CHAPTER OUTLINE

6.1 Introduction

This chapter presents two web page features—links and images—that are emblematic of what it means to be a web page. Almost all web pages include links and images, and they are usually crucial to the web page's popularity. And if you're a web page, popularity is everything.[1]

In Chapter 4, you learned a few link syntax details while implementing a `nav` container. In this chapter, you'll learn more link syntax details, plus various techniques for jumping to different link targets. In particular, you'll learn how to jump to a target location on a different web page, as well as to a target location on the current web page. Next, you'll learn how to download a file. For all the jumping and downloading operations, you can use a path to identify the location of your target, and you'll learn different techniques for specifying paths. You'll also learn techniques for formatting the links, using various CSS rules.

After learning about links, next you'll learn about another basic building block—images. You can implement a link to an image, but in this chapter, we focus on standalone images. In Chapter 4, you learned a few image syntax details while implementing a `figure` container. In this chapter, you'll learn a few more image syntax details, but most of the image discussion in this chapter will be about the different types of images that are available. In particular, you'll learn about bitmap image file formats (GIF, JPEG, and PNG) and a vector graphics file format (SVG).

6.2 a Element

To implement a link, you'll need to use the `a` element. Here's an example `a` element that implements a link to Park University's website:

```
<a href="http://www.park.edu">Park University</a>
```

[1] This rather shallow notion of success is particularly prevalent with younger web pages, where getting invited to the "popular" web pages' parties is paramount.

The text that appears between an a element's start tag and end tag forms the link label that the user sees and clicks on. So in this code, the link label is "Park University." By default, browsers display link labels with underlines. So this code renders like this:

Park University

The blue color indicates that the linked-to page has not been visited. We'll discuss visited and unvisited links later on.

When the user clicks on a link, the browser loads the resource specified by the href attribute. For this example, the "resource" is another web page, so when the user clicks on the "Park University" link, the browser loads that web page—the one at http://www.park.edu. As an alternative to specifying a web page for the href attribute's resource, you can specify an image file for the href attribute's resource. We'll show an example of that in the next chapter.

Besides enabling a user to load a resource (which usually means jumping to another web page), the a element can be used as a mechanism that enables a user to download a file of any type—image file, video file, PDF file, Microsoft Word file, and so on. To implement that download functionality, include a download attribute as shown here:

```
<a download href="http://www.park.edu/catalogs/catalog2018-2019.pdf">
  Park University 2018-2019 catalog</a>
```

As always for the a element, the browser displays the text that appears between the start tag and end tag; in this case, that's "Park University 2018–2019 catalog." When the user clicks on that text, the browser downloads the file specified by the href attribute. The user can then choose to view it or save it. Normally, the download attribute has no value, but if you (as the web programmer) want the end user to save the file using a different filename than that specified by the href atttibute, then you should include a filename for the download attribute's value, such as download="Park University 2018-2019 catalog.pdf". But be aware that the browser might override your download attribute's filename and use the href attribute's filename instead.

Continuation Rule for Elements that Span Multiple Lines

Did you notice that the preceding a element code fragment spans two lines and the second line is indented? This subsection explains that indentation. It's a style thing, and the explanation is not specific to the a element. This subsection is a digression from the rest of this chapter, and it's relevant for examples throughout the rest of the book.

The a element is a phrasing element (which means that it displays "inline," like a phrase within a paragraph). Most phrasing element code is short and can easily fit on one line, but the preceding a element is rather long and spans two lines. The solution was to press enter at the end of the a element's start tag and indent the next line. We use that same solution for other elements that are too long to fit on one line. Here's a meta element example:

```
<meta name="description"
  content="This web page presents Dean family highlights.">
```

Note that we press enter after the `name` attribute-value pair. In the `a` element example, we pressed enter after the `a` element's start tag. The goal is to press enter at a reasonable breaking point. For the preceding example, if we wait to press enter until after "Dean family," that would not be a "reasonable breaking point." It would split the `content` attribute's value across two lines. Doing so would make the code slightly harder to understand and thus defeat one of the primary purposes behind good style—understandability.

Back in Chapter 2, we introduced you to block formatting for block elements, which means the start and end tags go on lines by themselves. The `p` element is a block element, and here's a properly formatted `p` element:

```
<p>
  Known for its Computer Science program,
  <a href="http://www.park.edu/informationAndComputerScience/accolades">
  Park University</a> is tied for first in the number of Turing Award
  winners among all Missouri universities whose motto is "Fides es Labor."
</p>
```

Press enter.

Align the `a` element's continuation line with the prior line.

Within the `p` container, there's an `a` element that spans more than one line, and we align its continuation line with the line above it (we don't indent further). That's different from the prior `a` element example. Indenting continuation lines is a bit of a gray area. If the continuation line is in a container that represents something that is supposed to look like a paragraph (e.g., `p` or `blockquote`), then do not indent. Otherwise indent.

Note the line break in the source code after "program,". We inserted a line break so we could fit the `a` element's entire start tag on one line. If we attempt to put the `a` element's entire start tag on the same line as "program," then *line wrap* would occur if we printed the code on paper. Here's what that would look like:

```
<p>
  Known for its Computer Science program, <a
href="http://www.park.edu/informationAndComputerScience/accolades">
  Park University</a> is tied for first in the number of Turing Award
  winners among all Missouri universities whose motto is "Fides et Labor."
</p>
```

line wrap

Note how `href` wraps to the next line where it's not indented. The `href` is aligned with the `<p>` start tag, and that implies that `href` is separate from the `p` element rather than a part of it. And that implication makes the logic hard to follow. The moral of the story is, for statements that might be too long to fit onto one line, press enter at an appropriate breaking point to avoid line wrap.

Types of `href` Attribute Values	Where the Link Jumps To
absolute URL	Find the resource on a different web server than the current web page.
relative URL	Find the resource on the same web server as the current web page. Specify the location of the resource by providing a path from the current web page's directory to the destination web page.
jump within current web page	Find the resource within the current web page. Specify the location of the resource by providing an `id` value for an element in the web page.

FIGURE 6.1 `href` attribute values

Different Types of `href` Values

As you know, the `a` element's `href` attribute value specifies the resource that is to be loaded. In addition to specifying the resource, the `href` attribute value indicates where the link jumps to in order to find the resource. Indicating where the link jumps to is not a trivial task. Take a look at FIGURE 6.1, which provides an overview of the different link-jumping techniques employed by the `href` attribute's value.

