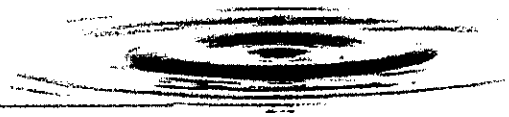
# Front-End Developer V2.0 Handouts (5-13-19)

# TABLE OF CONTENTS – V2.0

**Excerpt from "Spring into HTML and CSS" by Molly E. Holzschlag (Addison-Wesley Professional)**



CHAPTER 1

# Building an HTML Page

Web documents are meant to be constructed logically. You must have certain elements in place for your document to conform and validate. Conformance means that the document actually follows the language and language version in which it is being written. Validation is the technical process by which we test conformance, allowing us to find errors and fix mistakes.

The first thing you can do to make absolutely certain that your pages are given the best fighting chance at conformance is to begin with all the required and structural elements that are needed *before* you begin to add text and other content.

Ironically, it's only been in hindsight that the majority of people working on websites have improved upon the way they use markup. The Web was in such a state of evolutionary, rapid growth that new elements and features were being added to browsers—and HTML—all the time. Many of these features made it into the actual specifications, but many did not. What's more, elements of HTML pages that should have been included from the beginning have often been left out, even by professionals.

How is this possible, you might wonder? Well, the primary piece of software used to interpret HTML is the desktop web browser. These browsers have a long history of forgiving errors. Of course, they also have a long history of introducing errors! Browsers have been both the blessing and the curse of the Web because they have allowed for innovation but often spent more time adding fun features rather than basic support for the languages they are meant to support. As a result, the Web is a mishmash of HTML use—most of it not conforming or valid—and, in light of this chapter's discussion, many times authored without the basic structural components required by the language.

A movement is afoot to bring better standards to browsers, to the tools that people use to develop websites, and to those of us interested in creating pages that not only work, but work well, regardless of whether our goals are personal or professional.

In this chapter, you learn to create a template that will serve as the foundation for everything you do in this book. This template will contain all the necessary and helpful technical and structural bits that form the basis of a document that will conform and validate, too.

# Declaring and Identifying the Document

The first thing you'll want to do in your page is add a bit of code that declares which type of document you're using and identifies the language version. This is done with Standardized General Markup Language (SGML), which is the parent language to HTML and appears in this important declaration, known as the *DOCTYPE declaration.* This declaration is a unique piece of code, and a suitable declaration must be used in every document you create.

Example 1-1 shows the DOCTYPE declaration we'll be using in all examples in this book:

**EXAMPLE 1-1** The DOCTYPE declaration for XHTML 1.0 Transitional

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

Look a little weird? Not to worry. I'll go through it with you so you have a firm understanding of what each bit of this declaration means. First, there's the opening <!, which many readers who have looked at HTML before might wonder about. The angle bracket is a familiar component in HTML, but the exclamation mark appears in only one other situation with HTML: in comments, which you'll also learn about in this chapter. This symbol isn't used too often because it's SGML syntax being used in the context of HTML. Here, it simply means that a declaration is about to begin. This is then followed by the term `DOCTYPE`, which states that this code is declaring the *document type.*

The next bit is `html`, which defines this document type as being written in HTML. Note that it's in lower case here. This is significant because we're using XHTML—and because XHTML is case sensitive, this particular part of the declaration must be in lower case. If it's not, your document will not validate. The word `PUBLIC` is an important piece of information. This means that the particular document type being referenced is a public document. Many companies create unique versions of XHTML, with customized elements and attributes. For our purposes, the public version of HTML that we're going to use is absolutely sufficient.

The ensuing syntax `"-//W3C//DTD XHTML 1.0 Transitional//EN"` defines the host of the document's language type and version (The World Wide Web Consortium, W3C), and states that the document is being written according to the XHTML 1.0 Transitional Document Type Definition (DTD). A DTD is simply a long laundry list of allowed elements and attributes for that language and language version. Finally, there's a complete URL that goes to the DTD, `"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"`. If you were to load this into your browser, you'd see the actual DTD, for XHTML 1.0 Transitional (see Figure 1-1).

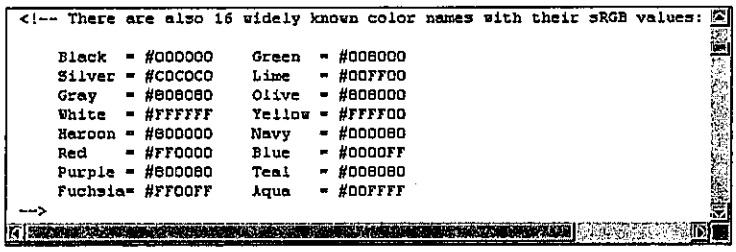



**FIGURE 1-1** A portion of the XHTML Transitional DTD.

With this declaration at the top of your document, you will be able to author the document and then run it through a validator to check for conformance. The validator uses the information in the declaration and compares your document to the DTD you've declared. If you've followed the rules allowed in the DTD you've declared here, you should have no errors whatsoever, which is, of course, our goal.

**QUANTUM LEAP**

Due to discrepancies in the way that many browsers were handling various aspects of HTML and CSS, a means of gaining better performance for those documents written to specification became evident. Tantek Celik, then a developer for Microsoft, created a switching mechanism in IE that corrected numerous problems. This switch uses properly formed DOCTYPE declarations to switch the browser from "quirks" mode (the forgiving mode I described in this chapter's introduction) to "compliance" mode, which allows sites written in compliant markup and CSS to perform much more efficiently. One important point: Never place anything above a DOCTYPE declaration, or you might end up with browser display issues.

To learn more about DOCTYPE switching, see http://gutfeldt.ch/matthias/articles/doctypeswitch.html. There's also a great chart there showing numerous declarations and which ones actually flip the switch. XHTML 1.0 Transitional with a proper DOCTYPE as shown in this section was chosen also because it performs this function.

Although DOCTYPE declarations are never displayed, their necessity is inarguable. Using these properly, you can't go wrong: You'll have valid pages that are also interpreted by the browser in as optimal of a situation as possible.

## Adding the *html* Element

After the DOCTYPE declaration, you'll want to begin building your document from its root element. I use the term *root* purposely because all documents create a document tree, something that we'll be exploring at length. Understanding the tree created by HTML documents plays an important role in being able to effectively style those documents using CSS.

The `html` element is considered the root element of any HTML document. Remember, the declaration isn't an HTML element—it's SGML. So the first element to appear takes on the important root status.

Example 1-2 shows the `html` element, with its opening tag and closing tag.

**EXAMPLE 1-2** The root HTML element

```
<html>

</html>
```

Pretty basic, right? Well, in XHTML, we have to add one other important piece to the opening tag, and that's the XML namespace for XHTML. This is just another way of identifying the language being used within the document. I won't go into the ideological reasons we do this, but suffice it to say that it must be there to validate (see Example 1-3).

**EXAMPLE 1-3** The root HTML element with the XML namespace attribute and value

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

</html>
```

You can see the `xmlns` attribute, which stands for "XML namespace" and the value is a URL, which, if you follow it, leads nowhere exciting, I promise! You'll just get a page saying you've reached the XML namespace for XHTML. Again, this is just another identifier.

> **NOTE**
> You'll notice a few other bits of syntax, including the `xml:lang` attribute, which defines the language of the document using XML syntax (remember, XHTML is a combo of HTML and XML), in this case `en` for English, and the `lang` attribute from HTML, declaring the same information. These are optional attributes, but we'll use them both, for full compatibility.

## The *head* and *title* Elements

Now you've got the very basic beginnings of a document, with the DOCTYPE declaration in place and the root element at the ready. You'll now begin adding other important pieces of the document, beginning with the head element. This element is where all things necessary for the document's display and performance are placed—but are not literally seen within the browser window. To create the head section, you simply add the head tags within the upper portion of your template, right below the opening <html> tag (see Example 1-4).

**EXAMPLE 1-4** Building the template: Adding a head section

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>

</head>

</html>
```

Notice that the head element requires no attributes but simply has the opening and closing tags. This identifies the head region. Table 1-1 shows some of the various elements that you can place within the head of your document.

**TABLE 1-1** Elements in the Head Portion of the Document

| Element | What It Does |
|---|---|
| `title` | This element enables you to title your document. This title will then appear in the title bar of your browser. The `title` element is required. |
| `meta` | The `meta` element is used for numerous concerns, including keywords and descriptions, character encoding, and document authorship. The `meta` element is not required, and your use of it will vary according to your specific needs. |
| `script` | This element enables you to insert scripts directly into your document or, as is the preference, link from the page to the script you'd like to use. It is used as needed. |
| `style` | The `style` element enables you to place style information into the individual page. This is known as *embedded* style, which you'll read more about in Chapter 7, "Using CSS." It is used as needed. |
| `link` | The `link` element is most commonly used to link to an external style sheet, although it can be used for other purposes, such as linking to an alternative page for accessibility, or to link to a favicon, those popular icons you see in the address bar on certain websites. |

### The *title* Element in Detail

The `title` element is the only *required* element within the `head` element. This element displays any text within it in the browser bar (see Figure 1-2) along with the browser's name at the end of the text.


**FIGURE 1-2** The title element text will appear in the browser's title bar.

Aside from the fact that you have to have the `title` element in place, writing good titles is a first-line technique that accomplishes three things:

- Provides a title for the page,
- Offers users *orientation*—that is, it helps them know where they are on the Web and within the site itself
- Provides additional information about the site page

Writing effective titles means addressing these three concerns. A good title example appears in Example 1-5.

**EXAMPLE 1-5** Title example with site name and location for user orientation

```
<title>molly.com - books - HTML &amp; CSS</title>
```

Note that the page is titled using the site name, the site section, and the subsection, providing useful information for the visitor.

An ineffective example can be seen in Example 1-6.

**EXAMPLE 1-6** Title example with site name and location for user orientation

```
<title>Read my books!</title>
```

Here, there's no information that helps us. So while the technical requirement of having a title is fulfilled, the practical needs are not.

---

**NOTE**
Although you cannot use HTML inside a title, you can use character entities, as you can see in Example 1-5, where I used the entity `&` to create the & symbol. For more information on available character entities, see Appendix A, "XHTML Reference."

---

# The *meta* Element

Although it is not required in a document, the meta element performs so many different functions that it's a good idea to become familiar with it right away.

## Document Encoding

Document encoding means setting the character set for your page, which is particularly important when writing documents in other languages. For many years, those of us writing in Latin characters (including English) used the ISO 8859-1 character set. The ISO sets and subsets cover a wide range of languages. But nowadays, we have UTF-8, a more universal format following a different standard than ISO values. UTF-8 can be helpful in a variety of browsers, but there are some limitations. If you are publishing in another language, such as Russian or Japanese, you'll want to have your document encoding set up under ISO rather than Unicode character sets.

**NOTE**
Ideally, character encoding is set on the server and not in a meta element. However, you can set it using a meta element. See http://www.webstandards.org/learn/askw3c/dec2002.html.

Example 1-7 shows a meta element that defines the UTF-8 format, suitable for documents in English as well as other languages, depending upon your browser support.

**EXAMPLE 1-7** Using meta to declare document encoding with Unicode

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
```

Example 1-8 shows a meta element for a document written in Russian, using the ISO method.

**EXAMPLE 1-8** Using meta to declare document encoding for Cyrillic, using ISO

```
<meta http-equiv="Content-Type" content="text/html; charset= iso-8859-5" />
```

## Keywords, Description, and Authorship

The meta element can be used to describe keywords, describe the site, and define the author, too. This is extremely helpful for public search engines as well as for any search engine you might be running on your own site.

Keywords are single words and word combinations that would be used during a search. This assists people looking for specific topics to find the information you're providing (see Example 1-9).

**EXAMPLE 1-9** Using `meta` for keywords and keyword combinations

```
<meta name="keywords" content="molly, molly.com, html, xhtml, css, design, web
design, development, web development, perl, color, web color, blog, web log,
weblog, books, computer books, articles, tutorials, learn, author, instructor,
instruction, instructing, training, education, consult, consultation, consultant,
famous people page, famous people list, standards, web standards, web standards
project, wsp, wasp, digital web, digital web magazine, web techniques, web
techniques magazine, web review, webreview, webreview.com, wow, world organization
of webmasters, conference, conferences, user interface, usability, accessibility,
internationalization, web culture" />
```

You'll notice that although I use the word *web* a great deal, it's in combination with other keywords. Most search engines will lock you out if you use multiple single keywords. This used to be a way of getting higher ranking, but no longer. Use keywords that make sense, or if you want to have multiple instances of a word, use it in a realistic combination.

Descriptions are typically 25 words or less and describe the purpose of your document (see Example 1-10).

**EXAMPLE 1-10** The `meta` element used for site or page description

```
<meta name="description" content="I'm Molly E. Holzschlag, and this Web site
shares my Web development work and personal thoughts." />
```

Short and to the point! Another use is to define the author of the document, as shown in Example 1-11.

**EXAMPLE 1-11** Using `meta` to describe page authorship

```
<meta name="Author" content="Molly E. Holzschlag" />
```

Of course, this information is never displayed on your web page itself. Instead, as with all elements and attributes within the head portion of a document, this information is used by the browser and other resources such as search engines.

---

**NOTE**
Other uses for the `meta` element are to refresh documents automatically and to restrict search engines from logging specific pages. Learn more at http://www.learnthat.com/courses/computer/metatags/meta.html.

---

## The *body* Element

The body element is where all the action takes place. It's the element where you'll be placing the content of your page and marking it up using XHTML to structure it accordingly. The element goes within the html element, directly below the head——makes sense, doesn't it? (See Example 1-12.)

**EXAMPLE 1-12** Placing the body element

