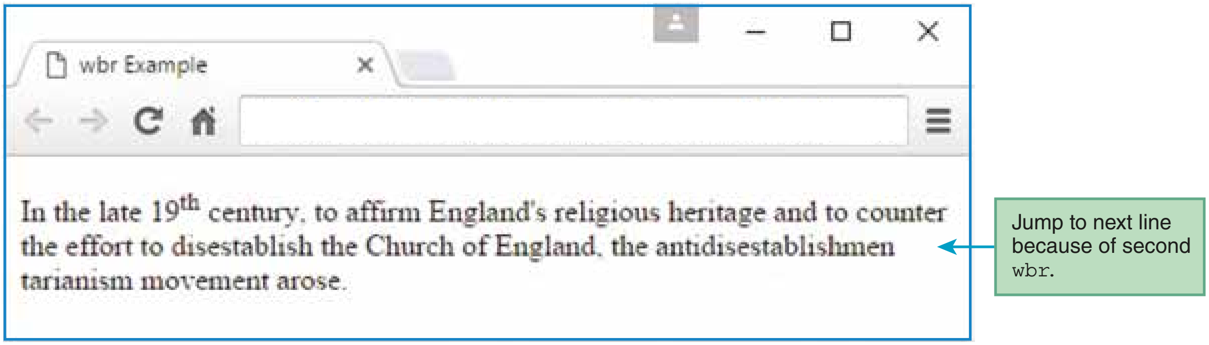
In this section, we discuss two elements that deal with line breaks. You already know about the `br` element, which is a void element that causes subsequent text to start on the next line. The `wbr` element (`wbr` stands for "word break") is similar to the `br` element in that it's a void element. But whereas the browser treats `br` as a required break between words, the browser treats `wbr` as a suggested break within a word if the word bumps into the right side of its containing box. So far, the only "containing box" has been the browser window's main box, defined by the `body` element. Later, you'll see how we can create smaller containing boxes within the browser window's main box.

See **FIGURE 2.10** and note the `sup` element. We'll get to that element in the next section. For now, let's focus on the two `wbr` elements within the word "antidisestablishmentarianism." In the resulting browser window, the second `wbr` element causes the browser to insert a break after the "men" syllable in "antidisestablishmentarianism." If the user resizes the window by dragging the right edge to the left, "men" will bump into the right side of the window, and that will cause the browser to insert a break after "anti" in "antidisestablishmentarianism." That's where the first `wbr` element resides.

Using the `wbr` element for regular words, even long ones like antidisestablishmentarianism, is a bit unusual. A more common scenario is when you have a long sequence of nonblank characters that forms a pattern or code of some sort. A URL matches that description. URLs can be quite long, and their dots (`.`) and slashes (`/`) form natural breaking points. Here's an example:



```
<p>
  In the late 19<sup>th</sup> century, to affirm
  England's religious heritage and to counter the
  effort to disestablish the Church of England, the
  anti<wbr>disestablishmen<wbr>traianism movement arose.
</p>
```



In the late 19th century, to affirm England's religious heritage and to counter the effort to disestablish the Church of England, the antidisestablishmen tarianism movement arose.

FIGURE 2.10 Code fragment with `wbr` and `sup` elements, plus resulting browser window

```
To counter an opponent's connected rooks on a closed file, consider
the Bronstein delay
(www<wbr>.chessgeeks<wbr>.com<wbr>/caughtoncamera<wbr>/bronstein.mp4).
```

Note that the `wbr` elements appear before the slashes. Why is it better to position `wbr` elements before a URL's slashes or dots rather than after? If they are positioned after the slashes or dots and there's a line break after one of the URL slashes or dots, readers might mistake it for the end of the URL.

2.16 `sub`, `sup`, `s`, `mark`, and `small` Elements

Now for some elements that don't quite fit together, but they're close enough, so we can combine them to avoid ridiculously small sections in the book. The `sub`, `sup`, `s`, `mark`, and `small` elements refer to "things," but for each of them, their most striking characteristic is their appearance and not what they are.

The `sup` element is for a superscript. Typically, browsers display `sup` elements with a slightly raised smaller font. For an example, see Figure 2.10, where the "th" in 19^{th} is superscripted.

The `sub` element is for a subscript. Typically, browsers display `sub` elements with a slightly lowered smaller font. Here's an example with 2 subscripted:

Why is water sometimes referred to as H_2O? Because it's made from two hydrogen atoms and one oxygen atom.

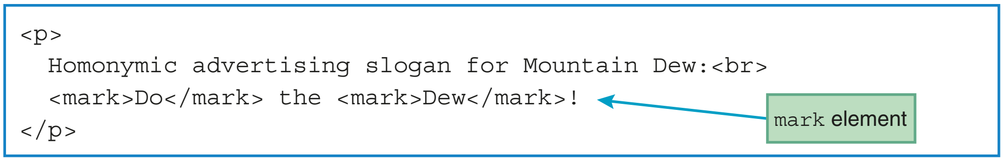
The `s` element indicates something that is no longer accurate or no longer relevant. Typically, browsers display `s` elements with a line-through, like this:

Corporate lobbyists pay huge sums of money to politicians in order to get them elected and gain access after they're elected. ~~But in America, democracy wins by giving equal say to everyday citizens.~~