For an example that uses an *absolute URL*, suppose you want to add a link on a Facebook page that directs the user to an Instagram page. Here's the link code to do that for a subscriber named Hannah Davis:

```
<a href=" https://www.instagram.com/hannahDavis.html">
  Hannah's Instagram</a>
```

Note the https value for the `href` attribute. You've seen http in the past; https is another popular protocol that you can use with the `href` attribute. It stands for hypertext transfer protocol secure. So the https protocol provides more security for communications than does http.

To jump to a web page that resides on the same web server as the current web page, for the link's `href` attribute, use a relative URL. The relative URL specifies a path from the current web page's directory to the destination web page. The next section provides complete details and examples.

To jump to a designated location within the current web page, for the link's `href` attribute, use a value starting with # such that that value matches an `id` attribute's value for an element in the web page. We introduced this technique earlier, and we'll provide more details and examples later in this chapter.

6.3 Relative URLs

As promised, in this section we provide additional details about relative URLs. A relative URL value allows you to jump to a web page that resides on the same web server as the current web page. It does so by specifying a path from the current directory to the destination web page. The

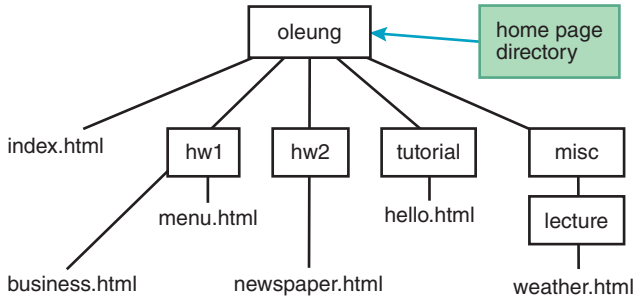


FIGURE 6.2 Example directory tree

current directory is the directory where the current web page resides. The destination web page is the page that the user jumps to after clicking on the link.

In forming a path for a relative URL value, you'll need to understand how files and directories are organized in a *directory tree* structure. Note the example directory tree in **FIGURE 6.2**. It shows the container relationships between all the files and directories that are within the `oleung` directory. The `oleung` directory is the home page directory for student Olivia Leung. A *home page* is the default first page a user sees when the user visits a website. We'll have more to say about home pages shortly, but for now, just realize that a *home page directory* is the directory where a home page resides, and a home page is the starting point for a user's browsing experience on a particular website. A *subdirectory* (also called a subfolder or a child folder) is a directory that is contained within another directory. Thus, in Figure 6.2, there are four subdirectories within the `oleung` home page directory and one subdirectory within the `misc` subdirectory. The other entities in Figure 6.2 (the words without borders) are files. The `index.html` file is in the `oleung` directory, and the other files are in the tree's subdirectories. Because `oleung` is the home page directory, you might have guessed that `index.html` is the home page file. That is indeed the case.

In forming a path for a relative URL value, you'll need to navigate between a starting directory and a target file. In doing so, you'll need to follow these rules:

- Use /'s to separate directories and files.
- Use ". ." to go from a directory to the directory's parent directory.

In the second bullet, what does *parent directory* mean? In Figure 6.2, `oleung` is the parent directory of `hw1` because `oleung` and `hw1` are connected by a single line with `oleung` on the top. That should make sense because it parallels how human parents are displayed in a genealogy tree. The point of all this is that if you're in a directory and you want to go to that directory's parent directory, you need to use `..` For example, suppose you want to provide a link on the newspaper page that takes a user to the home page, `index.html`. Because the newspaper page is in the `hw2` directory, the `hw2` directory is considered to be the current directory. The home page is in the `oleung` directory. The `oleung` directory is the parent directory of `hw2`, so you need to use `..` in order to navigate up to the `oleung` directory. Here's the relevant `a` element code:

```
<a href="../index.html">Olivia's Home Page</a>
```

Note the / (forward slash) between `..` and `index.html`. As explained earlier, the / is a delimiter that separates directories and files. The `..` refers to a directory and index.html is a file, so the / is

necessary to separate them. Does it make sense that `..` refers to a directory? Remember that `..` navigates from the current directory (`hw2` in this example) to its parent directory (`oleung` in this example), so the result is indeed a directory.

Relative Path Examples

Using Figure 6.2's directory tree, can you try to come up with the code for an `a` element that resides in the `index.html` file and takes the user to the business page? See if you can do that on your own without glancing down at the answer.

Assuming you've tried to work it out on your own, now you may look at the answer:

```
<a href="hw1/business.html">Business Page</a>
```

Note that `href`'s value starts with `hw1`. In coming up with the answer on your own, did you start with `..` instead? It's a common error among beginning web programmers to feel the need to use `..` to go up from the current web page to that web page's directory. Try to avoid that misconception. There's no need to go up. If you're in a web page, then you're already in that web page's directory. So to go from the `index.html` page to the business page, you simply go down from `oleung` to `hw1` by specifying `hw1`, then down to the business page by specifying `/business.html`.

The `index.html` page is the website's home page, and as such, it's the first page that you look at when you visit a website. To help with a user's viewing experience, home pages should normally contain links to other pages on the website. So for your next challenge, you should implement another link from the home page—this time, have the link go to the weather page. Try to come up with the code on your own. Assuming you've made an honest attempt, now you can look at the answer:

```
<a href="misc/lecture/weather.html">Weather Page</a>
```

Because the link resides on the home page, the path originates from the home page directory, `oleung`. To get to the weather page, you have to go down to the `misc` directory and then down to the `lecture` directory. In the example code, notice the `/`'s separating the two directories and the `weather.html` filename.

For one last relative URL challenge, try to come up with the code for an `a` element that resides in the business page and takes the user to the menu page. Once again, see if you can do that on your own without glancing down at the answer. Here's the answer:

```
<a href="menu.html">Menu Page</a>
```

Note that there is no directory in the `href` value—only the filename by itself. With the business and menu pages both in the same `hw1` directory, there's no need to change directories and therefore no need for a path in front of the filename.