```
<html>
<head>
<title>Appropriate Title Text Here</title>
</head>
<body>
</body>
</html>
```

When viewed in a browser, the information within the body element is what is displayed in the browser window, also referred to as the *viewport*. This is the content area only—no browser chrome (which refers to the browser's interface components, such as scrollbars and status bars). Figure 1-3 shows Google in a web browser. Only the displayed content is within the viewport.

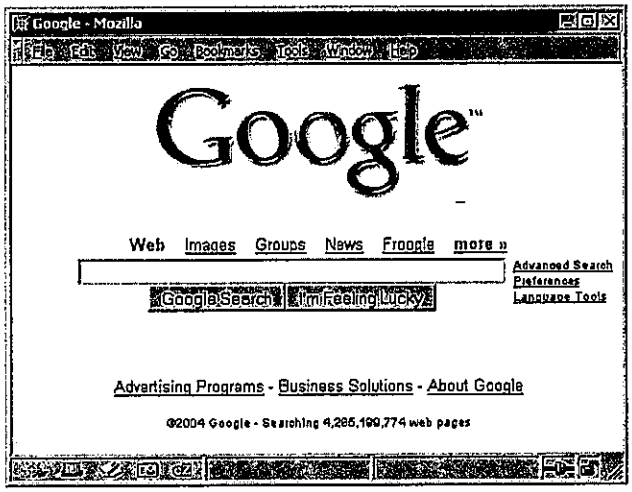



**FIGURE 1-3** Viewing body text within the browser viewport.

# HTML Comments

Another important piece of markup that you'll want to get started using right away is HTML comments. Comments enable you to hide content or markup for temporary purposes or backward compatibility, to identify sections within your document, and to provide directives for other folks who might be working on the page.

The syntax for an HTML comment looks like this:

```
<!-- -->
```

What you are hiding, identifying, or providing in terms of guidance goes between the opening and closing portions of a comment. Example 1-13 hides the text content within the body using comments.

**EXAMPLE 1-13** Hiding text content and markup

```
<body>
<!--
<p>The content of this paragraph will not appear within the body so long as it's
within a comment.</p>
-->
<p>The content of this paragraph will be displayed, because it's outside of the
comment field.</p>
</body>
```

You can denote sections within your document, as shown in Example 1-14.

**EXAMPLE 1-14** Hiding text content and markup

```
<body>

<!-- begin primary content -->


<!--begin footer information-->

</body>
```

Finally, comments are a useful way to provide directives (see Example 1-15).

**EXAMPLE 1-15** Providing guidance inside a comment

```
<body>
<!-- Angie: please be sure to use lists instead of tables in this section -->
</body>
```

# Reviewing the Template

Time to wrap up our exploration of a template by actually putting all the components together (see Example 1-16).

**EXAMPLE 1-16** Viewing the structure of an XHTML document

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<head>

<title>your site : location within site : topic title</title>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<meta name="keywords" content="your, keywords, here" />
<meta name="description" content="your description here" />
<meta name="author" content="your name here" />

</head>

<body>

<!-- main content section -->

</body>
</html>
```

Copy this markup, save it to a work folder on your computer, and name it `index.html`. This will be the file you'll open and add content and additional markup to as we progress.

## Text Is Next!

Of course, if you were to open the document you just created in a web browser, the viewport section—where content normally is displayed—would be completely blank! This is because the only thing in the body is a comment, which hides the text within it. The one thing you will notice is the title of your site, which appears in the title bar at the top of your browser.

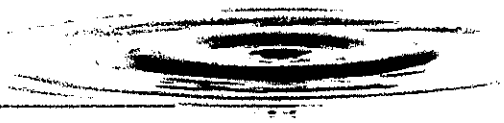
Beginning with structure is a great way to learn how to create great pages right away. Although it might seem frustrating to do all this complicated stuff and end up with no visible results, I promise you that the long-term rewards are worth it. You'll end up with far more control and understanding of everything to do with markup and CSS, I assure you.

So after you've saved your document, go ahead and validate it. Browse to http://validate.w3.org/, find the file upload section, and upload your file. Run it through the validator. Find any errors? Fix them and try again. No errors? Great job.

And I promise you that in the next chapter, "Adding Text and Links," you'll end up with something to actually view within your browser!

**Excerpt from "Spring into HTML and CSS" by Molly E. Holzschlag (Addison-Wesley Professional)**

CHAPTER 7



# Using CSS

Cascading Style Sheets have been around since the end of 1996. Despite the relative longevity of the technology, its use in real-world web design has been limited to managing fonts and color, at least until recently. This limitation was imposed by the lack of consistent browser support. Because not all browsers managed CSS equally (if at all), it has been very difficult for designers to tap into the true power of style sheets. Instead, there's been a reliance on HTML for presentation.

Now we have far better support for CSS, so to tap into its many valuable features, web designers are moving away from HTML as a means of adding style and laying out pages, and into pure CSS design. Why is this so important? The reasons are many:

- Keeping presentation separate from the document means you can style that document for numerous media, including the screen, print, projection, and even handheld devices.
- Separating presentation from the document means a lighter document, which, in turn, means the page loads and renders faster, making for happier visitors.
- CSS offers ways to control one document or millions of documents. Any time you'd like to make a change, you change that style in one location and automatically update to all the documents with which that CSS is connected. In HTML, this couldn't be done.
- CSS documents are cached. This means they are loaded into your browser's memory *one time*. As you move within a site, the browser never has to reinterpret the styles. The results are more fluid movement from page to page and faster-loading pages, which, of course, is always desirable.
- By separating presentation from structure and content, accessibility is easily achieved. Documents that don't have heavy tables and lots of presentational HTML are inherently more accessible than documents that do.

Clearly, CSS offers a lot. In this chapter, you'll learn how to set up CSS to be most efficient and flexible for your designs.

# CSS Theory Simplified

Before you can actually put CSS to use, you need to know some important things about the language and how to use it effectively. I'll try to make this quick and painless because I know you want to get down to it.

## CSS Rules

CSS rules are made up of a selector and at least one declaration. A selector is the code that selects the HTML to which you want to apply the style rule. We'll be focusing on common selectors in this book, but you can use more than a dozen selector types as you become more proficient at CSS. A declaration is made up of at least one CSS property and related property value. CSS properties define the style:

```
h1 {color: red;}
```

The `h1` is the selector for your `h1` headers, and the declaration is made up of the `color` property with a value of `red`. Simply said, this rule turns all `h1` elements red. You'll note that the syntax of the style rule looks different from what you've seen in HTML. The curly braces contain the declarations, each property is followed by a colon, and a semicolon is used after each property. You can have as many properties as you want in a rule.

## Applying CSS

Six types of style sheets exist:

- **Browser style**—This is the default style sheet within a browser. If you declare no style rules, these defaults are applied.
- **User style**—A user can write a style sheet and make it override any styles you create by changing a setting in the browser. These are used infrequently but can be helpful for individuals with special needs, such as low vision. In such a case, the user will create high-contrast, large-font styles that override your own.
- **Inline style**—This is style that is used right in the individual element and applied via the `style` attribute. It can be very useful for one-time styles by element, but it isn't considered ideal.
- **Embedded style**—This is style that controls one document and is placed inside the `style` element within the HTML document.
- **Linked style**—This is a style sheet that is linked to an HTML document using the `link` element in the head of the document. Any document linked to this style sheet gets the styles, and here's where the management power of CSS is found.

- **Imported style**—This is similar to linked styles, but it enables you to import styles into a linked style sheet as well as directly into a document. This is useful in workarounds and when managing many documents.

You'll see examples of these style sheets as we progress throughout the chapter.

### The Cascade

People often wonder where the term *cascading* comes from. The cascade is an *application hierarchy*, which is a fancy term for a system of how rules are applied. If you examine the five types of style sheets just introduced, you'll notice that there are numerous means of applying style to the same document.

What if I've got an inline style, an embedded style sheet, and a linked style sheet? The cascade determines how the rules are applied. In the case of style sheet types, user style overrides all other styles; inline style trumps embedded, linked, and imported styles; embedded style takes precedence over inline style; and linked and imported styles are treated equally, applying everywhere any of these other style sheet types are not applied. Browser style comes into play only if no style for a given element is provided; in that case, the browser style is applied.

The cascade also refers to the manner in which multiple style sheets are applied. If you have three linked style sheets, the one *on the bottom* is the one that is interpreted if any conflicts exist among those styles.

### Inheritance

Inheritance means that styles are inherited from their parent elements. Consider the following:

```
<body>
<h1>My header</h1>
<p>Subsequent Text</p>
</body>
```

Both the `h1` and `p` elements are considered children of the `body` element. The styles you give to the body will be inherited by the children until you make another rule that overrides the inherited style. Not all properties, such as margins and padding, are inherited in CSS but almost all others are.

### Specificity

Finally, if there are conflicts within any of your style sheets that aren't resolved by the cascade, CSS has an algorithm that resolves the conflict. This algorithm is based on how specific a rule is. It's a bit heavy for this discussion but worthy of mention.

Obviously, two pages can't really do justice to any of these topics, so if you're interested in learning more, be sure to look at the *Additional Resources* section.

## Adding Style Inline

Okay, enough with the theory—let's get down to work! Here you'll learn to apply inline style. You'll use inline style infrequently because it styles only the element to which it is applied. This defeats the management power of CSS.

What's more, inline style can be equated with presentational HTML because it goes right into the document instead of being separated from it, defeating the primary benefits of CSS. I use inline style mostly for situations in which a quick fix for a single element is called for, or in rare cases when it's the only style for one unique element in an entire site.

Consider the following element:

```
<h1>Welcome!</h1>
```

If this header were part of a complete HTML document and you viewed it in a browser, the results would be equivalent to Figure 7-1.


**FIGURE 7-1** Default size of an `h1` as defined by browser styles.

Say you don't like the default color and size. You can add CSS rules directly to the element using the `style` attribute:

```
<h1 style="color: gray; font-size: 24px;">Welcome!</h1>
```

Now you've got a gray header sized at 24 pixels (see Figure 7-2).


**FIGURE 7-2** Redefining color and size using inline style.

## Using Embedded Style

Embedded style controls only the document in which it is embedded. As with inline style, this defeats the purpose of being able to apply styles site-wide. However, there are good uses for embedded style. One would be if that document is the only document in the site that takes those specific styles. Another is workflow related. I like to use embedded style while working on a design because it's all in the same document. This way, I don't have to switch between applications or windows to accomplish my tasks. Because the style rules are the same, I can simply cut out the final styles from the embedded sheet and link them, which you'll see how to do in just a bit.

Embedded style is added to the `head` portion of the document, within the `style` element, and it uses the required `type` attribute (see Example 7-1).

**EXAMPLE 7.1** An HTML document snippet describing embedded style

```
<head>
<title>working with style</title>
        <style type="text/css">
        body {background-color: black; color: white;}
        h1 {font-size: 24px;}
        p {font-size: 12px;}
        </style>
</head>
<body>
<h1>Welcome!</h1>
<p>Paragraph one.</p>
<p>Paragraph two.</p>
</body>
```

Figure 7-3 shows the results.


**FIGURE 7-3** Notice how the color from the body is inherited by all its children.

# Creating a Linked Style Sheet

To truly tap into the power of CSS, you'll be using linked style sheets the majority of the time. A linked style sheet is a separate text file into which you place all your CSS rules (but *not* any HTML) and is named using the `.css` suffix. You then link any HTML file you want to have affected by that style sheet to the sheet using the link element in the head portion of the document.

Example 7-2 shows a style sheet ready for linking. In it, I've provided a range of style rules and then saved the file to my local folder, naming the file `styles.css`.

**EXAMPLE 7-2** A style sheet ready for linking

```
body {
        background-color: #999;
        color: black;
        }
h1 {
        font-family: Verdana;
        font-size: 24px;
        color: #ccc;
        }
p {
        font-family: Georgia;
        font-size: 12px;
        color: white;
}
```

In Example 7-3, you'll find the complete HTML along with the required link to the style sheet within the same directory.

**EXAMPLE 7-3** The HTML for the style sheet

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>working with style</title>

<link rel="stylesheet" type="text/css" href="styles.css" media="all" />

</head>
<body>

<h1>Welcome!</h1>
```

```
<p>Paragraph one.</p>

<p>Paragraph two.</p>

</body>
</html>
```

The results can be seen in Figure 7-4.

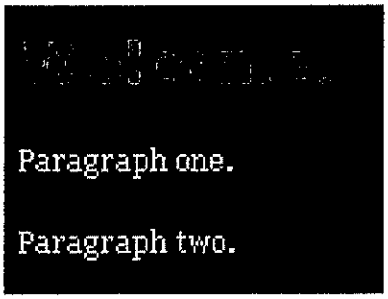



**FIGURE 7-4** Results from a linked style sheet.

Of course, you can link as many documents you want to this style sheet using the `link` element.

You'll note several attributes in use with the `link` element, as follows:

- `rel`—This is the relationship attribute, and it describes the relationship of the link. In this case, the relationship is with a primary style sheet, so the `stylesheet` value is given.
- `type`—As with the `style` element in embedded styles, you must define the type of language and format in use—in this case, `text/css`.
- `href`—This is the familiar reference attribute. In this case, I've identified only the file because both documents are in the same directory. You might want to put your style sheets into a different directory; just be sure your `href` info is accurate. You can also use absolute linking to directly link to a style sheet.
- `media`—The `media` attribute enables you to define different styles for different media. If you wanted to create a separate style sheet for this document that's for handheld devices only, you would link to that and use the `media="handheld"` attribute. Similarly, a `media="print"` attribute would send that style sheet only to print. In this case, the media is defined as `screen`. The default is `all`, so if you want the same styles to apply to all media, you can use that or simply leave out the `media` attribute.

As mentioned, you can link as many style sheets to the same document as you want.

# Importing Style Sheets

Imported style sheets are a lot like linked style sheets, in that you create a separate file for the styles you want to import. Then you can either import those sheets into a primary style sheet that is then linked to the document, or import directly into the document.

## Importing Directly into a Document

Importing into a document actually involves two types of style sheets: the separate style sheet that's to be imported (I'll call that `import.css`) and an embedded style sheet in the document. This is because importing isn't done with an element such as `link`; instead, the CSS directive `@import` is used (see Example 7-4).

**EXAMPLE 7-4** Importing style with an embedded sheet

```
<head>
<head>
<title>working with style</title>
<style type="text/css">

@import url(import.css);