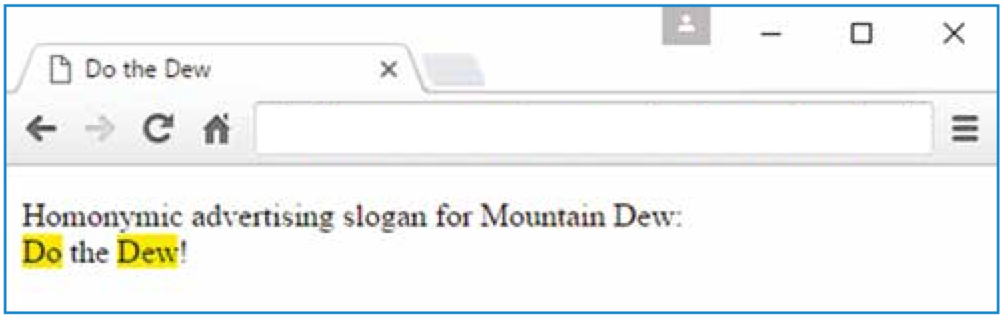
Previously, `s` was used to show stricken text for editing purposes, but the W3C now says `s` is not for editing; it's only for something that is no longer accurate or no longer relevant. Remember the phrasing element that indicates a deletion for editing purposes? The `del` element.

The `mark` element is for text that is marked or highlighted so it can be referred to from another place. Typically, browsers display `mark` elements with a yellow background. For an example, see **FIGURE 2.11**, where "Do" and "dew" are marked with a yellow background so they are easy to identify as homonyms.

The `small` element indicates something that would normally be considered "fine print." It's often used for disclaimers. As you would expect, browsers typically display `small` elements with smaller font than the default. For an example, see the medical disclaimer at the bottom of **FIGURE 2.12**.

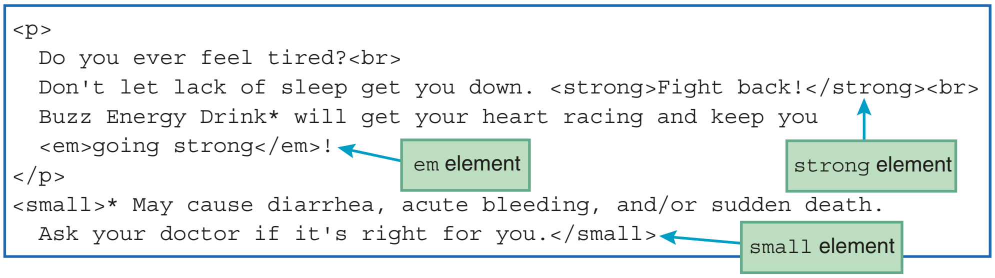


```
<p>
  Homonymic advertising slogan for Mountain Dew:<br>
  <mark>Do</mark> the <mark>Dew</mark>!
</p>
```

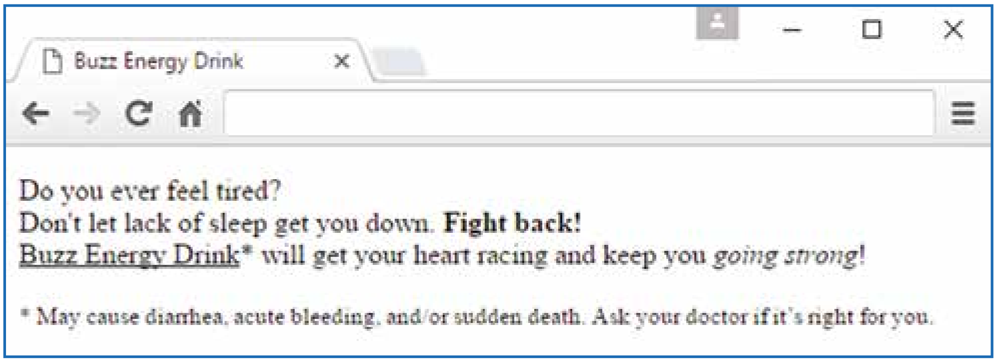


Homonymic advertising slogan for Mountain Dew:
Do the Dew!

FIGURE 2.11 Code fragment with the mark element, plus resulting browser window



```
<p>
  Do you ever feel tired?<br>
  Don't let lack of sleep get you down. <strong>Fight back!</strong><br>
  Buzz Energy Drink* will get your heart racing and keep you
  <em>going strong</em>!
</p>
<small>* May cause diarrhea, acute bleeding, and/or sudden death.
  Ask your doctor if it's right for you.</small>
```



Do you ever feel tired?
Don't let lack of sleep get you down. **Fight back!**
Buzz Energy Drink* will get your heart racing and keep you *going strong*!

* May cause diarrhea, acute bleeding, and/or sudden death. Ask your doctor if it's right for you.

FIGURE 2.12 Code fragment with strong, em, and small elements, plus resulting browser window

2.17 `strong`, `em`, `b`, `u`, and `i` Elements

The W3C makes it clear that elements should be used according to what they represent and not according to their appearance. In the old days—and even today—web developers violated this credo a lot. That is especially true for this section's elements. As in the previous section, this section's set of elements is loosely coupled. What these elements have in common is that web developers very often use them only for their appearance. You should be a force for good and resist that temptation.

The `strong` element is for text that is supposed to be given strong importance. Typically, browsers display `strong` elements with a boldface font. For an example, see "Fight back!" in Figure 2.12. The `em` element is for text that is supposed to be given emphatic stress (the W3C uses the term "emphatic stress," but most people think of "emphasis"). Typically, browsers display `em` elements with italics. For an example, see "going strong" in Figure 2.12.

Historically, the `b`, `u`, and `i` elements have been used for presentation exclusively—b for bold, u for underline, and i for italics. Using HTML elements for presentation is anathema to the guiding principles set forth in the HTML5 standard. For presentation, you're supposed to use CSS. Consequently, the W3C doesn't really like the `b`, `u`, and `i` elements. But the elements have been used so much in the past that the organization feels compelled to support them. They are concerned that if the elements are omitted from the HTML5 standard, (1) browsers will still support them, thus undermining the W3C's credibility, or (2) browsers will no longer support them, thus angering web developers and end users when their legacy code breaks. *Legacy code* is code created in the past that uses commands supported by an older standard and not the current standard.