Path-Absolute URLs

As stated previously, a relative URL is for jumping to a web page when the current web page and the target web page are on the same web server. In the prior examples, the relative URL's path started at the current web page's directory. As an alternative, you can have the relative URL's path start at the web server's root directory. A web server's *root directory* is the directory on the web

server that contains all the directories and files that are considered to be part of the web server. [2] If you want to have the relative URL's path start at the web server's root directory, preface the URL value with `/`. For example, using Figure 6.2's directory tree, if `oleung` is immediately below the web server's root directory, the following code could be added to any of the web pages shown, and it would implement a link to the `index.html` page:

```
<a href="/oleung/index.html">Olivia's Website</a>
```

A URL value that starts with a `/` is referred to as a *path-absolute URL*, and it's a special type of relative URL (it's considered a relative URL because the path is relative to the current web server's root directory). The term path-absolute URL comes from the WHATWG. Remember the WHATWG? It stands for Web Hypertext Application Technology Working Group. It's the organization that keeps track of the HTML5 standard with a living document. Its standard aligns very closely with the W3C's HTML5 standard, but because it's a living document, the WHATWG is free to update things at any time it sees fit. Unfortunately, the W3C does not have a formal term for a path that starts with /. But rest assured that the path-absolute URL syntax is valid—all major browsers have supported it for many years. By the way, if you see the term *root-relative path*, that's just another way to refer to a path-absolute URL. In the real world, both terms are popular.

6.4 `index.html` File

If a user specifies a URL address that ends with a directory name, then the web server will automatically look for a file named `index.html` or `index.htm` and attempt to load it into a browser. This occurs when you specify a URL for a link's `href` value and also when you enter a URL in a web browser's address box.[3]

The default searched-for file can be reconfigured by the web server's administrator. It's common for Microsoft IIS web server administrators to use `default.htm` as another default filename for displaying a web page when the URL address ends with a directory name. According to the coding-style conventions in Appendix A, we recommend that you use `index.html` for your home page file names because that name is the most standard. We prefer `.html` to `.htm` because `.html` is more descriptive—after all, "html" describes the code contained in the file.

If a user specifies a URL address that ends with a directory name, and the web server cannot find a file named `index.html` or `index.htm` (or possibly `default.htm`) in the specified

---

[2] As you might recall, the term "web server" can refer to (1) the physical computer that stores the web page files or (2) the program that runs on the computer that enables clients to retrieve the web pages. When talking about a "web server's root directory," we're not referring to the computer's hard drive root directory. We're referring to the directory that the web server program designates as the top-level container directory for all the web-related files that the web server computer uses.

[3] During the development process, you'll probably want to implement your web pages on your local computer's hard drive. You can load most of your hard drive web pages to a browser by simply double clicking on the file within Microsoft's File Explorer tool. But if you double click on a directory name, the index.html file won't load by default. Why? Because the web server is the thing that knows to look for the `index.html` file (i.e., it knows to search for it if it sees a directory name). And you bypass the web server if you double click a web page file within File Explorer.

directory, then the web server will either (1) load a web page that shows the contents of the specified directory, or (2) display an error page (e.g., "Directory Listing Denied" or "HTTP 404 - Page Not Found"). To avoid the directory contents web page (which is rather ugly) or the error page, you should include an `index.html` file in every directory that might be specified as part of a URL. It might seem a bit weird to have multiple `index.html` files within your website, but for larger websites, get used to it. Clearly, you need an `index.html` file in the directory that sits at the top of your website's directory tree structure—that particular `index.html` file is your website's home page. The other `index.html` files are not the official home page, but you can think of them as "home pages" for different areas within your website.

6.5 Web Design

Because a home page is the default first page a user sees when the user visits a website, it's the web developer's first—and possibly only—opportunity to make a good impression on users. So try to get it right, or your users might leave and never come back. In this section, you will learn a few tips on how to make a good first impression. These tips are part of a software area known as *web design*, which is comprised of these subareas: user interface design, user experience design, graphic design, and search engine optimization. Sorry, explaining all that is beyond the scope of this book. But we do explain some of it—we scratch the surface on user interface design and user experience design. That's a surface that needs to be scratched, so let's begin.

User Interface Design

In a general sense, *user interface design* (UID) refers to the mechanisms by which users of a product can use the product. For web pages, the mechanisms are things like text, color, pictures, buttons, text boxes, and progress bars. A good UID designer will anticipate users' needs and create an interface that meets those needs by incorporating components that are easy to understand and use.

After the home page downloads, users will want to identify the web page's main content quickly. To help in that regard, you should try to avoid clutter, and focus on clear, concise words (and graphics, if appropriate) that describe the web page's main content. Remember that for a nontrivial website, the home page is only for the main content and links to other web pages, not for lots of details. If you need to provide lots of details for something, you should put those details on a separate web page and link to that page from the home page. The link labels themselves are important. For each link label, use only one word or a few words that get to the point quickly. Don't be afraid to remove unnecessary text and to have whitespace on your home page. Whitespace can provide a nice respite for stressed-out web surfers.

In presenting the web page's content, it's important to be consistent with your text and colors. For text, you should limit the number of text fonts used. Pick pleasing fonts that go together well for your main content and your subsidiary content. Make sure that the foreground text colors contrast with the background colors so the text is easy to read. Normally, that means the contrasting colors should be different in terms of lightness and darkness. You should pick a set of colors for your text, background, and graphics that complement each other. Apply a consistent strategy for choosing which colors go to which type of content.

User Experience Design

User experience design is a bit more nuanced than user interface design. For a car, its user interface might be the power steering, the heated leather seats, and the wireless Internet communication. The user experience might be a feeling of calm comfortable control. For a web page, the UID incorporates the elements described in the previous subsection, whereas the user experience design is the feeling produced by those UID elements. For example, you should choose colors and fonts that generate the proper feeling for the user's browsing experience. The user experience design's feeling comes not only from UID elements, but also from things that enhance the user's ability to digest the web page's content, such as helpful pictures, familiar controls, fast downloads, and efficient navigation between web pages.

The first thing a web user notices when visiting a web page is how long it takes to load the page. So in designing your home page, pay heed to the size of the web page file and all its resource files. In particular, image files and video files tend to be large. Later on, you'll learn how to load such resource files without slowing the download for the web page's other content. But for a positive user experience, you should still try to avoid having your home page built with large files.