</style>
</head>
```

The style sheet `@import.css` will be imported directly into the document. Imagine the `style` element being filled with all the style rules within the `import.css` file—that's exactly what happens. So now the style is actually embedded in this file.

You can use this technique for as many documents as you want, but typically this technique is used primarily in workarounds. A number of browsers, particularly Netscape 4 versions, do not support the `@import` directive, yet they do support the `link` element. Because Netscape 4.*x* has limited support for CSS and you have to take care to send styles to it, separating out those styles that you don't want it to misinterpret and those styles you know it can support into linked and imported allows Netscape users to see some, but not all, styles. This is very effective as a workaround when you must support Netscape 4 versions.

Another workaround using the `@import` directive is to simply place all styles into the imported sheet. Then any browser that doesn't support the `@import` simply won't read the styles, and a plain, unstyled document gets sent to the browser instead.

**QUANTUM LEAP: FLASH OF UNSTYLED CONTENT (FOUC)**

If you are using the `@import` technique and have no `link` or `script` element in the head of your document, Internet Explorer will often display the unstyled content first and then redraw the page with the style. It's an annoying bug but is easily avoided by adding a `link` or `script` element to the head of the document. For more about FOUC, see http://www.bluerobot.com/web/css/fouc.asp.

Most of the time, you won't be using the `@import` in an embedded sheet unless you have a very specific reason to do so.

### Importing Style into a Linked Style Sheet

Another use for the `@import` directive, and the real reason `@import` exists, is to be able to modularize your styles and then import them into the primary style sheet. Consider Figure 7-5.

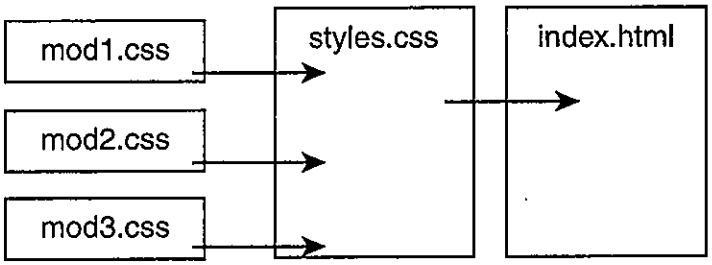



**FIGURE 7-5** Importing styles into a main sheet.

Imagine that each module file (`mod1.css`, `mod2.css`, and `mod3.css`) contains styles specific to a feature or function within your site. As an example, you might have styles set to manage ads, styles specific to tables, and styles specific to forms. You could place these in separate module files and then import them into the `styles.css` file, which is then linked to `index.html`. The rationale behind this approach is that you could make modifications to the modules independently or cut them out easily when they are no longer needed. This technique is most effective when you have very large sites with lots of styles to manage.

**QUANTUM LEAP: HACKING CSS**

Although CSS is now in widespread use, hacks sometimes must be employed to achieve cross-browser consistency. You can use the modular import method described for your hacks. This way, as soon as you no longer need the hack, you can simply delete the imported file and the `@import` directive, removing the hack completely and getting your CSS as clean as possible. For more about hacking CSS, see my InformIT article "Strategies for Long-Term CSS Hack Management," at http://www.informit.com/articles/article.asp?p=170511.

# Commenting and Formatting CSS

Just as you can add comments to your HTML files to describe sections, hide markup and content from the browser, or add directives to fellow document authors, you can comment your CSS documents. And just as HTML can be written with indentations or other personal formatting preferences, so can CSS.

## Commenting CSS

CSS comments are different than HTML comments. CSS comments open with a forward slash and an asterisk, and close with an asterisk followed by a forward slash. Any content within that area is *not interpreted* by the browser (see Example 7-5).

**EXAMPLE 7-5** Commenting CSS

```
/* global styles */

body {
        background-color: orange;
        font-family: Arial, Helvetica, sans-serif;
        color: white;
        }

/* layout styles */

#nav {
        position: absolute;
        top: 0;
        left: 0;
        width: 150px;
        }

/* hide this style and comment temporarily

.warning {
        color: red;
        }

John: please unhide the warning style when you're ready to launch */
```

Everything in bold will not be interpreted by the browser, but all the styles outside of comments will. As you can see, this can help you chunk your style sheets into logical groups, to aid both you and others to find given styles quickly. Additionally, you can hide styles you don't want for use later, and you can leave commentary for other people working with the style sheet.

You will sometimes see HTML comments surrounding CSS within an embedded sheet (see Example 7-6).

**EXAMPLE 7-6** HTML comments to hide CSS

```
<head>
<title>working with style</title>
<style type="text/css">
<!--
        body {
                        background-color: #999;
                        color: black;
                        }
        h1 {
                        font-family: Verdana;
                        font-size: 24px;
                        color: #ccc;
                        }
        p {
                        font-family: Georgia;
                        font-size: 12px;
                        color: white;
                        }
-->
</style>
</head>
```

In this case, the HTML comments are being used to hide the CSS from older browsers that do not interpret CSS. Many of those browsers would try to display the CSS rules in the browser window. Using HTML comments in this manner is still in widespread use today, although for contemporary browsers the technique is unnecessary.

## Formatting CSS

You might have noticed that I've used two formatting approaches in this chapter (sneaky, aren't I?). The first is to follow the selector with the declaration, all on the same line:

```
body {background-color: #999; color: black;}
```

The other is to break up the rule:

```
body {
        background-color: #999;
        color: black;
        }
```

Either approach is correct; it's just a matter of personal preference. Many CSS designers are of the mindset that every bit and byte counts, so they opt for the first approach. Others argue that breaking up the rule makes it easier to find the styles you want to modify. Either way, as long as all the required syntax is intact, the formatting of your style sheet is a personal choice.

## Time to Put Your Imagination to Work!

If you're thinking at this point that working with markup and CSS is no play, well, your frustration is well founded. It's imperative that you get the complexities down, and I assure you that if you've made it through thus far, you're grasping complex ideas.

But no doubt you want to put those ideas to work and really get a feel for how to use CSS to make things look good. After all, that's what I keep promising, right?

Fortunately, the next chapter sets you up for a little fun: putting your imagination to work.

You'll be using images and color to spruce up your documents, and exploring the fine control that CSS offers you when it comes to working with imagery and color in your designs.

Excerpt from "Sams Teach Yourself HTML5 Mobile Application Development" by Jennifer Kyrnin (Sams, 2011)

# HOUR 9

# Adding Meaning with HTML5 Sectioning and Semantic Elements

### *What You'll Learn in This Hour:*

- How to define sections of HTML5 documents
- The four primary sectioning elements
- Understanding elements like `<figure>` and `<details>`
- Using the many semantic elements
- Providing meaning with semantic elements

Many of the new elements in HTML5 relate to how the content is organized on the web page. These are called sectioning and heading elements. This hour you will learn more about these elements as well as some new semantic tags that help you define your web pages more precisely.

You will learn the value of marking up your pages semantically and about some of the new and existing semantic elements that you may not be using.

## What Are Sectioning Elements?

Sectioning elements were added to HTML5 to help web designers organize content. These elements create logical sections of the document, each of which would start with a heading element, and often end with a footer element.

The four sectioning elements in HTML5 are

- `<article>`
- `<aside>`
- `<nav>`
- `<section>`

Two new elements, though not explicitly sectioning elements, are sectioning roots:

- `<details>`
- `<figure>`

A few other new elements could be considered sectioning elements as well. These new elements work with the sectioning elements and sectioning roots:

- `<header>`
- `<footer>`
- `<hgroup>`

One thing to keep in mind is that sectioning and semantic elements do not, by themselves, make any visible changes to the contents of a web page. They are used to provide structure to an HTML document that a computer can read. This allows the computer to create outlines as described next or otherwise interact programmatically with the document. If you want these elements to have a visual style in your pages, you need to attach CSS styles to them.

# Using the New Sectioning Elements

One thing you should keep in mind is that the sectioning elements `<article>`, `<aside>`, `<nav>`, and `<section>` each work with the header and footer elements to create a section of your document. Following each of these elements in your document with a headline tag such as `<h1>` should be possible. And with those headline tags, your HTML5 document will have an outline.

## Creating Outlines with Sectioning Elements

HTML5 introduces a new concept into the HTML specification: outlines. HTML5 outlines look at how the document is structured, using sectioning elements and header elements to label the outline sections. These outlines are similar to ones that a word processing program might create.

The purpose of sectioning elements is to create outlines with headers and footers. Specifically, the outlines can be created with algorithms. Outlines are important because they make the pages more accessible. Accessible user agents, such as screen readers, can use the outline to determine what parts are the most important to present to the user.

An HTML5 document might look something like this:

```
<section>
    <h1>Part 1: Building Web Pages</h1>
    <section>
        <h1>Chapter 1: HTML</h1>
        <p>In this section you will learn how to write HTML
    </section>
    <section>
        <h1>Chapter 2: CSS</h1>
        <p>And this section will teach you CSS
        <section>
            <h1>CSS1</h1>
            <p>...
        </section>
    </section>
</section>
<section>
    <h1>Part 2: Publishing Web pages</h1>
    ...
</section>
```

This creates an outline that looks like this:

```
I. Part 1: Building Web Pages
    I.1 Chapter 1: HTML
    I.2 Chapter 2: CSS
        I.2.i CSS1
II. Part 2: Publishing Web pages
```

HTML 4 had no sectioning elements, and so web designers typically used the  tag to create artificial sections within their documents. Although these worked, they were inconsistent across sites, so assistive devices could never use them to help determine content priority.

**By the Way**

**Testing for the Correct Use of Sectioning Elements**

One way you can test your HTML5 document to see whether you are using the sectioning elements correctly is to test your page in the HTML5 Outliner (http://gsnedders.html5.org/outliner/). This shows you how your outline looks, and you'll be able to see what headers you may be missing.

The best thing about the new sectioning elements is how easy they make the HTML to read. For example, when you see the <article> tag, you know that the contents are an article or some other type of syndicatable content. The HTML5 outlines are a representation of that structure.

## The <article> Element

The <article> element represents a section of content that can stand on its own. If you were to syndicate a web page, you would syndicate the article portions—they are the part of the page that defines the page. An article might be

- A magazine or newspaper article
- A blog post
- A forum post
- A blog comment
- Any independent item of content

You can put a <header> and a <footer> element inside an <article>. Most articles also have some type of headline. You can also put <section> tags inside articles, or you can put an article inside a section.

The easiest way to think about what type of content should be in an article is to ask yourself whether it is content that can stand alone in syndication—such as in an RSS feed.

Here is how a simple <article> might look (assume it has a lot more content):

```
<article>
    <h1>My Pets</h1>
    <p>I have always loved animals, and I've always had a lot of pets....
</article>
```

## The <aside> Element

The <aside> element is defined as content that is tangentially or loosely related to the main content of the document. However, it can also be a sidebar on a web page, providing information that is related to the site as a whole, rather than the main content of the page.

The way you determine which type of <aside> you're using depends upon where that <aside> is found. If the <aside> is found inside an <article> tag, then it

defines content that is related to the article itself, such as a glossary or list of related articles. If the <aside> is found outside of an <article> tag, then it defines content that is related to the website as a whole, but not the article(s) on the page, necessarily. Examples would be sidebar elements such as a blogroll or advertising (that is related to the page content).

**Sectioning Elements Do Not Have Page Locations Defined**

By the Way

It is easy to think that an <aside> element will appear on the side of a web page and a <footer> element will appear at the bottom. But the placement of these elements on a web page is defined only by CSS, not by the tags. You can place <footer> content at the very top of your page, if you feel that semantically that content is information usually found in a footer, or content in an <aside> element can appear in the main column of the page. These elements are semantic only, meaning that they define what the content *is* rather than where it should display or what it should do.

Here is an example of the two types of <aside> elements. This is an <aside> inside the <article>, which indicates that it is related specifically to the content on the page:

```
<article>
    <h1>My Pets</h1>
    <p>I have always loved animals, and I've always had a lot of pets....
    <aside>
        <h1>Photos of my Pets</h1>
        <ul>
            <li><a href="">Shasta</a></li>
            <li><a href="">Suni</a></li>
            ...
        </ul>
    </aside>
</article>
```

This <aside> is outside the <article>, so it is related to the content of the site as a whole but not to the article specifically:

```
<aside>
    <h1>My Favorite Sites</h1>
    <ul>
        <li><a href="">Dogs</a></li>
        <li><a href="">Cats</a></li>
        <li><a href="">Horses</a></li>
    </ul>
</aside>
```

*Did you Know?*

**What's with All the <h1> Elements?**

One thing you may have noticed is that every section has a headline using the <h1> element. In HTML 4 the <h1> through <h6> tags were the only outlining option designers had, so the recommendation was to use only one <h1> tag—for the page title. However, HTML5 added all the additional sectioning elements, and that the recommendation is now to use <h1> as the first headline for every section. This indicates that the <h1> headline is the most important headline *of that section*, leaving the <h2> through <h6> headlines as subheads there. You may find that older browsers may respond awkwardly to this convention, especially with CSS disabled, and if so, you can continue to use <h2> for second-level headlines and so on, but the W3C recommends using <h1> as the first headline in any new section.

## The <nav> Element

The <nav> element defines a section of the content that links to other pages or areas of the site. You don't have to enclose every list of links you put on a page in a <nav> element. It is only for the major navigation of a page.

Most web designers are already using a form of the <nav> element when they put in a  or <ul> with an ID of "nav" or "navigation", such as:

```
<ul id="nav">
    <li><a href="">Home</a>
    <li><a href="">Products</a>
    <li><a href="">Services</a>
    <li><a href="">About Us</a>
</ul>
```

If you have a section on your web page like that, all you need to do is add the <nav> element around it.