The W3C describes the `b` element like this:

> The `b` element represents a span of text to which attention is being drawn for utilitarian purposes without conveying any extra importance and with no implication of an alternate voice or mood, such as key words in a document abstract, product names in a review, actionable words in interactive text-driven software, or an article lede.[7]

The W3C appears to have fudged `b`'s definition to try to make it content-oriented rather than presentation-oriented, but it's a stretch. Bottom line: Try to avoid using the `b` element, except for the examples mentioned in the previous definition. Instead, try to use other elements or use CSS.

The W3C describes the `u` element like this:

> The `u` element represents a span of text with an unarticulated, though explicitly rendered, non-textual annotation, such as labeling the text as being a proper name in Chinese text (a Chinese proper name mark), or labeling the text as being misspelt.[8]

---

[7] World Wide Web Consortium (W3C), "HTML 5.1 W3C Recommendation," *W3C.org*, November 1, 2016, https://www.w3.org/TR/2016/REC-html51-20161101/.
[8] Ibid.

As with the b element, the W3C appears to have fudged u's definition to try to make it content-oriented rather than presentation-oriented. Bottom line: Try to avoid using the u element, except when it comports with the preceding definition. Instead, try to use other elements or use CSS.

The W3C describes the i element like this:

> The i element represents a span of text in an alternate voice or mood, or otherwise offset from the normal prose in a manner indicating a different quality of text, such as a taxonomic designation, a technical term, an idiomatic phrase from another language, transliteration, a thought, or a ship name in Western texts.[9]

Same as with the b and u elements, try to avoid using the i element, except for the examples mentioned in the preceding definition. Instead, try to use other elements or use CSS.

2.18 span Element

The span element (like the div block element) has no innate characteristics, either with regard to content or with regard to presentation. Its presentation characteristics are given to it explicitly by CSS. If you want to apply formatting to some text, and the text doesn't coincide with one of the other container elements, then put the text in a span element and apply CSS to the span element.

The next chapter covers CSS in depth, and many of the CSS examples will use the span element. For now, here's a brief overview of the span element.

Think about the u element presented in the previous section. As you know, the u element is frowned upon. For underlining, you should normally avoid using the u element; instead, surround the text that is to be underlined with span tags, and then apply a CSS underline rule to the span element. So if you'd like to underline "Buzz Energy Drink" in Figure 2.12's web page, surround "Buzz Energy Drink" with span tags like this:

```
<span class="underlined">Buzz Energy Drink</span>
```

The class attribute's value (underlined in this case) is the glue that connects an element to a CSS rule. Here's the CSS rule that gets connected to the preceding span element:

```
.underlined {text-decoration: underline;}
```

You can see .underlined, which causes the rule to be connected to the span element. And note text-decoration: underline, which tells the browser to display the span element's text with an underline.

If this example was confusing, don't worry. It'll make more sense when we dig into CSS details in Chapter 3.

[9] Ibid.

2.19 Character References

A *character reference* is code that you can use in your HTML to display a character that would otherwise be difficult to display. Character references are sometimes called "character entities." We use the term "character reference" because that's the term that the W3C uses. See the table in FIGURE 2.13. It contains some of the more popular character references.

Character Reference Syntax

In Figure 2.13, note the odd-looking syntax, `&`*`name`*`;`, for the character references. Each character reference starts with an ampersand, then a name, and then a semicolon. For example, the table's first character reference is `&lt;`, where "lt" is the name. The `&lt;` character reference is for the < symbol, and "lt" stands for "less than." The table shows just a fraction of all the character references defined by the W3C. For a complete list of character references, go to https://www.w3.org/TR/html51/syntax.html#named-character-references. As a sanity check, scroll down to the `&lt;` character reference for the < symbol.

In your HTML source code, you can represent character references in one of two ways. You can use *named character references* or *numeric character references*. For example, you can display the < symbol using the `&lt;` named character reference or the `&#60;` numeric character reference. In your web pages, you should use named character references and not numeric character references. Why? Because numeric character references are more cryptic. For example, isn't `&#60;` harder to figure out than `&lt;`? After you know "lt" stands for "less than," remembering `&lt;` is easy. We won't use numeric character references in our examples going forward. With that in mind, we'll keep things simple and use the shortened term "character reference" when referring to a named character reference.

Character	Character Reference	Description
<	&lt;	less than
>	&gt;	greater than
≤	&le;	less than or equal
½	&frac12;	one-half
¼	&frac14;	one-fourth
&	&amp;	ampersand
"	&quot;	quote
'	&apos;	apostrophe
space	&nbsp;	nonbreaking space
←	&larr; &leftarrow;	left arrow
•	&centerdot;	bullet
✓	&check;	check mark
©	&copy;	copyright

FIGURE 2.13 Character references

In writing the code for a web page, if you want to display a character, normally you just type the character itself. So why are character references necessary? We'll answer that question by examining the character references in Figure 2.13.

Math-Oriented Character References

The table's first character reference is for the < symbol. If you want to display the < symbol for a math-oriented web page, you might attempt to do so by simply typing the "<" character. But browsers will treat that character as the start of an HTML tag, and the "<" character will not be displayed. So to display the "<" character you must use `&lt;`. Similar reasoning explains why you need to use `&gt;` to display the ">" character. If you simply type the ">" character, browsers will treat the ">" character as the end of an HTML tag. The solution is to use `&gt;`, where "gt" stands for "greater than."