Most home pages will have links to other web pages within the website. You should use the `nav` container to group those links. Normally, users navigate to other web pages by clicking links at the top and left, so that's where your `nav` containers should go. Suppose your website has lots of web pages. On the home page, you could include a link to every one of those web pages, but that might lead to a cluttered home page. As part of the web design process, you need to determine how many links to put on your home page and how to navigate to other pages after clicking on those links. Let's look at some different strategies.

See **FIGURE 6.3**, which shows the organization for a website whose pages are organized with a *linear structure*. That means the home page links to one other page, and that second page links to one other page, and so on. That type of strategy might be used for a website whose purpose is to present a long article. By splitting up the article into separate pages and connecting them with links, the user doesn't have to scroll so far down while reading. Other examples where a linear structure is beneficial are a shopping cart that steps through the transaction or a tutorial with a specific sequence of steps. But be aware that pure linear structures are not all that common. Why? It's difficult for users to go backwards. To return to the beginning, the user has to click the back button multiple times. Another reason linear structures have fallen out of favor is because people are now used to scrolling feverishly on their phones. Using the scroll bar to scroll through a lengthy web page doesn't seem all that bad when compared to lots of link clicking and back button clicking.

Now take a look at **FIGURE 6.4**, which shows a website whose pages are organized with a *hierarchical structure*. That means the home page links to several other pages where those pages serve as pseudo-home pages for the different areas within the website. When compared to linear

FIGURE 6.3 Website with a linear structure for its web pages

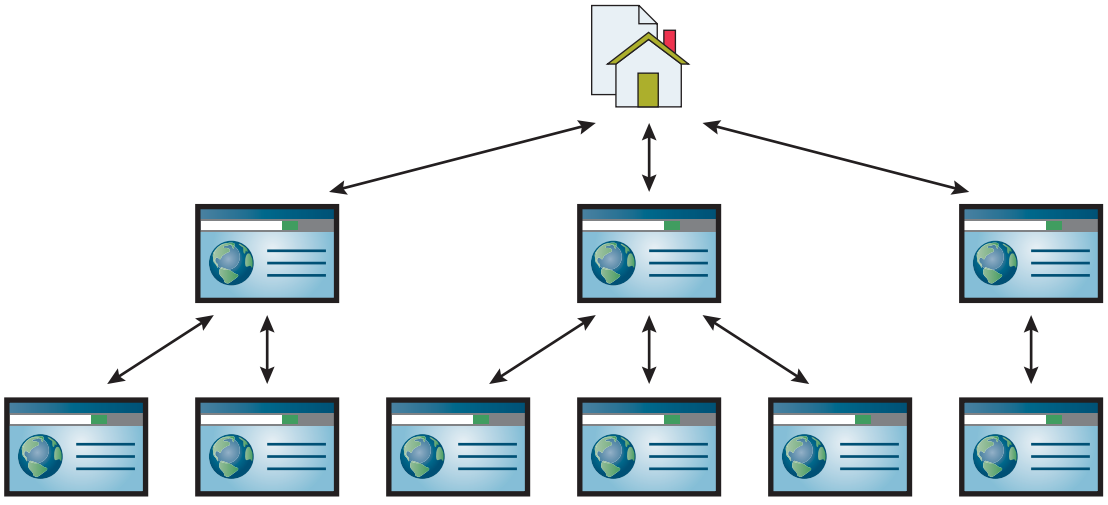


FIGURE 6.4 Website with a hierarchical structure for its web pages

structure websites, hierarchical structure websites tend to reduce the number of clicks needed to navigate through a website's web pages.

With hierarchical structures, the home page is often called the *top-level page*, and the next-level pages are often called *second-level pages*. Second-level pages have links to the other pages in their areas. Both top-level and second-level pages are sometimes referred to as landing pages because they can be targets (or "landing places") of links that reside outside the website.

Finally, take a look at **FIGURE 6.5**, which shows a website whose pages are organized with a mixed structure. There's a hierarchical structure for compartmentalizing the website's main areas, plus additional links that (1) connect from within the areas to other areas and (2) connect pages

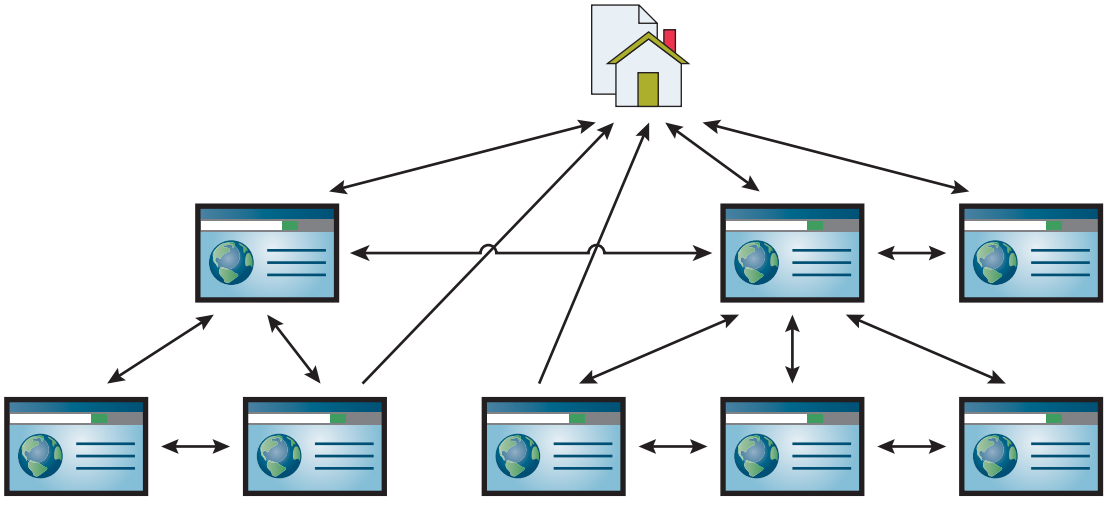


FIGURE 6.5 Website with a mixed structure for its web pages

back to the website's home page. For nontrivial websites, this sort of mixed structure is by far the most common type of structure because it leads to the best user experience—fewer clicks to drill down to the different pages within a given area (due to the hierarchical structure) and fewer clicks to jump from within one area to get to a different area. The ability to go from one area to another area is particularly useful when there's an area that serves as a repository for information needed by multiple other areas. To accommodate that scenario, just add a link from each of those areas to the common repository's second-level page.

