CHAPTER OUTLINE

2.1 Introduction
2.2 HTML Coding Conventions
2.3 Comments
2.4 HTML Elements Should Describe Web Page Content Accurately
2.5 Content Model Categories
2.6 Block Elements
2.7 blockquote Element
2.8 Whitespace Collapsing
2.9 pre Element
2.10 Phrasing Elements
2.11 Editing Elements
2.12 q and cite Elements
2.13 dfn, abbr, and time Elements
2.14 Code-Related Elements
2.15 br and wbr Elements
2.16 sup, sub, s, mark, and small Elements
2.17 strong, em, b, u, and i Elements
2.18 span Element
2.19 Character References
2.20 Web Page with Character References and Phrasing Elements
2.21 Case Study: A Local Hydroelectric Power Plant

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

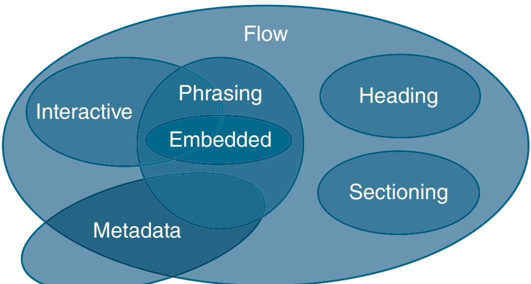


FIGURE 2.1 Content model categories

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.

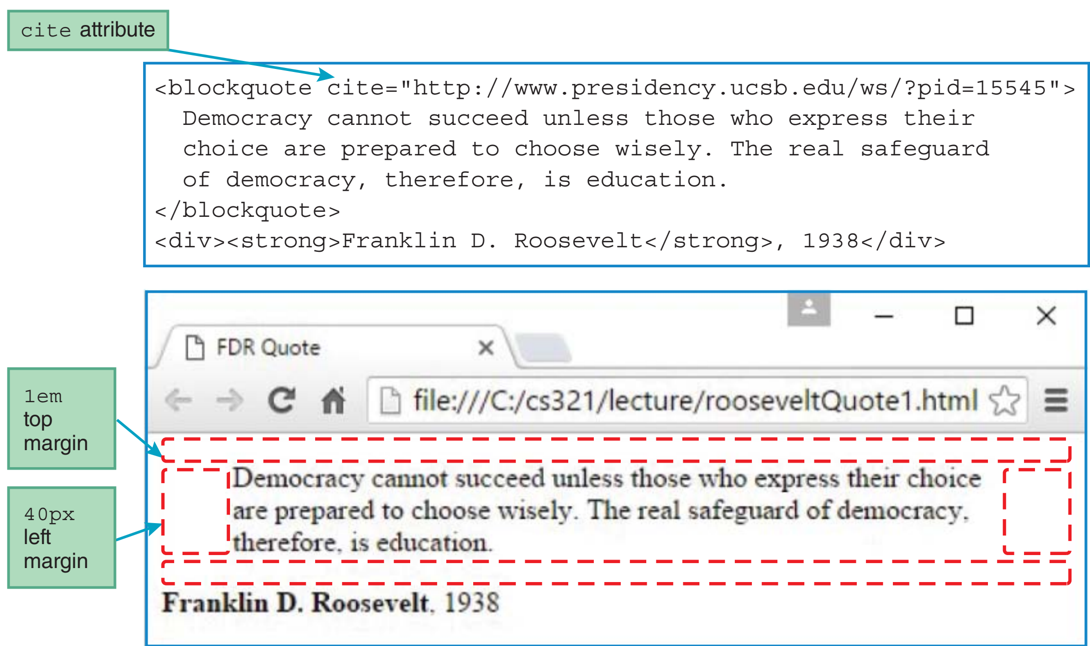


FIGURE 2.2 An example blockquote

[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 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:

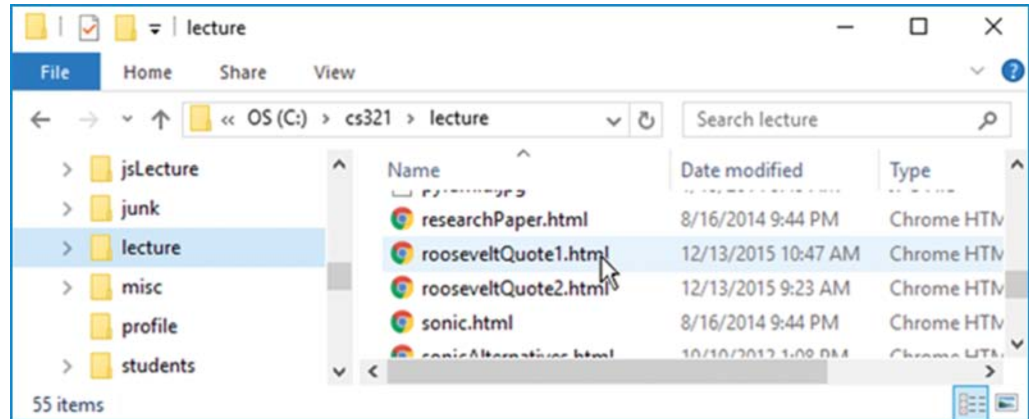
```
<div><strong>Franklin D. Roosevelt</strong>, 1938</div>
```

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.



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

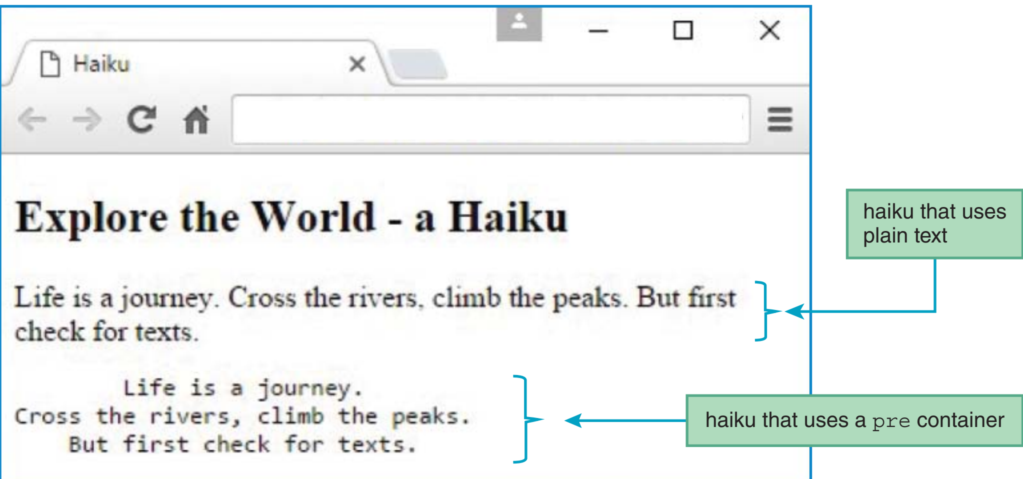


FIGURE 2.5 Haiku web page

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.