Now for the table's third character reference, `&le;`. The "le" in "&le;" stands for "less than or equal," and the `&le;` character reference is for the ≤ symbol. The need for the `&le;` character reference is pretty clear. There is no ≤ key on a standard keyboard.

There are quite a few math-oriented character references. The table in Figure 2.13 shows five of them, for <, >, ≤, ½, and ¼. Referring to the character reference for ≤, you can probably guess the character reference for ≥. That's right, it's `&ge;`. Likewise, referring to the character reference for ½, you can probably guess the character references for other fractions, such as ¼ and ⅔. To verify that your fraction guesses are correct, find the fraction character references on the W3C character reference web page mentioned earlier.

Characters with Special Meaning for HTML

The table's next character reference is for the & symbol. If you want to display it on a web page, you might attempt to do so by simply using the "&" character. But browsers will treat that character as the start of a character reference, and the "&" character will not be displayed. So, to display the "&" character, you must use `&amp;`.

The table's next two character references are `&quot;` and `&apos;`. They can be used to display a quote character (") or an apostrophe character ('), respectively. But normally, if you wish to display a " or ', you should simply use a " or ' and the character will display as is. Can you think of an exception to that rule, when using a " or ' has special meaning and will not display as is? You probably recall that quotes are used in HTML to surround an attribute value. If the attribute value is text that includes a quote mark inside the text, then make sure to use a character reference for the quote mark inside the text. If you use a quote character instead, the browser will treat it as the end of the attribute value's string. For example, this would not work:

This quote mark indicates the end of the `title` value's string.

```
Originally, <abbr title=""Music" Television">MTV</abbr>
actually played music.
```

Here's the corrected code, with `&quot;` character references:

```
Originally, <abbr title="&quot;Music&quot; Television">MTV</abbr>
actually played music.
```

In HTML, you can use single quotes or double quotes to indicate that something is a string. So here's an alternative implementation of the MTV code fragment, with ' characters and `&apos;` character references:

```
Originally, <abbr title='&apos;Music&apos; Television'>MTV</abbr>
actually played music.
```

If you don't like character references because you think they are a form of clutter, you can avoid them by nesting single quotes inside double quotes or double quotes inside single quotes. Note how this example nests single quotes inside double quotes:

```
Originally, <abbr title="'Music' Television">MTV</abbr>
actually played music.
```

Space Characters

Normally, you should use a regular space character to display a blank space. However, there are two cases where you'll need to use a character reference instead of a space character, and we'll discuss those cases in this subsection.

If you want to display a blank space without allowing a line break, you should use the `&nbsp;` character reference, where "nbsp" stands for "nonbreaking space." For example, suppose you're implementing a Motown web page that contains lyrics by the Jackson 5. In the song "ABC," one of the lyrics is "It's easy as 1 2 3." In displaying the lyric, you want 1, 2, and 3 to be on the same line so the reader reads "one two three," as opposed to "one twenty-three" or "twelve three." That means you need to avoid line breaks, and here's the code that does that:

```
It's easy as 1&nbsp;2&nbsp;3.
```

If the 1, 2, or 3 bumps against the browser's right edge, then the entire "1 2 3" wraps to the next line.

Remember that if you have consecutive space characters, browsers will collapse those spaces into one space. If you want to display consecutive spaces and avoid that collapsing, you should use one or more consecutive `&nbsp;` character references. Here's an example that could be used for a pep rally web page:

```
G O&nbsp;&nbsp;J A G U A R S&nbsp; !
```

For the first two-space gap, between "GO" and "JAGUARS," we use two consecutive `&nbsp;` character references. For the second two-space gap, between "JAGUARS" and "!," we use a `&nbsp;` character reference and then a space character. Both techniques would normally display two consecutive spaces. However, the first technique is usually preferred. Why? Because if the exclamation point bumped up against the right margin, the `&nbsp;` space would allow a line break to occur.

Character References When There's No Choice and When There's a Choice

Figure 2.13's last four character references are `&larr;`, `&centerdot;`, `&check;`, and `&copy;`. They are used to display the symbols ←, •, ✓, and ©, respectively. Those character references are just a small sample of character references where there is no key on a standard keyboard for the character reference. With no key, there's no choice. If you want to display the character, you must use the character reference.

On the other hand, if there's a choice, you should use regular characters and not character references. For example, if you want to display an apostrophe for a possessive word, use the apostrophe character (') and do not use `&apos;`. Why? Because the apostrophe character is more readable. All the displayable keys on a standard keyboard have associated numeric character references. Technically, you could use numeric character references for everything. Here's an ugly example: `&#117;&#103;&#108;&#121;`.[10] But proper style dictates that if there's a choice, use regular characters and not named character references or numeric character references.