6.6 Navigation Within a Web Page

You might recall from Figure 6.1 that there are three basic types of values for an `a` element's `href` attribute. We've already talked about an absolute URL value and a relative URL value. In both of those cases, the target is a web page separate from the current web page. The third type of `href` value shown in Figure 6.1 is for when you want a link that takes the user to some specified point within the current web page. That can be particularly useful for long web pages, so the user can quickly jump to designated destinations within the web page. For example, note the blue links near the top of the Clock Tower web page in **FIGURE 6.6**. If the user clicks the Clock Tower Photograph link, then the web page scrolls within the browser window so that the link's target (the clock tower photograph itself) gets positioned at the top of the browser window. In Figure 6.6, there is no scroll bar, so no scrolling takes place if the link is clicked. But don't think that this situation (no scrolling) is normal. If a web page has internal links, then the web page will normally be sufficiently long so as to justify the internal links. We'll show the code used to implement those internal links, but first, let's describe the web page's other links.

Note the Back to the Top link at the bottom of the page. With a long web page, it's common to have such a link so the user can quickly get back to the top after scrolling to the bottom. Once again, because there is no scroll bar in Figure 6.6's browser window, no scrolling takes place if the link is clicked.

Note the yellow sidebar at the left of the Clock Tower web page. It forms the navigation area with links to other web pages. For example, clicking on Zombie Bloodbath takes users to a web page that describes Park University's ill-fated foray into the brain-eating movie production business back in 1978.

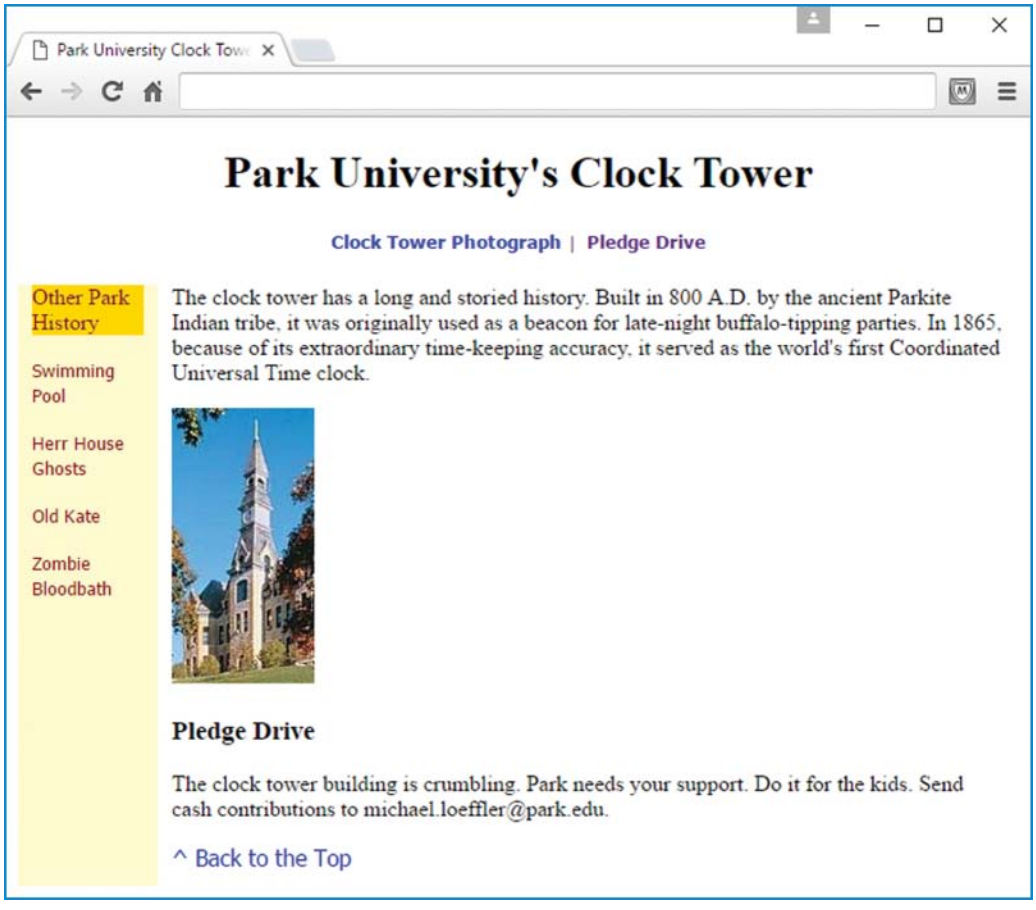
Syntax for Internal Link

Now it's time to show the code used to implement the internal links in the Clock Tower web page. To jump to a designated location within the current web page, you need to use a value starting with `#` such that that value matches an `id` attribute's value for an element in the web page. Here's the Clock Tower web page's code that links to the pledge drive section:

```
<a href="#pledge-drive">Pledge Drive</a>
```

And here's the element that that link jumps to:

```
<h3 id="pledge-drive">Pledge Drive</h3>
```



Picture permissions granted by Brad Biles, Park University Director of Communications and Public Relations.

FIGURE 6.6 Clock Tower web page

Note the spelling for `pledge-drive`. Standard coding conventions suggest using hyphens to separate multiple words in an `id` value. That should look familiar because you've already learned to use hyphens with multiple-word `class` attribute values.

For a given web page, you can have only one element with a particular `id` value. So because `pledge-drive` appears in this `h3` statement, `pledge-drive` cannot be used as an `id` value anywhere else within the Clock Tower web page. That should make sense—after all, if another element used that same `id` value, then the browser engine would be confused as to which target element to link to when the user clicks the `a` element (with `href="#pledge-drive"`) shown. If you accidentally use the same `id` value for more than one element, don't worry, the validation service will flag it as an error.[4]

Look for the `a` element code and the `h3` element code in **FIGURE 6.7A**'s Clock Tower web page source code. Also, look for two other internal links in the source code. Specifically, can you