```
<nav>
<h1>Navigation</h1>
<ul id="nav">
    <li><a href="">Home</a>
    <li><a href="">Products</a>
    <li><a href="">Services</a>
    <li><a href="">About Us</a>
</ul>
</nav>
```

The best way to use the <nav> element is to use it for major navigation, because that is typically the only navigation most users care about. However, what major navigation is depends on the site and the site's developers. Some sites have only one major

navigation section, with other navigation elements just being links within the site. Other sites might have three or four <nav> elements. For example, my site on About.com (http://webdesign.about.com/) has several forms of navigation:

- A bread crumb trail at the top
- Four tabs below the site name
- The "Must Reads" section
- The "Browse Topic" links
- And even the "Explore Web Design / HTML" section at the bottom of the page

Most of these could be considered major navigation, but if I were to convert this page to HTML5, I would add <nav> elements around the four tabs and the "Browse Topic" links, as those provide the best navigation through the site. Until the specification is clear on this topic, either solution, marking all navigation as <nav> or just the major navigation, is okay.

Some places you might use the <nav> element, beyond the primary navigation bar on your site include:

- Table of contents
- Previous and next links
- Breadcrumb trail

You should be careful not to confuse the <nav> element with the <menu> element. The <menu> element is a list of commands. It can be a navigation list where each item points to a different location—like a goto list. If the menu is a list of major navigation elements, then you can put it inside of a <nav> element, but it shouldn't replace the <nav> element.

## The <section> Element

The <section> element is possibly the most confusing of the new sectioning elements. Most designers who use it tend to use it incorrectly. The most common way it's used is to set up page divisions for styles, which is incorrect usage. You should be using the  element for styling your layout.

Don't feel bad if you have been using the <section> element incorrectly. Many people have. In fact, you can find it used incorrectly on many sites teaching HTML5.

According to the W3C, "The section element is not a generic container element. When an element is needed for styling purposes or as a convenience for scripting, authors are encouraged to use the div element instead."[1]

Here are a few ways you can use to determine whether you should use a <section> element:

- Does the section have a natural headline? If it doesn't, you shouldn't use a <section> element.
- Would the section be a natural part of a page outline? If it isn't, then it shouldn't be a <section>.
- Is there more purpose to the section than just the style? If you're using the <section> tag just as hooks for styles, you should use the  element instead.
- Does the section content meet the criteria of an <article>, <aside>, or <nav>? If syndicating the content makes sense then you should use an <article>. If the content is related to the site or to the article then <aside> is appropriate. Of course, if it's navigation, then the <nav> element is best.

One place you might put a <section> element is in an <aside> for your blog. Most blogs have an <aside> that includes sections for a Blogroll, latest posts, categories, and so on. Each of those could be a <section>. For example:

```
<aside>
    <section>
        <h1>Blogroll</h1>
        <ul>
            <li><a href="http://diveintohtml5.org">Dive into HTML5</a></li>
            <li><a href="http://html5gallery.com/">HTML5 Gallery</a></li>
            <li><a href="http://dev.w3.org/html5/spec/Overview.html">HTML5
➥Specification (W3C)</a></li>
        </ul>
    </section>
    <section>
        <h1>Categories</h1>
        <ul>
            <li>...</li>
        </ul>
    </section>
</aside>
```

[1] "The Section Element." HTML5 Working Draft. www.w3.org/TR/html5/sections.html#the-section-element. Referenced May 12, 2011.

**When to Use **

The  element has been the standby for designs and layout for standards-based designers for several years now, but now with HTML5, that use should lessen. In fact, you should really only use the  element when no other appropriate element exists. For now, you may want to continue to use it in conjunction with the sectioning tags (that is, <article>), but you should only do that if you are having trouble getting your pages to display correctly.

By the Way

## Sectioning Root Elements

The sectioning root elements are elements in HTML5 that can have their own outlines, but they don't contribute to the outlines of their ancestors. These elements are:

- <blockquote>
- <body>
- <details>
- <fieldset>
- <figure>
- <td>

You should already be familiar with the <body>, <blockquote>, <fieldset>, and <td> elements. The new elements in HTML5 are <details> and <figure>.

You use the <details> element to hide and show additional information about the content. It has a related element that is also new—<summary>. The <summary> element is an optional element inside of the <details> element that represents a caption or summary of the <details> element.

Use these elements to add information to your documents that isn't critical to the content. For example, you can add information about a form field in a <details> element that people can open if they need more help:

```
<input id="phone-number" type="phone">
<details>
<summary>Format</summary>
<p>(xxx) xxx-xxxx
</details>
```

You use the <figure> element to define self-contained units of content. Most commonly a figure is an image, but it can be any type of content that can stand on its own. The related element <figcaption> provides a legend or caption for a figure.

Some common figures include:

- Images or groups of images
- Blocks of code
- Poetry or quotations
- Charts and graphs

When you are deciding whether to put content into a <figure> you need to determine whether the content is essential to the content of the page. If it is essential, and the exact placement in the content is not critical, then a <figure> element is a good choice.

You can write a simple <figure> like this:

```
<figure>
<img src="shasta.jpg" alt="Shasta">
<figcaption>
<p>A photo of my dog Shasta.
</figcaption>
</figure>
```

Remember that you don't use the <figure> element around every image on your page. Banners and advertisements are not related to the page contents and so are not figures.

## Heading, Header, and Footer Elements

The heading, header, and footer elements are not technically sectioning elements because they do not contribute to the site outline, but most web designers think of them as sectioning elements because they are used to define the semantic layout of your web document.

Heading elements include <hgroup> and all the heading tags: <h1> through <h6>. Use the <hgroup> element to group a set of one or more heading elements so that they are collected as one element in the document outline. For example:

```
<h1>This is a Headline</h1>
<h2>This is a Sub-Head</h2>
<section><h1>And this is a Sub-section</h1></section>
```

This has the following outline:

```
1. This is a Headline
    1. This is a Sub-Head
    2. And this is a Sub-section
```

Suppose you group the <h1> and <h2> together in an <hgroup>:

```
<hgroup>
<h1>This is a Headline</h1>
<h2>This is a Sub-Head</h2>
</hgroup>
<section><h1>And this is a Sub-section</h1></section>
```

Then you get the following outline:

```
1. This is a Headline
    1. And this is a Sub-section
```

You can use the <header> and <footer> elements the same way you might have used  and  in your HTML. However, don't use the <hgroup> element unless you really are grouping together more than one heading tag. Writing the following is redundant and incorrect:

```
<hgroup><h1>My Headline</h1></hgroup>
```

Also, although the <hgroup> element is valid at the time of this writing, some controversy about whether it should remain in the HTML5 specification still exists, and it may be left out of the final HTML5 specification.

The <header> can contain a section's headline as well as additional information that introduces the section, such as a table of contents, dateline, or relevant icons. What is crucial to understand is that though most of the time you will use the <header> element at the top of your document, you can also include a <header> element at the beginning of any sectioning element, such as an <article>, <aside>, or <nav>.

The <footer> element acts just like the <header> element except that it comes at the end of a section. A footer contains information about the section such as who wrote it, copyright data, or even related links. Just like the <header> element, you can include a <footer> in any sectioning element, and it does not create a new section.

**By the Way**

**Don't Get Hung Up on the Name "Footer"**

Even though the most common location for a footer is at the bottom or foot of the page, you don't have to place it there in your design. For example, in a blog post, information such as the date the post was written and the author's name could be considered footer information, but you can put that footer at the top of the post if that's where you want it to appear in your design.

If you have contact information in your footer for the author of a post or the page, you should wrap it in the <address> element. You can then put the <address> in the <footer>.

There is a lot to learn when starting to build correctly sectioned HTML5 documents. But with practice you will start to understand the purpose of each of the sectioning elements so that you can build a correctly sectioned document yourself.

## ▼ Try It Yourself

### Building a Blog Using HTML5 Sectioning Elements

Blogs typically have standard page elements such as posts, comments, navigation, and so on. By following these steps you can create a blog page that contains two posts, with space for comments, site navigation, a sidebar, and a site header and footer.

1. Start your HTML5 document as you normally would:

```
<!DOCTYPE HTML>
<html lang="en-us">
<head>
    <meta charset="UTF-8">
    <title>My Blog</title>
    <script src="modernizr-1.7.min"></script>
</head>
<body class="no-js">
</body>
</html>
```

2. Add a container  inside the <body> tag so that you have a hook for your CSS styles:

```
<body class="no-js" >
    <div id="container">
    </div>
</body>
```

3. Create a <header> element inside the  to contain your blog title, tagline, and navigation:

```
<div id="container">
    <header>
        <hgroup>
            <h1>My Blog</h1>
            <h2>Where I Talk About What Interests Me</h2>
        </hgroup>
        <nav>
            <h1>Navigation</h1>
            <ul>
```

```
            <li><a href="">Home</a></li>
            <li><a href="">About Me</a></li>
            <li><a href="">Photos</a></li>
        </ul>
    </nav>
</header>
```

4. After the <header> add an <article> with a <header> that contains the post title:

```
</header>
<article>
    <header>
        <h1>Post #1</h1>
    </header>
</article>
```

5. Place your post contents after the article <header>:

```
<article>
    <header>
        <h1>Post #1</h1>
    </header>
    <p>This is my first post. It includes a photo of my dog.
    <figure>
        <img src="shasta.jpg" alt="Shasta">
        <figcaption>My Dog Shasta</figcaption>
    </figure>
</article>
```

6. Put a <footer> including your name and the date inside the <article>:

```
<footer>
    <p>By: <address><a
href="mailto:enn@html5in24hours.com">Jennifer</a></address>
    <p>Date: <time datetime="2011-05-12">May 12, 2011</time>
</footer>
</article>
```

7. Put the comments in a second <article> inside the post's <article>:

```
<article>
    <h1>Comments</h1>
    <p><a href="">Post</a> your comments or <a href="">read</a>
other comments
    </article>
</article>
```

8. Outside the post's <article> but still inside the  put an <aside> to include a site sidebar:

```
<aside>
    <h1>Learn More</h1>
    <section>
```

```
        <h1>Recent Comments</h1>
        <p>None
    </section>
    <section>
        <h1>Stay in Contact</h1>
        <p><a href="">RSS</a> or <a href="">Newsletter</a>
    </section>
</aside>
</div>
```

9. Finally, add a <footer> with your copyright information at the bottom of the document:

```
</aside>
<footer>
    <p>Copyright &copy; 2011 Jennifer Kyrnin
</footer>
</div>
```

You will want to add styles to make the page look nicer, and of course, you will want to add more content. But if you check, the HTML will outline correctly with no untitled sections.

The complete HTML is listed at www.html5in24hours.com/examples/sectioning-example-html.html.

▲

# Marking Up HTML Semantically

Marking up HTML5 documents semantically involves more than just using sectioning elements. Semantic elements describe what the content is. HTML5 has only a couple new semantic tags, but a number of HTML 4 elements have also been changed in HTML5 to be more semantic.

The benefit of using semantic HTML tags is that you provide more information to the browsers, but if the browsers don't support the tags, they just ignore them. Using them can only improve your web pages.

Mobile support by iOS and Android is really good for the sectioning and semantic elements. Android has supported them since 2.1, and iOS had partial support in 3.2 and full support in 4.0 and later.

In fact, the majority of difficulty you will have with these elements is with Internet Explorer before version 9. If you want to support these browsers, you can use the scripts mentioned in Hour 8, "Converting Web Apps to Mobile."

## Semantic HTML 4 Elements

A bunch of elements that are part of the HTML 4 specification are semantic:

- `<abbr>` – Abbreviations

**The `<acronym>` Element Is Obsolete**

By the Way

The `<acronym>` tag is a part of HTML 4 and defines acronyms—abbreviations that form words themselves, such as NASA, FAQ, or SCUBA. But because of the confusion between abbreviations and acronyms, and the fact that acronyms are also abbreviations, this element was made obsolete in HTML5.

- `<code>`—Code samples
- `<del>`—Deleted text
- `<dfn>`—Defining instance of a term
- `<em>`—Emphatic stress
- `<ins>`—Inserted stress
- `<kbd>`—Keyboard input
- `<samp>`—Sample output
- `<strong>`—Strong importance
- `<var>`—Variable or placeholder text

## Newly Semantic HTML 4 Elements

Several HTML elements were not semantic in HTML 4, but had their focus changed in HTML5 to become semantic:

- `<b>`—This used to mean just bold text, but now it represents text that would normally be displayed in bold, but doesn't have extra emphasis.
- `<hr>`—This used to represent a horizontal line, but now it represents a thematic break in the content.
- `<i>`—This used to mean italic text, but now it represents text that would normally be displayed in italics, but doesn't have any extra emphasis.
- `<s>`—This used to mean text that had a line through it (strikeout), but now represents content that is no longer accurate and has been "struck" from the document.

- `<small>`—This used to be text that was printed smaller than the surrounding text, but now it represents "small print" such as legalese.
- `<u>`—This used to be text that was underlined, but now it represents text that would normally be displayed with an underline.

**By the Way**

**Emphasis Refers to Contents Read Aloud**

When referring to text that is emphasized or not to be emphasized, this generally refers to how the text might be read in a screen reader. A book title would be italicized, but when read out loud, no special distinction would be made regarding the italics, as it is just a book title. In written form, the text may seem to stand out more, but with elements like `<i>` and `<b>` they would not be called out when read aloud.

## New Semantic Elements in HTML5

A few new semantic elements in HTML5 enable you to further define the meaning of your content: `<mark>`, `<meter>`, `<progress>`, and `<time>`.

You use the `<mark>` element to indicate text that should be highlighted or stand out for reference purposes. You see text that is marked every time you use a search engine. The words that you searched for are highlighted (usually in bold) in the results text, as you can see in Figure 9.1.

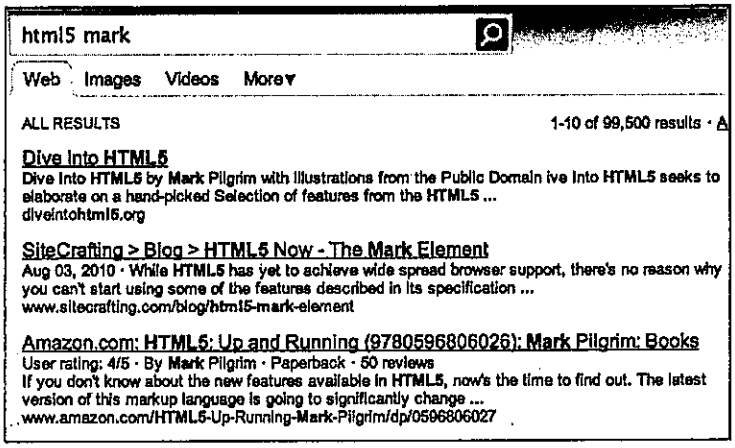



**FIGURE 9.1** Searching Bing for "HTML5 mark" with the results in bold.

You should use the `<mark>` element whenever you want to draw attention to sections of the text in a document, such as with the search terms in Figure 9.1 or to otherwise

highlight a portion of text, that wouldn't otherwise be highlighted. It is different from the  and  elements because <mark> adds no extra emphasis or importance to the text, it is just highlighted.

You use the <meter> element to indicate a scalar measurement within a range, such as a star rating on a review, a letter grade on an exam, a percentage, or a fraction.

The <meter> element has six attributes:

- **value**—The measured amount shown by the meter. For example, a grade on a paper might be written <meter value="91">A-</meter>. This attribute is required, but can be what is written in the element. For example, <meter>3 out of 15</meter>. The value is "3" and the max is "15," because that is what is written in the element.
- **min**—The minimum for the range.
- **max**—The maximum for the range. For example, <meter value="91" min="0" max="100">A-</meter>.
- **low**—The point that marks the upper boundary of the "low" portion of the meter.
- **high**—The point that marks the lower boundary of the "high" portion of the meter. Along with low, high lets you divide your meter range into three parts, for example <meter value="5" low="3" high="7">1 to 10</meter>.
- **optimum**—The point that marks the optimal point on the range.

You can also use the title attribute to specify the unit, such as inches, days, or dollars.

Use the <meter> element to define ranges and provide extra information about that range. The contents of the <meter> element can be text that is then defined by the value attribute, such as:

```
Keep the dough <meter min="80" max="90" title="degrees">warm</meter>.
```

**Meter Is Not for Measurements**

Watch Out!

Although putting a <meter> element around things such as weights and heights is tempting, if there is not a definitive range, you should leave the text alone. Using it on measurements is incorrect unless there is a known maximum value.

The <progress> element is a form element that you use to indicate the progress of something that might take time to do, such as:

- Something being downloaded or uploaded
- A computation performing
- A game loading

Following is an example <progress> element:

```
I am <progress max="24" value="12" title="chapters">half way</progress>
through the book.
```

The <progress> element has limited browser support right now, working only in Opera and Chrome. However, you can use a polyfill (such as this one at http://blogupstairs.com/html5-polyfill-progress-element/) or a detection script and a jQuery plug-in for non-compliant browsers. You detect <progress> support like this:

```
return 'value' in document.createElement('progress');
```

The <time> element indicates a date or time. You use it to mark up times and dates in your document. You can add the datetime attribute to give the precise date and time, numerically. You can also add the attribute pubdate to indicate that the time is the publication date and time of the nearest ancestor <article> element or of the document as a whole.

The datetime attribute can contain a date, a time, or a date and time. It is written according to RFC3339 (http://tools.ietf.org/html/rfc3339) in the following formats:

- **date**—YYYY-MM-DD.
- **time**—HH:MM:SS. Use a 24-hour clock; seconds are optional.
- **time with timezone**—HH:MM:SS+Zone. Timezones range from –12:00 to +14:00.
- **time at UTC**—HH:MM:SSZ Z is the same as Universal Coordinated Time (UTC), or +00:00.
- **time and date**—YYYY-MM-DDTHH:MM:SS+Zone. The T in the middle is just a T, to separate the date from the time.

The text in your <time> elements doesn't have to display a full date or time. Here are some examples:

```
<time datetime="1940-10-04">October 4, 1940</time>
<time datetime="1940-10-04">Oct 4</time>
<time datetime="00:00:00-08:00">Midnight</time>
```

One thing to keep in mind is that the <time> element cannot set specific datetime values for pre-AD dates. You also cannot encode imprecise dates such as "March 2008." To indicate dates of this nature, you need to use microformats, which Hour 15, "Microformats and Microdata," discusses in more detail.

# Summary

This hour taught you all about the new semantic elements in HTML5. You learned how to create semantic markup that can be outlined using sectioning elements as well as the semantic elements added and changed in HTML5.

Outlines may not seem important to some designers, but anyone who designs with accessibility in mind knows that outlines help screen readers use the documents more easily. Well-marked-up documents with sectioning elements create outlines that are easy to read.

Semantic elements may also not seem important, but providing extra information by indicating the semantics of text doesn't hurt either. Plus, you might be surprised at how much it does help.

# Q&A

**Q.** *Do I have to use every sectioning element in every document I create?*

**A.** No, of course not. Like all other semantic elements, you should use the sectioning elements that make sense in your document. If you don't have tangential text, then you shouldn't use the <aside>. Likewise, if you have no sections that can't be defined by other sectioning elements, then the <section> element is not needed.

**Q.** *Why do you refer to some elements as "obsolete?" I thought the term was "deprecated."*

**A.** HTML 4 was a standard written primarily for browser makers. The term *deprecated* was intended to tell them that they no longer needed to support that item in the future. HTML5 was written more for web authors and so *obsolete* was considered a clearer term.

**Q.** *What if I used the `<meter>` element with a value outside the minimum or maximum?*

**A.** Although the possibility exists that a future browser might display some type of error message if you did this, the chances are very slim. As with all semantic elements, you should try to use the elements correctly.

# Workshop

The workshop contains quiz questions to help you process what you've learned in this chapter. Try to answer all the questions before you read the answers. See Appendix A, "Answers to Quizzes," for answers.

## Quiz

1. What are the four new sectioning elements in HTML5?
2. Which of the following are valid within a `<figure>`?
    a. An image
    b. A block of text
    c. Both
3. True or False. An `<aside>` should be used to mark up a site sidebar.
4. True or False. A `<section>` should be used as just a hook for styles.
5. True or False. A sectioning root contributes its contents to the entire site outline.
6. True or False. A `<footer>` is not a sectioning element.
7. What are the new, non-sectioning, semantic elements in HTML5?

## Exercises

1. Write an HTML document with correct sectioning elements. Check your outline in the HTML5 Outliner (http://gsnedders.html5.org/outliner/). If you have any untitled sections, check to make sure that you need that section, and if you do, add a headline with an `<h1>` element.

2. Examine some existing web pages, either your own or online. See whether you can find places where the `<mark>`, `<meter>`, `<progress>`, and `<time>` elements would be appropriate. If you are examining your own pages, add in the elements so that your pages are more semantic.

Excerpt from "Sams Teach Yourself AngularJS, JavaScript, and jQuery All in One"
by Brad Dayley and Brendan Dayley (Sams, 2015)

LESSON 5

# Jumping into jQuery and JavaScript Syntax

**What You'll Learn in This Lesson:**

- Ways to add jQuery and JavaScript to your web pages
- Creating and manipulating arrays of objects
- Adding code logic to JavaScript
- Implementing JavaScript functions for cleaner code

Throughout the book, you'll see several examples of using jQuery and JavaScript to perform various dynamic tasks. jQuery doesn't replace JavaScript; it enhances it by providing an abstract layer to perform certain common tasks, such as finding elements or values, changing attributes and properties of elements, and interacting with browser events.

AngularJS uses JavaScript and jQuery syntax to provide its functionality. It is important for you to understand the jQuery and JavaScript syntax before getting into AngularJS. That is why these are covered first and AngularJS is covered in later lessons.

In this lesson, you learn the basic structure and syntax of JavaScript and how to use jQuery to ease some of the development tasks. The purpose of this lesson is to help you become familiar with the JavaScript language syntax, which is also the jQuery language syntax.

## Adding jQuery and JavaScript to a Web Page

Browsers come with JavaScript support already built in to them. That means all you need to do is add your own JavaScript code to the web page to implement dynamic web pages. jQuery, on the other hand, is an additional library, and you will need to add the jQuery library to your web page before adding jQuery scripts.

### Loading the jQuery Library

Because the jQuery library is a JavaScript script, you use the `<script>` tag to load the jQuery into your web page. jQuery can either be downloaded to your code directory and then hosted on

your web server, or you can use the hosted versions that are available at jQuery.com. The following statement shows an example of each; the only difference is that the first loads it from the jQuery CDN source and the second loads it from the web server:

```
<script src="http://code.jquery.com/jquery-latest.min.js"></script>
<script src="includes/js/jquery-latest.min.js"></script>
```

CAUTION

Remember that you need to place the `<script>` element to load the jQuery library before any script elements that are using it. Otherwise, those libraries will not be able to link up to the jQuery code.

The jQuery library downloads can be found at the following location:

```
http://jquery.com/download/
```

The jQuery library hosted links can be found at the following location:

```
http://code.jquery.com/
```

## Implementing Your Own jQuery and JavaScript

JQuery code is implemented as part of JavaScript scripts. To add jQuery and JavaScript to your web pages, first add a `<script>` tag that loads the jQuery library, and then add your own `<script>` tags with your custom code.

The JavaScript code can be added inside the `<script>` element, or the `src` attribute of the `<script>` element can point to the location of a separate JavaScript document. Either way, the JavaScript will be loaded in the same manner.

The following is an example of a pair of `<script>` statements that load jQuery and then use it. The `document.write()` function just writes text directly to the browser to be rendered:

```
<script src="http://code.jquery.com/jquery-latest.min.js"></script>
<script>
  function writeIt(){
    document.write("jQuery Version " + $().jquery + " loaded.");
  }
</script>
```

NOTE

The `<script>` tags do not need to be added to the `<head>` section of the HTML document; they can also be added in the body. It's useful to add simple scripts directly inline with the HTML elements that are consuming them.

## Accessing HTML Event Handlers

So after you add your JavaScript to the web page, how do you get it to execute? The answer is that you tie it to the browser events. Each time a page or element is loaded, the user moves or clicks the mouse or types a character, an HTML event is triggered.

Each supported event is an attribute of the object that is receiving the event. If you set the attribute value to a JavaScript function, the browser will execute your function when the event is triggered.

For example, the following will execute the `writeIt()` function when the body of the HTML page is loaded:

```
<body onload="writeIt()">
```

TRY IT YOURSELF ▼

### Implementing JavaScript and jQuery

Those are the basic steps. Now it is time to try it yourself. Use the following steps to add jQuery to your project and use it dynamically in a web page:

1. In Eclipse, create a source folder named lesson05.
2. In the same folder as the lesson05 folder, add an additional directory called js.
3. Now create a source file named jquery_version.html in the lesson05 folder.
4. Add the usual basic elements (html, head, body).
5. Inside the `<head>` element, add the following line to load the library you just downloaded:

```
06      <script src="https://code.jquery.com/jquery-2.1.3.min.js"></script>
```

6. Now you can add your own `<script>` tag with the following code to print out the jQuery version to the browser windows:

```
07      <script>
08          function writeIt(){
09              document.write("jQuery Version " + $().jquery + " loaded");
10          }
11      </script>
```

7. To have your script execute when the document is loaded, tie the `writeIt()` function to the `<body>` `onload` event using the following line:

```
13    <body onload="writeIt()">
```

8. Save the file and view it in your web browser at the following location. The output should be similar to Figure 5.1:

```
http://localhost/lesson06/jquery_version.html
```

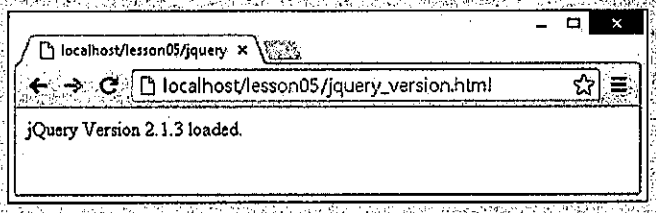



**FIGURE 5.1**
The function `writeIt()` is executed when the body loads and writes the jQuery version to the browser.

**LISTING 5.1** jquery_version.html: Very Basic Example of Loading Using jQuery in a Web Page to Print Out Its Own Version

```
01 <!DOCTYPE html>
02 <html>
03   <head>
04     <title>jQuery Version</title>
05     <meta charset="utf-8" />
06     <script src="https://code.jquery.com/jquery-2.1.3.min.js"></script>
07     <script>
08       function writeIt(){
09         document.write("jQuery Version " + $().jquery + " loaded.");
10       }
11     </script>
12   </head>
13   <body onload="writeIt()">
14   </body>
15 </html>
```

# Accessing the DOM

One of the most important aspects of JavaScript, and especially jQuery, is the capability to access and manipulate the DOM. Accessing the DOM is how you make the web page dynamic by changing styles, size, position, and values of elements.

In the following sections, you learn about accessing the DOM through traditional methods via JavaScript and the much improved methods using jQuery selectors. These sections are a brief introduction. You will get plenty of practice as the lessons roll on.

## Using Traditional JavaScript to Access the DOM

Traditionally, JavaScript uses the global `document` object to access elements in the web page. The simplest method of accessing an element is to directly refer to it by `id`. For example, if you have a paragraph with the `id="question"`, you can access it via the following JavaScript `getElementById()` function:

```
var q = document.getElementById("question");
...
<p id="question">Which method to you prefer?</p>
```

Another helpful JavaScript function that you can use to access the DOM elements is `getElementsByTagName()`. This returns a JavaScript array of DOM elements that match the tag name. For example, to get a list of all the `<p>` elements, use the following function call:

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

## Using jQuery Selectors to Access HTML Elements

Accessing HTML elements is one of jQuery's biggest strengths. jQuery uses selectors that are very similar to CSS selectors to access one or more elements in the DOM; hence the name jQuery. jQuery returns back either a single element or an array of jQuerified objects. jQuerified means that additional jQuery functionality has been added to the DOM object, allowing for much easier manipulation.

The syntax for using jQuery selectors is `$(selector).action()`, where *`selector`* is replaced by a valid selector and *`action`* is replaced by a jQuerified action attached to the DOM element(s).

For example, the following command finds all paragraph elements in the HTML document and sets the CSS `font-weight` property to `bold`:

```
$("p").css('font-weight', 'bold');
```

TRY IT YOURSELF ▼

### Using jQuery and JavaScript to Access DOM Elements

Now to solidify the concepts, you'll run through a quick example of accessing and modifying DOM elements using both jQuery and JavaScript. Use the following steps to build the HTML document shown in Listing 5.2:

1. Create a source file named dom_elements.html in the lesson05 folder.
2. Add the usual basic elements (html, head, body).
3. Inside the `<head>` element, add the following line to load the library you just downloaded:

```
06      <script src="https://code.jquery.com/jquery-2.1.3.min.js"></script>
```

4. Add the following `<script>` element that accesses the DOM using both the JavaScript and jQuery methods. Notice that with jQuery, two actions are chained together. The first sets the CSS `font-weight` property and the second changes text contained in element. With JavaScript, you use the `getElementById()` method, and then you set the `innerHTML` property directly in the DOM to change the text displayed in the browser:

```
07      <script>
08        function writeIt(){
09          $("#heading").css('font-weight', 'bold').html("jQuery");
10          var q = document.getElementById("question");
11          q.innerHTML = "I Prefer jQuery!";
12        }
13      </script>
```

5. To have your script execute when the document is loaded, tie the `writeIt()` function to the `<body>` `onload` event using the following line:

```
15    <body onload="writeIt()">
```

6. Add the following `<p>` elements to the `<body>` to provide containers for the JavaScript code to access:

```
16      <p id="heading">jQuery or JavaScript</p>
17      <p id="question">Which method to you prefer?</p>
```

7. Save the file and view it in a web browser. The output should be similar to Figure 5.2.

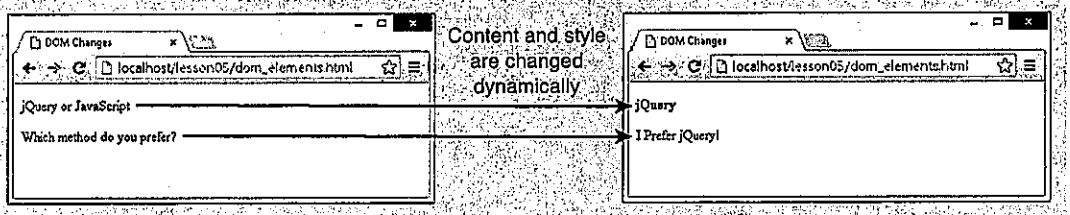



**FIGURE 5.2**
The function `writeIt()` is executed when the body loads and changes the content and appearance of the text.

**LISTING 5.2** Very Basic Example of Using JavaScript and jQuery to Access DOM Elements