2.20 Web Page with Character References and Phrasing Elements

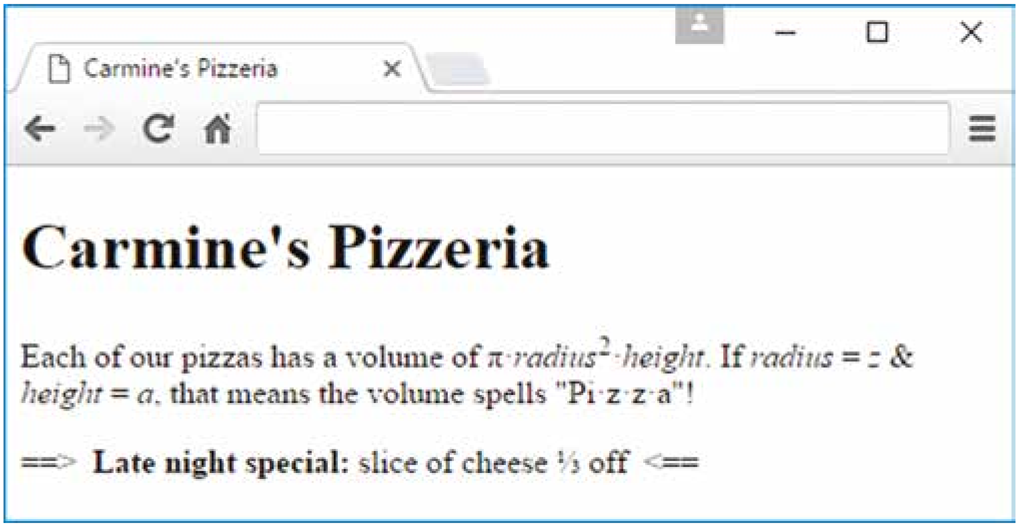
In the past several sections, you learned about phrasing elements and character references. In this section, we put what you learned into practice by examining a complete web page. See the pizzeria web page in **FIGURE 2.14**. Can you figure out what character references and phrasing elements are incorporated into that web page? Try to do that on your own, now, before we analyze the web page together.

Character References

Let's start with the character references. In the pizzeria web page, can you identify the characters that are implemented with character references? What symbols do you see that are not associated with keys on a standard keyboard? The π, •, and ½ symbols are not associated with keys on a standard keyboard, so they have to use character references. Figure 2.13 shows the character references for • and ½. To get the character reference for π, look it up in the W3C character reference table.

By the way, do you understand the purpose of the • symbol? It's for multiplication, so that means Pi • z • z • a is π times the *z* variable squared times the *a* variable.

[10] Can you figure out what word is formed by the given code? Hint: Look up "ASCII characters."



Carmine's Pizzeria

Each of our pizzas has a volume of π·*radius*²·*height*. If *radius* = *z* & *height* = *a*, that means the volume spells "Pi·z·z·a"!

==> **Late night special:** slice of cheese ½ off <==

FIGURE 2.14 **Pizzeria web page**

Next are the < and > symbols, which are used to form `<==` and `==>` arrows. Unlike π, •, and ½, the < and > symbols do have associated keys on the keyboard. However, as you know, if you use a "<" character or a ">" character, they will not display as is. To display the < and > symbols, you need to use the character references shown in Figure 2.13.

Note the web page's & symbol. The & symbol is like the < and > symbols in that it has a special meaning in HTML. To display it, you need to use a character reference.

Can you see anything else in the web page that might need a character reference? Think hard before reading on. This one's not so easy.

Do you see where `&nbsp;` character references might be helpful? To avoid a line break within "radius = z" and within "height = a," you should use `&nbsp;` character references around the equals signs. See Figure 2.14's browser window. With the browser window's width, if regular spaces were used (and not `&nbsp;` character references), the browser would have split "height = a." Not a disaster, but somewhat awkward. Another place for `&nbsp;` character references is the bottom line. Do you see the two spaces at the right of `==>` and also at the left of `<==`? If regular space characters had been used, then whitespace collapsing would have occurred. But `&nbsp;` character references are used and whitespace collapsing is avoided.

FIGURE 2.15 shows the source code for the Pizzeria web page. The code highlighted in red is for all the character references. As a sanity check, read through the web page's character reference code and make sure it makes sense.

Before we move on to the phrasing elements, there's one more thing to consider in regard to character references. In Figure 2.14, do you see the quote marks around "Pi • z • z • a"? You have a choice of implementing them with either " characters or with character references. Remember, whenever you have a choice, you should use the more straightforward option, which means " characters in this case. You can see the " characters in Figure 2.15.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="author" content="John Dean">
<title>Carmine's Pizzeria</title>
</head>

<body>
<h1>Carmine's Pizzeria</h1>
<p>
  Each of our pizzas has a volume of
  &pi;&centerdot;<var>radius</var><sup>2</sup>&centerdot;<var>height</var>.
  If <var>radius</var>&nbsp;=&nbsp;<var>z</var> &amp;
  <var>height</var>&nbsp;=&nbsp;<var>a</var>, that means the volume spells
  "Pi&centerdot;z&centerdot;Z&centerdot;a"!
</p>
<p>
  ==&gt;&nbsp; <strong>Late night special:</strong> slice of cheese
  &frac13; off&nbsp; &lt;==
</p>
</body>
</html>
```

Character references are in red.

Phrasing elements are in green.

FIGURE 2.15 Source code for Pizzeria web page

Phrasing Elements

Now go back to Figure 2.14 and see if you can figure out what phrasing elements are used in the pizzeria web page. The web page refers to variables named *radius*, *height*, *z*, and *a*. Because they are variables, you should identify them as such by using a separate `var` container for each variable. Note the following line, copied from the pizzeria web page's code, which has a `var` container for each of the two variables, *radius* and *height*:

```
&pi;&centerdot;<var>radius</var><sup>2</sup>&centerdot;<var>height</var>
```

In Figure 2.14, you can see how the `var` containers affect the variables' appearance. The browser displays the variables with italics. Using italics for `var` elements is typical for all the major browsers.

The pizzeria web page displays this formula:

$$\pi \bullet \mathit{radius}^2 \bullet \mathit{height}$$

Because the 2 is a superscript, you should identify it as such by using a `sup` container. That's what the pizzeria web page does, and you can see the relevant code fragment several lines up.

Here's the last noteworthy item in the pizzeria web page. In the web page, do you see the boldfaced "Late night special:"? We want that label to be forceful, so we surround it with a `strong` container, like this:

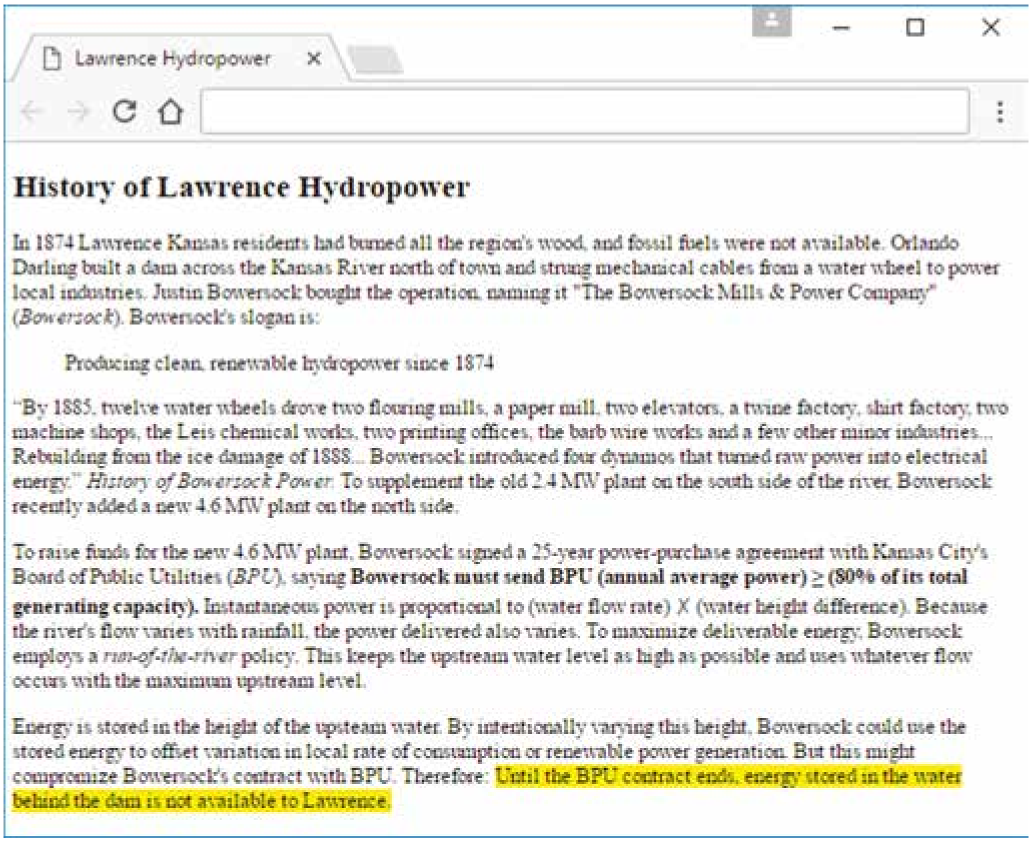
```
<strong>Late night special:</strong>
```

2.21 CASE STUDY: A Local Hydroelectric Power Plant

This section adds another web page to our case study website. The web page describes hydropower produced at a small dam across the Kansas River at Lawrence, Kansas. **FIGURE 2.16** shows the desired result.

Let's partition the task of creating this page. The first five lines of HTML source code should be the same as they were in the previous web page. The `meta description` element should identify this chapter rather than the previous one, and the title element should provide an identifier that describes this page's particular content. That completes the `head` element.

The `body` begins with a `header` whose size is the same as the `header` of the previous web page. As you may recall, that was an `h2` element. Then comes the first text paragraph. Nothing special seems to be happening until we get to the italicized term in parentheses, (*Bowersock*). This is a defined item, whose definition is the preceding quoted string. Now look at the HTML source code in **FIGURE 2.17A**. What puts *Bowersock* in italics is the enclosing `dfn` element (not a deprecated `i` element).



History of Lawrence Hydropower

In 1874 Lawrence Kansas residents had burned all the region's wood, and fossil fuels were not available. Orlando Darling built a dam across the Kansas River north of town and strung mechanical cables from a water wheel to power local industries. Justin Bowersock bought the operation, naming it "The Bowersock Mills & Power Company" (*Bowersock*). Bowersock's slogan is:

> Producing clean, renewable hydropower since 1874

"By 1885, twelve water wheels drove two flouring mills, a paper mill, two elevators, a twine factory, shirt factory, two machine shops, the Leis chemical works, two printing offices, the barb wire works and a few other minor industries... Rebuilding from the ice damage of 1888... Bowersock introduced four dynamos that turned raw power into electrical energy." *History of Bowersock Power.* To supplement the old 2.4 MW plant on the south side of the river, Bowersock recently added a new 4.6 MW plant on the north side.

To raise funds for the new 4.6 MW plant, Bowersock signed a 25-year power-purchase agreement with Kansas City's Board of Public Utilities (*BPU*), saying **Bowersock must send BPU (annual average power) ≥ (80% of its total generating capacity).** Instantaneous power is proportional to (water flow rate) *X* (water height difference). Because the river's flow varies with rainfall, the power delivered also varies. To maximize deliverable energy, Bowersock employs a *run-of-the-river* policy. This keeps the upstream water level as high as possible and uses whatever flow occurs with the maximum upstream level.

Energy is stored in the height of the upsteam water. By intentionally varying this height, Bowersock could use the stored energy to offset variation in local rate of consumption or renewable power generation. But this might compromize Bowersock's contract with BPU. Therefore: Until the BPU contract ends, energy stored in the water behind the dam is not available to Lawrence.

FIGURE 2.16 Lawrence Hydropower web page

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="author"content=John Dean">
<title>Lawrence Hydropower</title>
</head>

<body>
<h2>History of Lawrence Hydropower</h2>
<p>
  In 1874 Lawrence Kansas residents had burned all the region's wood,and
  fossil fuels were not available. Orlando Darling built a dam across the
  Kansas River north of town and strung mechanical cables from a water
  wheel to power local industries. Justin Bowersock bought the operation,
  naming it "The Bowersock Mills & Power Company" (<dfn>Bowersock<dfn>).
  Bowersock's slogan is:
</p>
<blockquote cite="http://www.bowersockpower.com">
  Producing clean, renewable hydropower since 1874
</blockquote>
```