[4] Of course, this works only if you use the W3C's validation service. You've been using the validation service for all of your web pages, right?

```
<body>
<header>
  <h1 id="top">Park University's Clock Tower</h1>
  <h5>
    <a href="#tower-photo">Clock Tower Photograph</a>  |  
    <a href="#pledge-drive">Pledge Drive</a>
  </h5>
</header>

<div class="table">
  <nav class="cell">
    <div class="nav-heading">Other Park History</div>
    <div><a>Swimming Pool</a></div>
    <div><a>Herr House Ghosts</a></div>
    <div><a>Old Kate</a></div>
    <div><a>Zombie Bloodbath</a></div>
  </nav>

  <section class="cell">
    <p>
      The clock tower has a long and storied history.
      Built in 800 A.D. by the ancient Parkite Indian tribe,
      it was originally used as a beacon for late-night
      buffalo-tipping parties.
      In 1865, because of its extraordinary time-keeping accuracy,
      it served as the world's first Coordinated Universal Time clock.
    </p>
    <p>
      <img id="tower-photo" src="../images/clockTower.jpg"
      alt="Clock Tower Photograph">
    </p>
    <h3 id="pledge-drive">Pledge Drive</h3>
    <p>
      The clock tower building is crumbling.
      Park needs your support. Do it for the kids.
      Send cash contributions to michael.loeffler@park.edu.
    </p>
    <a href="#top">^ Back to the Top</a>
  </section>
</div>
</body>
</html>
```

It's common to use a vertical bar to separate internal links.

Note the character references to ensure two spaces on each side of the vertical bar.

FIGURE 6.7A Source code `body` container for Clock Tower web page

find the link that jumps down to the `tower-photo img` element? And how about the link at the bottom that jumps up to the top of the page?

Walking Through the Clock Tower Web Page's Source Code

There are quite a few details in the Clock Tower web page that are worth looking at. They're not specific to the main point of this section (internal links within a web page), but they are noteworthy nonetheless. The Clock Tower web page's code is long, requiring two figures to show it all. Figure 6.7A shows the web page's `body` container, and **FIGURE 6.7B** shows the `head` container. It might seem odd to have the first figure display the `body` code and the second figure display the `head` code, even though the clockTower.html file positions the `head` container above the `body` container (of course). In the book's figures, we position the two containers in reverse order because when we examine the web page, it'll make more sense to look at the `body` container before the `head` container.

Let's start by examining the `body` container code that separates the `tower-photo` and `pledge-drive` links:

```
  |  
```

Using a vertical bar (|) to separate internal links is a very common technique. The vertical bar has no impact on the web page's functionality; it's just for appearance purposes. It's a way to indicate a separation between internal links. Using common techniques helps users to feel comfortable, and that in turn might encourage users to buy what the web page is selling.

Also in Figure 6.7A, note the ` ` character references and the blank spaces surrounding the vertical bar. By inserting a ` ` character reference next to a space character at the left of the vertical bar and also at the right of the vertical bar, that causes two spaces to display on each side of the vertical bar, which looks nicer than single spaces.

Below the vertical-bar-separated links, you can see a `nav` container and also a `section` container, both with `class="cell"`. Can you figure out the purpose of `class="cell"`? We're using a simple CSS table for layout with only one row and two cells—one cell for the navigation bar at the left and one cell for the web page's main content. As you probably already know, a *navigation bar* is a group of links that enable users to navigate (link to) the various pages on a website. To make the user feel comfortable, the navigation bar should look the same on each web page. Typically, navigation bars go at the left, but top and bottom are common as well.

In the Clock Tower `nav` container, note the `a` elements and the fact that they have no `href` attributes. An `a` element that has no `href` attribute is called a *placeholder link*. Although the `nav` container's placeholder links currently go nowhere, the idea is that they will be replaced later on with active links. For now, each link is "holding a place" for a future link.

Now let's focus on the web page's table layout. As you can see in Figure 6.7A, the `nav` and `section` elements both contain `class="cell"` attribute-value pairs. As you'll confirm later when we examine the CSS rules, that causes the `nav` and `section` elements to act like table cells. What element acts like the surrounding table? The `div` element, with `class="table"`.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="author" content="John Dean">
<title>Park University Clock Tower</title>
<style>
  header {text-align: center;}
  .table {display: table;}
  .cell {
    display: table-cell;
    padding: 0px 10px 10px;
  }
  nav.cell {width: 80px;}

  /* Avoid yellow background at left extending above the
    page's main content. */
  .cell > :nth-child(1) {margin-top: 0;}
  nav {
    color: darkred;
    background-color: lemonchiffon;
  }
  .nav-heading {background-color: gold;}

  /* Adjust link appearances. */
  a {
    text-decoration: none;
    font-family: Tahoma, Geneva, sans-serif;
  }
  a:hover {text-decoration: underline;}
  nav a {font-size: .8em;}
  nav > * {margin: 1em 0;}
</style>
</head>
```

The `.table` and `.cell` rules form a table that holds the navigation bar at the left and the main content at the right.

Fixed width for the navigation area at the left.

`:nth-child()` selector

FIGURE 6.7B Source code `head` container for Clock Tower web page

Clock Tower Web Page's CSS Rules

Now let's work our way through some of the more interesting details in the web page's CSS rules in the `head` section. Note the `.table` and `.cell` selectors in Figure 6.7B. As you learned in Chapter 5, they use the `display` property with `table` and `cell` values to implement a table and the cells of a table, respectively. But note that there is no `.row` selector. For a browser to render a table, the cells need to be inside rows, so how can there be no `.row` selector that implements the rows? Here's the deal: if you have `table-cell` elements that are not surrounded by a `table-row` element (or even a `table` element), then the browser engine generates an anonymous (hidden) `table-row` element around those `table-cell` elements. For the Clock Tower web page, there are indeed `table-cell` elements that are not surrounded by a `table-row` element. Can you find where the anonymous `table-row` element tags are inserted in the web page body code? Here's the relevant code from Figure 6.7A:

```
<div class="table">
  <nav class="cell">
  ...
  <section class="cell">
  ...
</div>
```

Anonymous `table-row` element start tag goes here.

Anonymous `table-row` element end tag goes here.

Note the callouts that show where the anonymous table-row element tags get inserted in the code.

Now let's turn to the web page's CSS rules. Here's the CSS rule for the navigation area's cell:

```
nav.cell {width: 80px;}
```

Remember what the dot means in the `nav.cell` selector? It means that the browser engine locates all `nav` elements and then matches only those `nav` elements that use `cell` as a `class` attribute value. If you look at the web page body's source code, you'll see that `cell` appears as a `class` attribute value in two places—in the `nav` container (which implements the navigation bar at the left) and also in the `section` container (which holds the page's main content). The `nav.cell` selector matches only the cell in the navigation area at the left. And after matching that navigation area cell, it specifies a fixed width for it. Why do you need a fixed width for the navigation area, but not for the main content area? If the user expands or shrinks the window size, the user would normally expect the paragraphs at the right to expand and shrink, not the navigation area at the left. And that's what happens for the Clock Tower web page.

In the following CSS rule, we already talked about the `table-cell` value for the `display` property. Now let's talk about its `padding` property:

```
.cell {
  display: table-cell;
  padding: 0px 10px 10px;
}
```

The rule applies to both table cells (the navigation bar cell and the main content cell). It specifies that there's no padding on the top, 10 pixels of padding on the left and right (from the second value), and 10 pixels at the bottom. Why is it appropriate to have no padding on the top? As you can see in the `nav` CSS rule, the left cell has a yellow background and the right cell has no background. If there was padding at the top of the navigation bar's cell, then it would show up as a yellow area jutting above the navigation bar's text. Because the main content area has no background color, there would be no comparable color jutting above the main content area's text, and that asymmetry would look weird. So that's why it's appropriate to have no padding on the top of the navigation bar cell. Knowing to specify 0 pixels for the top of the navigation bar cell is not all that intuitive. Coming up with that solution took some tweaking. Get used to such tweaking when trying to make your web pages look good.

The issue previously described (avoiding a yellow area jutting above the navigation bar's text) actually requires even more tweaking than just setting the top padding to 0 pixels. It also requires setting no margin above each of the top elements in the navigation bar cell and the main content area cell. Here's the relevant CSS rule from the Clock Tower web page's code:

```
.cell > :nth-child(1) {margin-top: 0;}
```

As you know, this rule is known as a child selector, where child selectors use the `>` symbol to match elements that are child elements of other elements. In processing this rule, the browser engine searches for child elements whose parents use `cell` for their `class` attribute value. The `:nth-child()` selector is a special child selector in that it allows you to specify which child element is selected. This particular rule selects immediate children of "cell" elements where the child is a first child (the `1` value is for the first child).

The `:nth-child()` selector thing is a *pseudo-class*. In Chapter 5, you learned that a pseudo-class begins with a colon and it conditionally selects elements from a group of elements specified by the selector at the immediate left of the colon. For the `:nth-child` example, there is no type selector at the immediate left of the colon. So, why's that? If you don't specify a type selector, then the browser inserts the `*` universal selector as the implicit type selector. So behind the scenes, that rule converts to this rule:

```
.cell > *:nth-child(1) {margin-top: 0;}
```

As you may recall, the `*` universal selector matches all elements, so all elements are matched (that are children of elements with a `class` attribute value of `cell`) and the browser then applies the `:nth-child(1)` pseudo-class to those matches.

By the way, as an alternative to using `:nth-child(1)`, you can use the `:first-child` pseudo-class, which also selects the first child from among a list of child elements. We'll use the `:first-child` pseudo-class in a later chapter.

There are still a few Clock Tower web page CSS rules that we have not yet talked about, but before we describe them, let's take a short break from rules and talk about everyone's favorite topic—style. You'll need to view the CSS code in Figure 6.7B to appreciate the following style commentary, so be prepared to jump back and forth.

The `nth-child` rule is kind of tricky, so you should include a comment for it. The comment is too long to fit at the right of the `nth-child` rule, so the comment appears above the rule. Note the indentation on the comment's line continuation.

Because there are so many CSS rules for the Clock Tower web page, you should separate the rule groups with blank lines. For each rule group that is nonintuitive, you should preface the group with a comment above the rule group. Note the rule group comments in the Clock Tower web page's source code.

In the Clock Tower web page, the last four CSS rules deal with links (see the `a` element in each rule). We'll wait until the next section to cover those rules, because CSS for links is a big enough subject that it deserves its own section.

6.7 CSS for Links

Have you ever noticed that after clicking on a link and returning later, the link's color is different? For example, note the different colored links at the top of the Clock Tower web page in Figure 6.6. The left link, labeled Clock Tower Photograph, is blue, indicating that the link has not been clicked. On the other hand, the right link, labeled Clock Tower Photograph, is purple, indicating that the link has been clicked in the past. By default, the major browsers use blue text for unclicked links and purple text for clicked links. More formally, those links are referred to as *unvisited links* and *visited links*, respectively. The HTML5 standard does not mention blue and purple as typical defaults, but those colors have been used for decades, so it's reasonable to assume they'll remain as defaults for many more years down the road.

A link is defined as a "visited link" if it leads to a location that the computer's browser has been to recently. Browsers have different time limits to determine whether a location has been visited "recently." If you clear your browser's history, the browser will consider all links to be unvisited, so they will go back to their unvisited color. That can be useful for testing a link's color during development because once you load a page, it no longer displays the unvisited link color.

Be aware that end users have the ability to override the link colors specified by the browser by adjusting their browser's settings. But also be aware that as a developer, you have even more power than the user in this regard. You can use CSS to override the link colors specified by the browser, and those CSS rules override the user's link color browser settings as well.

Now for the CSS that enables you to specify link colors. For unvisited links, use this syntax:

```
a:link {color: color-value;}
```

The `a` is the element type for a link element. The `:link` thing is a pseudo-class. It qualifies the `a` element type by searching only for links that have not been visited. As you'd expect, the `a:link` selector is known as a *pseudo-class selector*.

You now know to use `a:link` for unvisited links. For visited links, use `a:visited`, like this:

```
a:visited {color: color-value;}
```

Here are a couple of examples. The first rule specifies burlywood for unvisited links, and the second rule specifies light blue for visited links:

```
a:link {color: burlywood;}
a:visited {color: #aaaaff;}
```

Despite the undeniable excitement of being able to change web page link colors with CSS, please try to show restraint. Using different colors might confuse your end users. They might not consciously realize that a web page's link colors are different, but they might realize it at a subconscious level. That can be particularly annoying for users who explicitly configure their browser's link colors.

In addition to being able to change the colors in your web page's links, you can also change your links' underline scheme. By default, browsers display links with underlines. If that leads to visual clutter or confusion with regular text that's underlined, and you'd like to have no link underlining, then use `text-decoration: none`, like this:

```
a {text-decoration: none;}
```

If link underlines are disabled using this CSS rule, but you want to display underlines when the mouse hovers over a link, use the `a:hover` pseudo-class selector with `text-decoration: underline`, like this:

```
a:hover {text-decoration: underline;}
```

In case you were wondering, the `:hover` selector matches any element that is being hovered over, not just links, so to limit the matches to just links, you need to preface `:hover` with `a`, as shown in the example.

When using both of these rules, there's a conflict in that they both apply to link elements. How is that conflict resolved? Because the `a:hover` selector is more specific than the a selector, the `a:hover` selector overrides the `a` selector for links that are being hovered over.

We encourage you to find these CSS rules in the Clock Tower web page code in Figure 6.7B. While you're there, also note the two `nav` element CSS rules, which are copied here for your convenience:

```
nav a {font-size: .8em;}
nav > * {margin: 1em 0;}
```

The first rule specifies a smallish font size of `.8em` for all link elements in the `nav` container. The second rule adds a `1em` margin to the top and bottom of all elements that are child elements of the `nav` container. We hope those rules make sense—you want the navigation bar links to be unobtrusive (with a smaller font size), but to make them easily clickable, you want them separated vertically (with added margin space at the top and bottom of each link).

As with all the book's web page examples, we encourage you to retrieve the Clock Tower web page's source code from the book's website and play around with it. After loading the page in a browser, try hovering your mouse over the top links. That should cause your standard mouse

pointer to change to a hand icon, indicating an active link. Then hover your mouse over the placeholder links. That should cause your mouse pointer to change to an I-bar icon. That's a bit confusing, so to improve the user's experience, you could add this CSS rule:

```
nav a {cursor: not-allowed;}
```

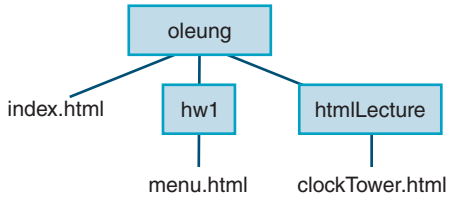
With that rule in place, hovering your mouse over the placeholder links causes your mouse pointer to change to a blocked icon (), indicating that the link is inactive.

6.8 a Element Additional Details

There are just a couple more things to discuss about links that don't quite fit with the earlier sections. Previously, we discussed how to link to a different web page and also how to link to a specified location within the current web page. It's also legal to combine those techniques and link to a specified location within a different web page.

Linking to a Specified Location Within a Different Web Page

To link to a specified location within a different web page, use the `href` attribute to specify the other page (using an absolute url with http or a relative URL) and then append a `#` value to specify the location within that page. To explain that more fully, we need an example. Given the following directory tree, can you provide a link from the home page to the clock tower photograph? The photograph's `img` element resides in the `clockTower.html` web page, and it uses an `id` value of `tower-photo`. Try to come up with the code for the link element before you glance down at the answer.



As shown, student Olivia Leung has an `index.html` home page in her `oleung` home page directory. In implementing a link from the home page to the clock tower photograph, don't fall into the trap of thinking you need to use `..` to first go up to the `oleung` home directory. Remember: when you're in the `index.html` home page, the "current directory" is the `oleung` directory. So the path to the clock tower photograph starts by going down (from the `oleung` directory) to the `htmlLecture` directory. Here's the link code:

```
<a href="htmlLecture/clockTower.html#tower-photo">
  Park University Clock Tower Photograph</a>
```

Values for the a Element's target Attribute	Description
_self	Overlay the current web page with the target web page.
_blank	Open the target web page in a new browser window or in a new tab within the current browser window.
_parent	Open the target web page in the current web page's parent document, which is typically the browser window that caused the current web page to open.

FIGURE 6.8 Specifying the environment in which the web page opens

After the target web page's filename (clockTower.html), you can see #tower-photo. Go back to Figure 6.7A and find id="tower-photo" in the clock tower photograph's img element. Remember: to link to a particular element, you preface the element's id value with #.

target Attribute

So far, we've discussed only one attribute for the a element—the href attribute. Now let's discuss the target attribute. The a element's target attribute value specifies where to open the linked-to web page. Note the three values listed in FIGURE 6.8.

The target attribute's default value is _self. As noted in the figure, if you have a link that uses the _self value and the link is clicked, the specified new page loads within the current web browser and overlays the previous page. Most links omit the target attribute and stick with the _self default behavior. But every now and then, you might want to open a targeted web page in a new browser window or in a new tab, and we address that next.

As noted in the figure, if you have a link that uses the _blank value and the link is clicked, the specified new page loads in a new browser window or in a new tab within the current browser window. With the _blank value, it's up to the browser to decide whether to use a new window or a new tab. Here's an example link element that uses the _blank value:

```
<a href="https://www.youtube.com/watch?v=zm48WoRs0hA&noredirect=1"
  target="_blank">Bethany Mota: Perfect Back to School Hair, Makeup
  &amp; Outfit!</a>
```

Here's a coding conventions refresher: In the code, note that target is indented. That's because it's a continuation of the a start tag that began on the previous line. Note, & Outfit is also indented, to the same column as the second line. Both lines are a continuation of the a element, so they both get indented one level.

So with a target="_blank" link, if the link is clicked, the specified new page loads in a new browser window or in a new tab within the current browser window. Can you think of a disadvantage of that behavior? Clicking the back arrow won't take the user back to the previous page. Maybe that's OK if you want to give prominence to a particularly important linked-to

page (like Bethany Mota's classic 2013 video "Perfect Back to School Hair, Makeup & Outfit!"[5]), but if you want to counteract that behavior, the `target` attribute's `_parent` value can come to your rescue.

As noted in Figure 6.8, if you have a link that uses the `_parent` value and the link is clicked, the target web page loads in the current web page's *parent document*, which is typically the browser window that caused the current web page to open. Effectively, that means if you use `target="_blank"` to open a web page in to a new window or new tab, you can use `target="_parent"` to open another web page in the original window or tab. If you return to the original web page with `_parent`, you'll often want to use JavaScript to close the newly opened window or tab after you leave it. We'll discuss JavaScript extensively later in the book.

[5] According to my middle school and high school daughters, vlogger Bethany Mota is the coolest, barely twentysomething in the whole world. Formerly bullied, she's now a *motavatour* megastar.