FIGURE 2.17A Source code for Lawrence Hydropower web page

Back in Figure 2.16, notice that Bowersock's slogan stands out because it has space around it. An easy way to get this effect is to use `blockquote` element. Since it's just a short string and the preceding colon indicates it's logically a continuation of the preceding paragraph, it's tempting to put it before that paragraph's closing `</p>`. But if you try that, the W3C Markup Validator will complain with the rather cryptic error message "No `p` element in scope but a `p` end tag seen." The problem is that because both `p` and `blockquote` are block elements, neither can fit inside the other. So the `blockquote` element should go after the first paragraph's closing `<\p>`.

In Figure 2.17A, also notice that the `blockquote` includes something that does not appear in the browser's presentation—a `cite` attribute. This hidden documentation tells future website maintainers where this (and other related) information might be found.

Back in Figure 2.16, the second paragraph begins with a long quotation, followed by an italicized title. What's going on here? The paragraph at the top of **FIGURE 2.17B** shows it's just an ordinary quotation followed by a citation identifying the source of that quotation. But this paragraph also has other features. In the HTML source code, the first instance of the abbreviation, MW, is contained in an `abbr` element, which includes a `title` attribute whose value is "megawatt." In some browsers, there is no initial indication of this documentation, but if you hover the mouse over this particular "MW" abbreviation, after a while a box will pop up and display the "megawatt" expansion of this abbreviation. This paragraph also contains another hidden feature: Enclosing the general term "recently" in a `time` element, whose `datetime` attribute specifies "recently" as May 10, 2013, documents the exact date of start of construction of the new plant.

In Figure 2.16's third paragraph, in the middle of the second line, notice the italicized term, *BPU*. Like the first paragraph's italicized *Bowersock*, this italicized *BPU* indicates a defined item, whose definition is the preceding string, "Board of Public Utilities." Shortly after the *BPU* definition, notice the bold-faced clause, **Bowersock ... capacity**). Figure 2.17B shows this to be a

```
<p>
  <q>By 1885, twelve water wheels drove two flouring mills, a paper mill,
  two elevators, a twine factory, shirt factory, two machine shops, the
  Leis chemical works, two printing offices, the barb wire works and a
  few other minor industries... Rebuilding from the ice damage of 1888...
  Bowersock introduced four dynamos that turned raw power into electrical
  energy.</q> <cite>History of Bowersock Power.</cite> To supplement the
  old 2.4 <abbr title="megawatt">MW</abbr> plant on the south side of the
  river, Bowersock <time datetime="2013-05-10">recently</time> added a
  new 4.6 MW plant on the north side.
</p>
<p>
  To raise funds for the new 4.6 MW plant, Bowersock signed a 25-year
  power-purchase agreement with Kansas City's Board of Public Utilities
  (<dfn>BPU</dfn>), saying <strong>Bowersock must send BPU (annual
  average power) &ge; (80% of its total generating capacity).</strong>
  Instantaneous power is proportional to (water flow rate) &cross;
  (water height difference). Because the river's flow varies with
  rainfall, the power delivered also varies. To maximize deliverable
  energy, Bowersock employs a <em>run-of-the-river</em> policy. This
  keeps the upstream water level as high as possible and uses whatever
  flow occurs with the maximum upstream level.
</p>
<p>
  Eenrgy is stored in the height of the upsteam water. By intentionally
  varying this height, Bowersock could use the stored energy to offset
  variation in local rate of consumption or renewable power generation.
  But this might compromize Bowersock's contract with BPU. Therefore:
  <mark>Until the BPU contract ends, energy stored in the water
  behind the dam is not available to Lawernce.</mark>
</p>
</body>
</html>
```

FIGURE 2.17B Source code for Lawrence Hydropower web page

`strong` element, and this `strong` element contains the embedded `&ge;` character reference for the ≥ symbol. The informal multiplication symbol on the next line is a `&cross;` character. Alternative multiplication symbols are 'X', 'x', '*', and the `&centerdot;` character. Near the end of Figure 2.16's third paragraph, notice the italicized term, *run-of-the-river*. Figure 2.17B shows that this emphasis comes from the `em` element, not the deprecated `i` element.

In Figure 2.16's final paragraph, notice the highlighting of the last sentence. This indicates a caveat—a caution to remember while reading other material. Figure 2.17B's last text rows show how a `mark` element flags this warning.

Review Questions

2.2 HTML Coding Conventions

1. Web browsers are strict in terms of forcing developers to write good code. True or false.
2. Who developed the original HTML Tidy tool? Name one person.

2.3 Comments

3. What is a browser engine?
4. What is documentation?

2.5 Content Model Categories

5. Using the Content model categories diagram, specify five categories that are completely inside the flow category.

2.6 Block Elements

6. Provide two characteristics of a block element.

2.7 `blockquote` Element

7. What are the typical default display properties for a `blockquote` element?

2.8 Whitespace Collapsing

8. What are three characters that are subject to whitespace collapsing?

2.11 Editing Elements

9. In the world of web programming, what does "presentation" mean?

2.12 `q` and `cite` Elements

10. What is the difference between a `q` element and a `blockquote` element?

2.14 Code-Related Elements

11. Times New Roman is an appropriate font for displaying code because it enables you to align similar types of things within your program. True or false.

2.15 `br` and `wbr` Elements

12. What does "wbr" stand for?

2.19 Character References

13. When you want to display a quote character (") on a web page, you should normally use the `&quot;` character reference. True or false.

Exercises

1. [after Section 2.2] Why do companies like their programmers to follow standard coding conventions?
2. [after Section 2.2] What does Google's Style Guide have to say about trailing whitespace?
3. [after Section 2.3] Convert the following code so that it is compliant with HTML5 standards and also proper coding conventions. To ensure compliance with the HTML5 standards, I recommend that you enter your converted code into the W3C's HTML validation service. To ensure proper coding conventions, please carefully review the HTML5 coding conventions document.

```
<html>
<head>
<meta charset="utf-8">
<meta name="author" content="John Dean">
<title>Mock Trial How-To<title>
</head>

<body>
<H1>Mock Trial Opening Statements</H1>
<hr>
<strong><p>Prosecuting Attorney</strong>:<br>
Good morning, I am the prosecuting attorney, and I represent the
State. I will call three witnesses. At the conclusion of the case,
we will ask you to convict the defendant of the crime as charged,
thank you.
</p>
<p><strong>Defense Attorney</strong>:<br>
Ladies and Gentlemen of the jury, I intend to prove that my
client xxxxxx is innocent of the alleged murder of yyyyyy, and the
evidence presented by the prosecution is circumstantial.
</body>
```

4. [after Section 2.3] Suppose that your company requires you to include this copyright notice at the top of every one of your web pages:

INVESTMENT INTELLIGENCE SYSTEMS CORP.
THIS MATERIAL IS COPYRIGHTED AS AN UNPUBLISHED WORK UNDER SECTIONS 104 AND 408 OF TITLE 17 OF THE UNITED STATES CODE. UNAUTHORIZED USE, COPYING, OR OTHER REPRODUCTION IS PROHIBITED BY LAW.

Show an HTML5 comment container that includes this copyright notice. As always, use proper coding conventions. Note that the copyright notice is a comment and, as such, it should not display on your web pages.

5. [after Section 2.5] This question gives you practice using the HTML5 language specification website to determine the permitted contents of elements. In your answers, specify one or more of the content model categories (such as “phrasing” or “flow”), or specify “empty.”

a) What are the permitted contents of the `blockquote` element?

b) What are the permitted contents of the `br` element?

c) What are the permitted contents of the `q` element?

6. [after Section 2.13] Provide an HTML5 code fragment for a paragraph element that displays this message:

Attention Walmart shoppers:
Christmas sales begin September 15 at 5 am, just in time for the holidays.

You must provide code that enables JavaScript to understand the date and time. There is no need to provide the JavaScript code itself. For the date value, use the current year.

7. [after Section 2.13] Provide an HTML5 code fragment for a paragraph element that describes a solid-state device (SSD). Your paragraph must include at least two sentences, and in those sentences, you must include the acronym SSD. You must use an element that indicates that SSD is an acronym and another element that indicates that SSD is a term that is being defined. In your paragraph, you must include a definition for SSD. You must provide code that generates a tooltip for the words that SSD stands for.

8. [after Section 2.17] Using the `b`, `u`, and `i` elements is generally frowned upon. So why does the W3C include them in their HTML5 standard?

9. [after Section 2.19] Provide a paragraph element that would render the following line. Use the browser’s default font face (there is no need to specify a font). There are two spaces between the two sentences. Display both spaces.

The ampersand symbol is “&.” The greater than or equal symbol is “≥.”

10. [after Section 2.19] Provide an HTML code fragment that would render the quadratic equation as follows:

$$x = (-b \pm (b^{2} - 4ac)^{\frac{1}{2}}) / 2a$$

Note:

- You don’t have to provide a style container with your answer, but you should assume that the following style container appears at the top of your quadratic equation web page. It causes the entire web page to use monospace font.

```
<style>
  body {font-family: monospace;}
</style>
```

- To further the goal of describing your web page's content, you must surround each variable (*x*, *a*, *b*, and *c*) individually with proper tags.
- Insert single spaces on each side of the equals sign, at the left of the ± symbol, on each side of the minus sign, and on each side of the / sign. Do not insert spaces elsewhere.
- You don't need to worry about the equation being too long to fit on one line and exhibiting line wrap. That means you don't need to use `&nbsp;` character references.
- In the preceding equation, note how ½ is one character, with the 1 on top of the /, not 1 at the left of the /.

Project

Create a web page with a filename of `pgmExplanation.html` that explains something interesting about a program. Your explanation must make sense. You must use grammatically correct sentences that provide a reasonable flow through your web page. Creativity and aesthetics are part of web programming, and they are part of this assignment. Follow these guidelines:

- Include at least one heading (`h1`, `h2`, etc.) in your web page.
- Display a code fragment from the program, or display the whole program, if appropriate. Feel free to create the program yourself or use a program that you find in a book or on the Internet.
- Refer to a specific variable in the program, specific input, and specific output.
- Provide a quote from a book that refers to the program or to a concept illustrated by the program, and provide a citation for the book's title. Feel free to use a real quote from a real book or make up a fictional quote from a fictional book.
- Provide an acronym (real or made up) or a definition that somehow relates to the web page's discussion.
- Include a total of at least 10 different types of phrasing elements in your web page. You must use appropriate phrasing elements that fit the flow of your web page. You will lose points if any of your elements are inappropriate (e.g., using `wbr` in a normal-length word).

For future projects, you will use CSS for formatting. But for practice purposes, for this project, use only HTML5 elements and not CSS.