```
01 <!DOCTYPE html>
02 <html>
03   <head>
04     <title>DOM Changes</title>
05     <meta charset="utf-8" />
06     <script src="https://code.jquery.com/jquery-2.1.3.min.js"></script>
07     <script>
08       function writeIt(){
09         $("#heading").css('font-weight', 'bold').html("jQuery");
10         var q = document.getElementById("question");
11         q.innerHTML = "I Prefer jQuery!";
12       }
13     </script>
14   </head>
15   <body onload="writeIt()">
16     <p id="heading">jQuery or JavaScript</p>
17     <p id="question">Which method do you prefer?</p>
18   </body>
19 </html>
```

# Understanding JavaScript Syntax

Like any other computer language, JavaScript is based on a rigid syntax where specific words mean different things to the browser as it interprets the script. This section is designed to walk you through the basics of creating variables, working with data types, and using looping and functions in JavaScript to manipulate your web pages.

TIP

For the simple JavaScript examples in this lesson, you can test them by starting Node.js using the node command from a console prompt to bring up the Node.js interpreter. From the interpreter, you can type in JavaScript code and have it execute as you type each line.

## Creating Variables

The first place to begin with in JavaScript is variables. Variables are a means to name data so that you can use that name to temporarily store and access data from your JavaScript files.

Variables can point to simple data types, such as numbers or strings, or they can point to more complex data types, such as objects.

To define a variable in JavaScript, you must use the `var` keyword and then give the variable a name; for example:

```
var myData;
```

You can also assign a value to the variable in the same line. For example, the following line of code creates a variable `myString` and assigns it the value of "Some Text":

```
var myString = "Some Text";
```

This works as well as the following:

```
var myString;
myString = "Some Text";
```

After you have declared the variable, you can use the name to assign the variable a value and access the value of the variable. For example, the following code stores a string into the `myString` variable and then uses it when assigning the value to the `newString` variable:

```
var myString = "Some Text";
var newString = myString + " Some More Text";
```

Your variable names should describe the data that is stored in them so that it is easy to use them later in your program. The only rule for creating variable names is that they must begin with a letter, `$`, or `_`, and they cannot contain spaces. Also remember that variable names are case sensitive, so using `myString` is different from `MyString`.

## Understanding JavaScript Data Types

JavaScript uses data types to determine how to handle data that is assigned to a variable. The variable type will determine what operations you can perform on the variable, such as looping or executing. The following list describes the most common types of variables that we will be working with through the book:

- **`String`**—Stores character data as a string. The character data is specified by either single or double quotes. All the data contained in the quotes will be assigned to the string variable. For example:

  ```
  var myString = 'Some Text';
  var anotherString = "Some Other Text";
  ```

- **Number**—Stores the data as a numerical value. Numbers are useful in counting, calculations, and comparisons. Some examples are as follows:

```
var myInteger = 1;
var cost = 1.33;
```

- **Boolean**—Stores a single bit that is either true or false. Booleans are often used for flags. For example, you might set a variable to false at the beginning of some code and then check it on completion to see whether the code execution hit a certain spot. The following shows an example of defining a true and a false variable:

```
var yes = true;
var no = false;
```

- **Array**—An indexed array is a series of separate distinct data items all stored under a single variable name. Items in the array can be accessed by their zero-based index using the `[index]`. The following is an example of creating a simple array and then accessing the first element, which is at index 0:

```
var arr = ["one", "two", "three"]
var first = arr[0];
```

- **Associative Array/Objects**—JavaScript does support the concept of an associative array, meaning accessing the items in the array by a name instead of an index value. However, a better method is to use an object literal. When you use an object literal, you can access items in the object using `object.property` syntax. The following example shows how to create and access an object literal:

```
var obj = {"name":"Brad", "occupation":"Hacker", "age", "Unknown"};
var name = obj.name;
```

- **Null**—At times, you do not have a value to store in a variable, either because it hasn't been created or you are no longer using it. At this time, you can set a variable to `null`. That way, you can check the value of the variable in your code and use it only if it is not `null`:

```
var newVar = null;
```

NOTE

JavaScript is a typeless language, meaning you do not need to tell the browser what data type the variable is; the interpreter will automatically figure out the correct data type for the variable.

# Using Operators

JavaScript operators provide the capability to alter the value of a variable. You are already familiar with the = operator because you used it several times in the book already. JavaScript provides several operators that can be grouped into two types—arithmetic and assignment.

## Arithmetic Operators

Arithmetic operators are used to perform operations between variable and direct values. Table 5.1 shows a list of the arithmetic operations along with the results that get applied.

**TABLE 5.1** Table Showing JavaScripts' Arithmetic Operators as Well as Results Based on y=4 to Begin With

| Operator | Description | Example | Resulting x | Resulting y |
|---|---|---|---|---|
| + | Addition | `x=y+5` | `9"45"` | 4 |
| | | `x=y+"5"` | `"Four44"` | 4 |
| | | `x="Four"+y+"4"` | | 4 |
| - | Subtraction | `x=y-2` | 2 | 4 |
| ++ | Increment | `x=y++` | 4 | 5 |
| | | `x=++y` | 5 | 5 |
| -- | Decrement | `x=y--` | 4 | 3 |
| | | `x=--y` | 3 | 3 |
| * | Multiplication | `x=y*4` | 16 | 4 |
| / | Division | `x=10/y` | 2.5 | 4 |
| % | Modulous (remainder of Division) | `x=y%3` | 1 | 4 |

TIP

The + operator can also be used to add strings or strings and numbers together. This allows you to quickly concatenate strings and add numerical data to output strings. Table 5.1 shows that when adding a numerical value and a string value, the numerical value is converted to a string, and then the two strings are concatenated.

## Assignment Operators

Assignment operators are used to assign a value to a variable. You are probably used to the = operator, but there are several forms that allow you to manipulate the data as you assign the value. Table 5.2 shows a list of the assignment operations along with the results that get applied.

**TABLE 5.2** JavaScripts' Assignment Operators as Well as Results Based on x=10 to Begin With

| Operator | Example | Equivalent Arithmetic Operators | Resulting x |
|---|---|---|---|
| = | x=5 | x=5 | 5 |
| += | x+=5 | x=x+5 | 15 |
| -= | x-=5 | x=x-5 | 5 |
| *= | x*=5 | x=x*5 | 50 |
| /= | x/=5 | x=x/5 | 2 |
| %= | x%=5 | x=x%5 | 0 |

# Applying Comparison and Conditional Operators

Conditionals are a way to apply logic to your applications so that certain code will be executed only under the correct conditions. This is done by applying comparison logic to variable values. The following sections describe the comparisons available in JavaScript and how to apply them in conditional statements.

## Comparison Operators

A comparison operator evaluates two pieces of data and returns true if the evaluation is correct or false if the evaluation is not correct. Comparison operators compare the value on the left of the operator against the value on the right.

The simplest way to help you understand comparisons is to provide a list with some examples. Table 5.3 shows a list of the comparison operators along with some examples.

**TABLE 5.3** JavaScripts' Comparison Operators as Well as Results Based on x=10 to Begin With

| Operator | Example | Example | Result |
|---|---|---|---|
| == | Is equal to (value only) | x==8 | false |
| | | x==10 | true |
| === | Both value and type are equal | x===10 | true |
| | | x==="10" | false |
| != | Is not equal | x!=5 | true |
| !== | Both value and type are not equal | x!=="10" | true |
| | | x!==10 | false |
| > | Is greater than | x>5 | true |

| Operator | Example | Example | Result |
|---|---|---|---|
| >= | Is greater than or equal to | x>=10 | true |
| < | Is less than | x<5 | false |
| <= | Is less than or equal to | x<=10 | true |

You can chain multiple comparisons together using logical operators. Table 5.4 shows a list of the logical operators and how to use them to chain comparisons together.

**TABLE 5.4** JavaScripts' Comparison Operators as Well as Results Based on x=10 and y=5 to Begin With

| Operator | Description | Example | Result |
|---|---|---|---|
| && | and | (x==10 && y==5) (x==10 && y>x) | true<br>false |
| \|\| | or | (x>=10 \|\| y>x) (x<10 && y>x) | true<br>false |
| ! | not | !(x==y) !(x>y) | true<br>false |
| | mix | (x>=10 && y<x \|\| x==y) ((x<y \|\| x>=10) && y>=5) (!(x==y) && y>=10) | true<br>true<br>false |

## If

An `if` statement enables you to separate code execution based on the evaluation of a comparison. The syntax is shown in the following lines of code where the conditional operators are in `()` parenthesis and the code to execute if the conditional evaluates to true is in `{}` brackets:

```
if(x==5){
  do_something();
}
```

In addition to executing code only within the `if` statement block, you can specify an else block that will get executed only if the condition is false. For example:

```
if(x==5){
  do_something();
} else {
  do_something_else();
}
```

You can also chain `if` statements together. To do this, add a conditional statement along with an `else` statement. For example:

```
if(x<5){
  do_something();
} else if(x<10) {
  do_something_else();
} else {
  do_nothing();
}
```

### switch

Another type of conditional logic is the `switch` statement. The `switch` statement allows you to evaluate an expression once and then, based on the value, execute one of many sections of code.

The syntax for the switch statement is the following:

```
switch(expression){
  case value:
    <code to execute>
    break;
  case value2:
    <code to execute>
    break;
  default:
    <code to execute if not value or value2>
}
```

This is what is happening. The `switch` statement will evaluate the expression entirely and get a value. The value may be a string, a number, a Boolean, or even an object. The `switch` value is then compared to each value specified by the `case` statement. If the value matches, the code in the `case` statement is executed. If no values match, the `default` code is executed.

NOTE

Typically, each `case` statement will include a break command at the end to signal a break out of the `switch` statement. If no break is found, code execution will continue with the next `case` statement.

▼ TRY IT YOURSELF

## Applying If Conditional Logic in JavaScript

To help you solidify using JavaScript conditional logic, use the following steps to build conditional logic into the JavaScript for a dynamic web page. The final version of the HTML document is shown in Listing 5.3:

1. Create a source file named if_logic.html in the lesson05 folder.
2. Create a folder under lesson05 named images.
3. Add your own images for day.png and night.png to the ./images folder in your project or download the ones from the book's website.
4. Add the usual basic elements (html, head, body).
5. Add the following `<script>` element that gets the lesson value using the `Date().getLessons()` JavaScript code. The code uses `if` statements to determine the time of day and does two things: it writes a greeting onto the screen and sets the value of the `timeOfDay` variable:

```
06      <script>
07          function writeIt(){
08              var lesson = new Date().getLessons();
09              var timeOfDay;
10              if(lesson>=7 && lesson<12){
11                  document.write("Good Morning!");
12                  timeOfDay="morning";
13              } else if(lesson>=12 && lesson<18) {
14                  document.write("Good Day!");
15                  timeOfDay="day";
16              } else {
17                  document.write("Good Night!");
18                  timeOfDay="night";
19              }
32          }
33      </script>
```

6. Now add the following `switch` statement that uses the value of `timeOfDay` to determine which image to display in the web page:

```
20              switch(timeOfDay){
21                  case "morning":
22                  case "day":
23                      document.write("<img src='images/day.png' />");
24                      break;
25                  case "night":
26                      document.write("<img src='images/night.png' />");
27                      break;
```

TRY IT YOURSELF

```
28            default:
29              document.write("<img src='images/day.png' />");
30          }
```

7. Save the file and view it in a web browser. The output should be similar to Figure 5.3, depending on what time of day it is.


**FIGURE 5.3**
The function `writeIt()` is executed when the body loads and changes the greeting and image displayed on the web page.

**LISTING 5.3** if_logic.html Simple Example of Using Conditional Logic Inside JavaScript

```
01 <!DOCTYPE html>
02 <html>
03   <head>
04     <title>If Logic</title>
05     <meta charset="utf-8" />
06     <script>
07       function writeIt(){
08         var hour = new Date().getHours();
09         var timeOfDay;
10         if(hour>=7 && hour<12){
11           document.write("Good Morning!");
12           timeOfDay="morning";
```

```
13          } else if(hour>=12 && hour<18) {
14            document.write("Good Day!");
15            timeOfDay="day";
16          } else {
17            document.write("Good Night!");
18            timeOfDay="night";
19          }
20          switch(timeOfDay){
21            case "morning":
22            case "day":
23              document.write("<img src='../images/day.png' />");
24              break;
25            case "night":
26              document.write("<img src='../images/night.png' />");
27              break;
28            default:
29              document.write("<img src='../images/day.png' />");
30          }
31        }
32      </script>
33    </head>
34    <body onload="writeIt()">
35    </body>
36  </html>
```

## Implementing Looping

Looping is a means to execute the same segment of code multiple times. This is extremely useful when you need to perform the same tasks on a set of DOM objects, or if you are dynamically creating a list of items.

JavaScript provides functionality to perform `for` and `while` loops. The following sections describe how to implement loops in your JavaScript.

### `while` Loops

The most basic type of looping in JavaScript is the `while` loop. A `while` loop tests an expression and continues to execute the code contained in its { } brackets until the expression evaluates to false.

For example, the following `while` loop executes until the value of `i` is equal to 5:

```
var i = 1;
while (i<5){
```

```
  document.write("Iteration " + i + "<br>");
  i++;
}
```

The resulting output to the browser is as follows:

```
Iteration 1
Iteration 2
Iteration 3
Iteration 4
```

### `do/while` Loops

Another type of `while` loop is the `do/while` loop. This is useful if you always want to execute the code in the loop at least once and the expression cannot be tested until the code has executed at least once.

For example, the following `do/while` loop executes until the value of day is equal to Wednesday:

```
var days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"];
var i=0;
do{
  var day=days[i++];
  document.write("It's " + day + "<br>");
} while (day != "Wednesday");
```

The resulting output to the browser is as follows:

```
It's Monday
It's Tuesday
It's Wednesday
```

### `for` Loops

The JavaScript `for` loop allows you to execute code a specific number of times by using a `for` statement that combines three statements into one using the following syntax:

```
for (statement 1; statement 2; statement 3;){
  code to be executed;
}
```

The `for` statement uses those three statements as follows when executing the loop:

- **statement 1**—Executed before the loop begins and not again. This is used to initialize variables that will be used in the loop as conditionals.
- **statement 2**—Expression that is evaluated before each iteration of the loop. If the expression evaluates to true, the loop is executed; otherwise, the `for` loop execution ends.

- statement 3—Executed each iteration after the code in the loop has executed. This is typically used to increment a counter that is used in statement 2.

To illustrate a `for` loop, check out the following example. The example not only illustrates a basic `for` loop, it also illustrates the capability to nest one loop inside another:

```
for (var x=1; x<=3; x++){
  for (var y=1; y<=3; y++){
    document.write(x + " X " + y + " = " + (x*y) + "<br>");
  }
}
```

The resulting output to the web browser is as follows:

```
1 X 1 = 1
1 X 2 = 2
1 X 3 = 3
2 X 1 = 2
2 X 2 = 4
2 X 3 = 6
3 X 1 = 3
3 X 2 = 6
3 X 3 = 9
```

## `for/in` Loops

Another type of for loop is the `for/in` loop. The `for/in` loop executes on any data type that can be iterated on. For the most part, you will use the `for/in` loop on arrays and objects. The following example illustrates the syntax and behavior of the `for/in` loop on a simple array:

```
var days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"];
for (var idx in days){
  document.write("It's " + days[idx] + "<br>");
}
```

Notice that the variable `idx` is adjusted each iteration through the loop from the beginning array index to the last. The resulting output is as follows:

```
It's Monday
It's Tuesday
It's Wednesday
It's Thursday
It's Friday
```

### Interrupting Loops

When working with loops, at times you need to interrupt the execution of code inside the code itself without waiting for the next iteration. There are two ways to do this using the `break` and `continue` keywords.

The `break` keyword stops execution of the `for` or `while` loop completely. The `continue` keyword, on the other hand, stops execution of the code inside the loop and continues on with the next iteration. Consider the following examples:

Using a `break` if the day is `Wednesday`:

```
var days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"];
for (var idx in days){
  if (days[idx] == "Wednesday")
    break;
  document.write("It's " + days[idx] + "<br>");
}
```

When the value is `Wednesday`, loop execution stops completely:

```
It's Monday
It's Tuesday
```

Using a `continue` if the day is `Wednesday`:

```
var days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"];
for (var idx in days){
  if (days[idx] == "Wednesday")
    continue;
  document.write("It's " + days[idx] + "<br>");
}
```

Notice that the write is not executed for `Wednesday` because of the continue; however, the loop execution did complete:

```
It's Monday
It's Tuesday
It's Thursday
It's Friday
```

## Creating Functions

One of the most important parts of JavaScript is making code that is reusable by other code. To do this, you combine your code into functions that perform specific tasks. A function is a series of code statements combined in a single block and given a name. The code in the block can then be executed by referencing that name.

## Defining Functions

Functions are defined using the keyword `function` followed by a function name that describes the use of the function, list of zero or more arguments in `()` parentheses, and a block of one or more code statements in `{}` brackets. For example, the following is a function definition that writes "Hello World" to the browser:

```
function myFunction(){
  document.write("Hello World");
}
```

To execute the code in `myFunction()`, all you need to do is add the following line to the main JavaScript or inside another function:

```
myFunction();
```

## Passing Variables to Functions

Frequently, you will need to pass specific values to functions that they will use when executing their code. Values are passed in comma-delimited form to the function. The function definition will need a list of variable names in the `()` parentheses that match the number being passed in. For example, the following function accepts two arguments, a `name` and `city`, and uses them to build the output string:

```
function greeting(name, city){
  document.write("Hello " + name);
  document.write(". How is the weather in " + city);
}
```

To call the `greeting()` function, we need to pass in a `name` value and a `city` value. The value can be a direct value or a previously defined variable. To illustrate this, the following code will execute the `greeting()` function with a `name` variable and a direct string for the `city`:

```
var name = "Brad";
greeting(name, "Florence");
```

## Returning Values from Functions

Often, functions will need to return a value to the calling code. Adding a `return` keyword followed by a variable or value will return that value from the function. For example, the following code calls a function to format a string, assigns the value returned from the function to a variable, and then writes the value to the browser:

```
function formatGreeting(name, city){
  var retStr = "";
  retStr += "Hello <b>" + name + "</b><br>";
  retStr += "Welcome to " + city + "!";
 return retStr;
}
```

```
var greeting = formatGreeting("Brad", "Rome");
document.write(greeting);
```

You can include more than one `return` statement in the function. When the function encounters a `return` statement, code execution of the function is stopped immediately. If the `return` statement contains a value to return, that value is returned. The following example shows a function that tests the input and returns immediately if it is zero:

```
function myFunc(value){
  if (value == 0)
    return;
  code_to_execute_if_value_nonzero;
}
```

TRY IT YOURSELF ▼

## Creating JavaScript Functions

To help solidify functions, use the following steps to integrate some functions into a JavaScript application. The following steps take you through the process of creating a function, calling it to execute code, and then handling the results returned:

1. Create a source file named js_functions.html in the lesson05 folder.
2. Add the usual basic elements (html, head, body).
3. Add a `<script>` tag to the `<head>` element to house the JavaScript.
4. Insert the following object literal definition at the beginning of the script. The object will have planet names for attributes, and each hero name is a reference to an array of villains:

```
07      var superData = {"Super Man":["Lex Luther"],
08                       "Bat Man":["Joker", "Riddler",],
09                       "Spider Man":["Green Goblin",
10                                     "Vulture", "Carnage"],
11                       "Thor":["Loki", "Frost Giants"]};
```

5. Add the following function that will be called by the `onload` event. In this function, you use a nested `for/in` loop to iterate through the `superData` object attributes. The outer loop gets the hero `name` and the inner loop loops through the index of the `villains` array:

```
12      function writeIt(){
13        for (hero in superData){
14          var villains = superData[hero];
15          for (villainIdx in villains){
16            var villain = villains[villainIdx];
17            var listItem = makeListItem(hero, villain);
18            document.write(listItem);
```

```
19            }
20        }
21    }
```

6. Notice that on line 16 of the `writeIt()` function is a call to `makeListItem()`. That function needs to return a value that can be used in line 17 to write to the document. Add the following code to create the function. The function takes two arguments: a `name` and a `value`, then generates an HTML string to create a `<li>` element and returns the string:

```
22    function makeListItem(name, value) {
23        var itemStr = "<li>" + name + ": " + value + "</li>";
24        return itemStr;
25    }
```

7. Save the file and open it in a web browser. You should see the results shown in Figure 5.4. You have just created two JavaScript functions: one that takes no arguments and does not return a value and the other that takes two arguments and returns a formatted HTML string containing the argument strings.

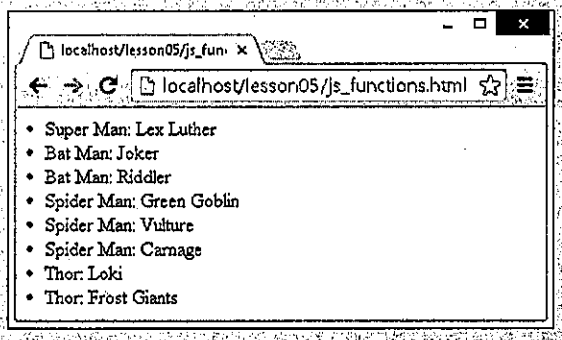
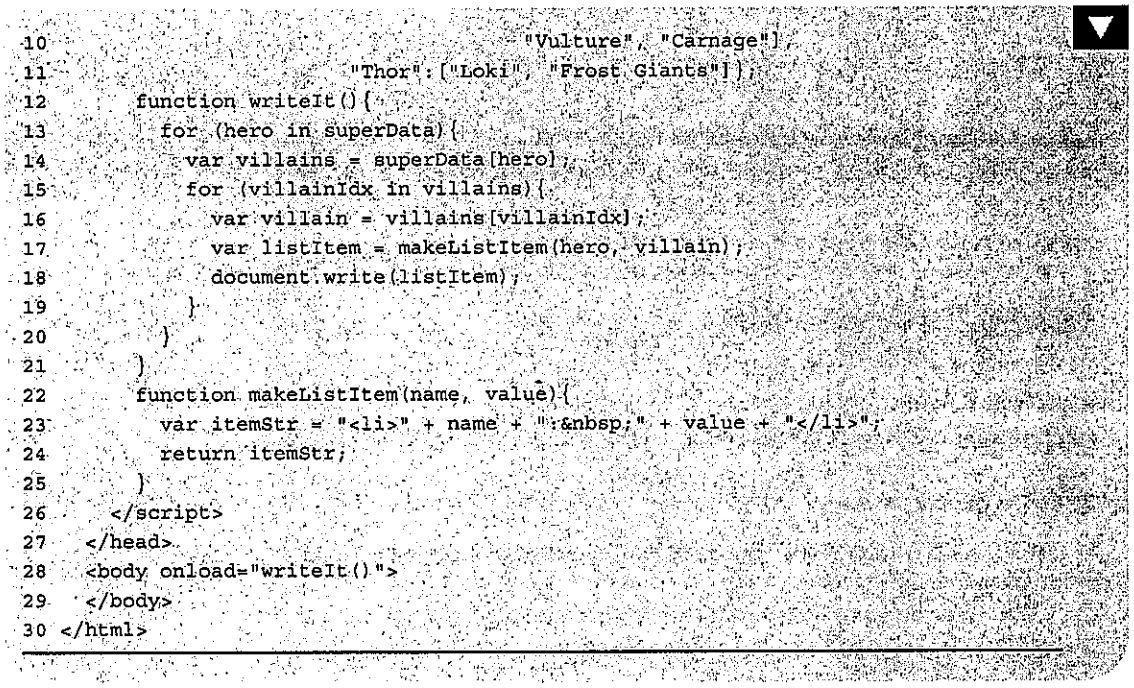



**FIGURE 5.4**
The function `writeIt()` is executed, which iterates through the `moonData` object and makes calls to the `makeListItem()` function to format the planet and moon names as an HTML `<li>` element.

**LISTING 5.4** js_functions.html Simple Example of JavaScript Functions

```
01 <!DOCTYPE html>
02 <html>
03   <head>
04     <title>JavaScript Functions</title>
05     <meta charset="utf-8" />
06     <script>
07       var superData = {"Super Man":["Lex Luther"],
08                        "Bat Man":["Joker", "Riddler"],
09                        "Spider Man":["Green Goblin",
```



```
10                                    "Vulture", "Carnage"],
11                        "Thor":["Loki", "Frost Giants"]};
12        function writeIt(){
13          for (hero in superData){
14            var villains = superData[hero];
15            for (villainIdx in villains){
16              var villain = villains[villainIdx];
17              var listItem = makeListItem(hero, villain);
18              document.write(listItem);
19            }
20          }
21        }
22        function makeListItem(name, value){
23          var itemStr = "<li>" + name + ": " + value + "</li>";
24          return itemStr;
25        }
26      </script>
27    </head>
28    <body onload="writeIt()">
29    </body>
30  </html>
```

## Understanding Variable Scope

After you start adding conditions, functions, and loops to your JavaScript applications, you need to understand variable scoping. Variable scope is simply this: "what is the value of a specific variable name at the current line of code being executed."

JavaScript enables you to define both a global and a local version of the variable. The global version is defined in the main JavaScript, and local versions are defined inside functions. When you define a local version in a function, a new variable is created in memory. Within that function, you will be referencing the local version. Outside that function, you will be referencing the global version.

To understand variable scoping a bit better, consider the following code:

```
01 <script>
02   var myVar = 1;
03   function writeIt(){
04     var myVar = 2;
05     document.write(myVar);
06     writeMore();
07   }
08   function writeMore(){
```

```
09      document.write(myVar);
10    }
11 </script>
```

The global variable `myVar` is defined on line 2. Then on line 4, a local version is defined within the `writeIt()` function. So, line 5 will write to the document. Then in line 6, `writeMore()` is called. Because there is no local version of `myVar` defined in `writeMore()`, the value of the global `myVar` is written in line 9.

## Adding Error Handling

An important part of JavaScript coding is adding error handling for instances where there may be problems. By default, if a code exception occurs because of a problem in your JavaScript, the script fails and does not finish loading. This is not usually the desired behavior.

### `Try/Catch` Blocks

To prevent your code from totally bombing out, use `try/catch` blocks that can handle problems inside your code. If JavaScript encounters an error when executing code in a `try/catch` block, it will jump down and execute the `catch` portion instead of stopping the entire script. If no error occurs, all of the `try` will be executed and none of the `catch`.

For example, the following `try/catch` block will execute any code that replaces `your_code_here`. If an error occurs executing that code, the error message followed by the string ": happened when loading the script" will be written to the document:

```
try {
    your_code_here
} catch (err) {
  document.write(err.message + ": happened when loading the script");
}
```

### Throw Your Own Errors

You can also throw your own errors using a `throw` statement. The following code illustrates how to add throws to a function to `throw` an error, even if a script error does not occur:

```
01 <script>
02   function sqrRoot(x) {
03     try {
04       if(x=="")     throw "Can't Square Root Nothing";
05       if(isNaN(x)) throw "Can't Square Root Strings";
06       if(x<0)       throw "Sorry No Imagination";
07       return "sqrt("+x+") = " + Math.sqrt(x);
08     } catch(err){
09       return err;
10     }
```

```
11    }
12    function writeIt(){
13      document.write(sqrRoot("four") + "<br>");
14      document.write(sqrRoot("") + "<br>");
15      document.write(sqrRoot("4") + "<br>");
16      document.write(sqrRoot("-4") + "<br>");
17    }
18 </script>
```

The function `sqrRoot()` accepts a single argument x. It then tests x to verify that it is a positive number and returns a string with the square root of x. If x is not a positive number, the appropriate error is thrown and returned to `writeIt()`.

### Using `finally`

Another valuable tool in exception handling is the `finally` keyword. A `finally` keyword can be added to the end of a `try/catch` block. After the `try/catch` blocks are executed, the `finally` block is always executed. It doesn't matter if an error occurs and is caught or if the `try` block is fully executed.

Following is an example of using a `finally` block inside a web page:

```
function testTryCatch(value){
  try {
    if (value < 0){
      throw "too small";
    } else if (value > 10){
      throw "too big";
    }
    your_code_here
  } catch (err) {
    document.write("The number was " + err.message");
  } finally {
    document.write("This is always written.");
  }
}
```

# Summary

In this lesson, you learned the basics of adding jQuery and JavaScript to web pages. The basic data types that are used in JavaScript and, consequently, jQuery were described. You learned some of the basic syntax of applying conditional logic to JavaScript applications. You also learned how to compartmentalize your JavaScript applications into functions that can be reused in other locations. Finally, you learned some ways to handle JavaScript errors in your script before the browser receives an exception.

# Q&A

**Q. When should you use a regular expression in string operations?**

**A.** That depends on your understanding of regular expressions. Those who use regular expressions frequently and understand the syntax well would almost always rather use a regular expression because they are so versatile. If you are not familiar with regular expressions, it takes time to figure out the syntax, and so you will want to use them only when you need to. The bottom line is that if you need to manipulate strings frequently, it is absolutely worth it to learn regular expressions.

**Q. Can I load more than one version of jQuery at a time?**

**A.** Sure, but there really isn't a valid reason to do that. The one that gets loaded last will overwrite the functionality of the previous one. Any functions from the first one that were not overwritten may be completely unpredictable because of the mismatch in libraries. The best bet is to develop and test against a specific version and update to a newer version only when there is added functionality that you want to add to your web page.

# Workshop

The workshop consists of a set of questions and answers designed to solidify your understanding of the material covered in this lesson. Try to answer the questions before looking at the answers.

## Quiz

1. What is the difference between `==` and `===` in JavaScript?
2. What is the difference between the `break` and `continue` keywords?
3. When should you use a `finally` block?
4. What is the resulting value when you add a string "1" to a number 1, ("1"+1)?

## Quiz Answers

1. `==` compares only the relative value; `===` compares the value and the type.
2. `break` will stop executing the loop entirely, whereas `continue` will only stop executing the current iteration and then move on to the next.
3. When you have code that needs to be executed even if a problem occurs in the `try` block.
4. The string "11" because the number is converted to a string and then concatenated.

## Exercises

1. Open js_functions.html and modify it to create a table instead of a list. You will need to add code to the `writeIt()` function that writes the `<table>` open tag before iterating through the planets and then the closing tag after iterating through the planets. Then modify the `makeListItem()` function to return a string in the form of:

   ```
   <tr><td>planent</td><td>moon</td></tr>
   ```

2. Modify if_logic.html to include some additional times with different messages and images. For example, between 8 and 9, you could add the message "go to work" with a car icon, and between 5 and 6, you could add the message "time to go home" with a home icon. You will need to add some additional cases to the switch statement and set the `timeOfDay` value accordingly.

**Excerpt from "Learning to Program" by Steven Foote (Addison-Wesley Professional, 2014)**

# 6

# Regular Expressions

Note

**Project:** Validate a phone number using regular expressions.

*Most introductory programming books don't include an entire chapter on regular expressions because regular expressions can be intimidating. As a result, many programmers choose to never learn regular expressions. Although other tools can do much of what regular expressions can do, they don't do the job as well. You can drive nails with the backside of a screwdriver, for example, but using a hammer is so much easier. For certain jobs, nothing but regular expressions work. Becoming comfortable with regular expressions early will benefit you as long as you program.*

*Having said that, I should mention that regular expressions are hard. They are terse and can be quite confusing. They are difficult to document and difficult to read, and making mistakes is easy. Still, they are remarkably useful.*

## Ctrl+F on Steroids: Looking for Patterns

The last chapter briefly introduced you to regular expressions, but you are probably still wondering what they are and how they are used. Regular expressions are a lot like the Find (keyboard shortcut: Ctrl+F) feature in Word, Chrome, and Sublime Text. Regular expressions search a string for a given set of characters, but they are far more powerful. Imagine that you have a 300-page document and you want to find all the places where a year is mentioned (for example, `2014`). With Find, you have to search for every single year individually—first you'd search for `1900`, then `1901`, then `1902`, and so on before you'd give up as it would take way too long. With regular expressions, you can search for patterns, and a year is a very simple pattern: a sequence of four digits from `0` to `9`. See Listing 6.1.

Listing 6.1 **Searching for Years with Regular Expressions**

```
var yearPattern = /[0-9]{4}/;
```

As you can see, the syntax for regular expressions is a bit strange, but if you look closely, you can see all the relevant pieces. The digits from `0` to `9` are represented by `[0-9]`, and the requirement that there are four of them is represented by `{4}`. So `yearPattern` is really saying, "Any number from `0` to `9`, repeated four times." A perfect regular expression will match everything you want to find in a string and not match anything you don't want to find. For example, `yearPattern` will match any four-digit sequence of numbers, so it will match all the four-digit years (good), but it will also match phone numbers like `800-555-3456` (bad) because there is a sequence of four digits in phone numbers. In this chapter, you learn the tools you need to turn `yearPattern` into a much better regular expression.

## Using Regular Expressions in JavaScript

Regular expressions do nothing without a string to test them on. Let's start with a simple string and a simple regular expression. Open the Chrome Dev Tools on any page, and create a string with your name in it; then create a regular expression with your name in it. See Listing 6.2.

Listing 6.2 **My Name As a Regular Expression**

```
> var myName = 'Steven Foote';
  "Steven Foote"
> var namePattern = /Steven/;
  /Steven/
```

Now you need to test whether the string matches the regular expression. You can do this in a few different ways, each with a slightly different purpose and output.

- `test` returns a Boolean, `true` if the string matches and `false` if it doesn't.
- `exec` either returns an array of the first part of the string that matches or else returns nothing.
- `match` either returns the part (or parts) of the string that match in an array or else returns nothing. `match` is different because it is a method on the string and the regular expression is the argument (see Listing 6.3).

Listing 6.3 **Testing Our Regular Expression/String Combo All Three Ways (In the Console)**

```
> namePattern.test(myName);
  true
> namePattern.exec(myName);
  ["Steven"]
> myName.match(namePattern);
  ["Steven"]
```

## Repetition

The regular expression `namePattern` isn't much of a pattern at all; it's no different than using Find. What if I want to match `Steven` or `Steve`? Listing 6.4 shows how inflexible `namePattern` is.

Listing 6.4 **Trying to Match `Steve`**

```
> var myNickname = 'Steve Foote';
  "Steve Foote"
> namePattern.test(myNickname);
  false
```

Dang. My regular expression isn't flexible enough for even my nickname. We can fix that—but be warned, this is where regular expressions start to look weird.

### ?

Leave behind any preconceptions you might have about what the question mark means. In the land of regular expressions, unassuming characters can leave behind the roles they play in normal life and take on new, meaningful powers. The simple `?` has the power to solve our nickname problem. See Listing 6.5.

Listing 6.5 **The Powerful `?`**

```
> namePattern = /Steven?/;
  /Steven?/
> namePattern.test(myNickname);
  true
> namePattern.text(myName);
  true
```

Magic! The `?` tells the regular expression that the character before the `?` should appear once or not at all. In other words, the new `namePattern` says that the `n` is optional.

### +

Right now you're probably thinking "Wooooooooow! Regular expressions are amazing!" Well, maybe you're not quite that excited about it, but guess what? Regular expressions can handle it, no matter how excited you are. In Regular Expression Land, the `+` tells the regular expression to match the preceding character one or more times. See Listing 6.6.

Listing 6.6 **Regular Expressions Can Handle All Levels of Excitement**

```
> var excitementPattern = /Wo+w/;
  /Wo+w/
> var notYetExcited = 'Wow... Regular Expressions are weird :/';
  "Wow... Regular Expressions are weird :/"
> excitementPattern.exec(notYetExcited);
  ["Wow"]
> var startingToGetExcited = 'Wooow. Regular Expressions are... pretty cool.';
  "Wooow. Regular Expressions are... hard."
> excitementPattern.exec(startingToGetExcited);
  ["Wooow"]
> var totallyLovingRegEx = 'Wooooooooooww! Regular Expressions are awesome!!!';
  "Wooooooooooww! Regular Expressions are awesome!!!"
> excitementPattern.exec(totallyLovingRegEx);
  ["Wooooooooooww"]
> var excitedButBadAtSpelling = 'Www! Relugar Expresions, I <3 u!';
  "Wwwwww! Relugar Expresions, I <3 u!"
> excitementPattern.exec(excitedButBadAtSpelling);
  null
```

*

The star (sometimes called the Kleene star, after its creator, Stephen Kleene) tells the regular expression to match the preceding character zero or more times. It's truly a regular expression rock star. We can make our `excitementPattern` regular expression flexible enough to even handle bad spelling. See Listing 6.7.

Listing 6.7 **A Regular Expression Rock Star**

```
> excitementPattern = /Wo*w/;
  /Wo*w/
> var excitedButBadAtSpelling = 'Www! Relugar Expresions, I <3 u!';
  "Wwwwww! Relugar Expresions, I <3 u!"
> excitementPattern.test(excitedButBadAtSpelling);
  true
```

## Special Characters and Escaping

Now that you've met some of the super-powerful characters of Regular Expression Land, you might want to create a regular expression that matches strings that contain those amazing characters, just to show what a huge fan you are. Give it a shot and check out Listing 6.8.

Listing 6.8 **Looking for Stars, Plus More**

```
> var starPattern = /*/;
  SyntaxError: Unexpected token ILLEGAL
> var plusPattern = /+/;
  SyntaxError: Invalid regular expression: /+/: Nothing to repeat
```

Well, that didn't go well. The first `SyntaxError` is pretty mysterious, but the second error is a bit more helpful. Our `plusPattern` regular expression is invalid because the + has no preceding character that should be repeated. But we're trying to match the +, not repeat some other character. We need to tell the regular expression that we want to match a literal + instead of giving + super powers. Do you remember when we ran into this problem in Chapter 1, "Hello World! Writing Your First Program," with the apostrophe inside a string? We needed an escape character then, as we do now. Regular expressions also use the backslash for escaping. See Listing 6.9.

Listing 6.9 **Looking for Stars and Actually Finding Them**

```
> var starPattern = /\*/;
  /\*/
> var aStar = 'Kleene, you\'re a *';
  "Kleene, you're a *."
> starPattern.test(aStar);
  true
```

## `{1,10}`: Make Your Own Super Powers

The special characters you just learned about are some of the most commonly used characters in regular expressions, but there's more. If `?`, +, and * aren't specific enough, you can define your own lengths using curly braces. Remember the `yearPattern` example that matches a sequence of exactly four digits? You can use the curly braces in three ways:

- `{n}` matches the preceding character exactly *n* times.
- `{n,}` matches the preceding character at least *n* times.
- `{m,n}` matches the preceding character at least *m* times, but no more than *n* times.

You can actually use the curly brace syntax to get the same result as `?`, +, and *. `?` is the same as `{0,1}`, + is the same as `{1,}`, and * is the same as `{0,}`. We will be using the curly braces extensively in the project in this chapter.

## Match Anything, Period

Sometimes you want a pattern that matches anything. For instance, we could make our `excitementPattern` match only strings that contain some form of `Wow` *and* contain an

exclamation point (!), but there may be some stuff between the Wow and the ! that we don't really care about. Regular expressions have another super-powerful special character to handle that situation: The period, usually called the dot (as in *dot-com*), matches any character. It's like the regular expression wildcard. You can combine the dot and the star to match zero or more characters, regardless of what those characters are. See Listing 6.10.

Listing 6.10 **Matching Anything**

```
> excitementPattern = /Wo*w.*!/;
  /Wo*w.*!/
> excitementPattern.test(notYetExcited);
  false
> excitementPattern.test(excitedButBadAtSpelling);
  true
```

As our patterns get more complex, it can be useful to visualize what is happening. www.regexper.com/ is an amazing online tool that creates a visualization for any regular expression. Figure 6.1 is the visualization for the `excitementPattern`, in the "railroad" diagram format. Your train starts at the dot on the left, and each of the squares is like a train station. Each time the train stops at a station, exactly one character from the string has to get off the train, and the characters have to get off the train in the correct order. The label on the station describes what kind of character is allowed to get off the train. If the train stops at a station and the next character in line is not allowed to get off, the train derails, which means the string does not fit the pattern defined in the regular expression.

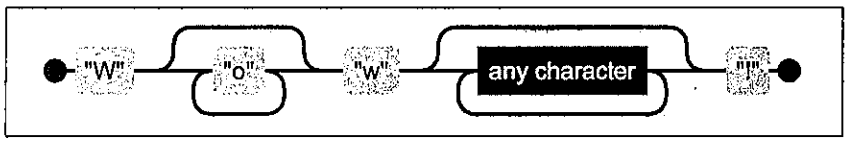



Figure 6.1 A visualization of /Wo*w.*!/

## Don't Be Greedy

Regular expressions are greedy—not the *Wall Street*/Gordon Gekko kind of greed, but they are greedy all the same. The regular expression greed means that a regular expression wants to match as much of a string as it can. For example, given the string `'Www! Regular Expressions, I <3 u!'`, the `excitementPattern` (`/Wo*w.*!/`) will match the entire string instead of just the `Www!` part. This is because the `.*!` tells the regular expression to match everything (including the `!`) until it finds the *last* `!`. Sometimes greed is good, but if you want your regular expression not to be greedy, there is hope. Add a `?` (wow, `?` sure has a lot of super powers) after the `*` or `+`, and its greed will go away.

# Understanding Brackets from [A-Za-z]

The powerful . is great when you want to match anything and everything, but sometimes "anything" is too broad. Using brackets, you can define exactly which characters you want to match. If I want to match my name whether or not the s is capitalized, I can use brackets to match either S or s. See Listing 6.11.

Listing 6.11 **Use Brackets to Handle Lowercase Names**

```
> lowerCaseName = 'steven foote';
  "steven foote"
> namePattern     // remember what the name pattern looks like
  /Steven?/
> namePattern.test(lowerCaseName);
  false
> namePattern = /[Ss]teven?/;
  /[Ss]teven?/
> namePattern.test(lowerCaseName);
  true
```

## Lists of Characters

The simplest way to use brackets is to list all the acceptable characters within the brackets. In the updated `namePattern`, the acceptable characters are S and s, so those two characters go inside the brackets. If you wanted to match double or single quotes, you could use `["']`. If you wanted to match any lowercase letter in the English alphabet, you could use `[abcdefghijklmnopqrstuvwxyz]`, and if you wanted to match any lowercase or uppercase letter in the English alphabet, you could use `[abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]`. Wow, that is ugly. There must be a better way.

## Ranges

There *is* a better way! With the `yearPattern`, we matched any digit from 0 to 9 using `[0-9]`, which is way easier than typing out `[0123456789]`, although they both do the same thing. Within brackets, a dash between two characters is called a range. Ranges are most commonly used for ranges of numbers and letters. Ranges on other characters can be quite confusing, and I don't recommend using them. You can match any upper- or lowercase letter in the English alphabet using `[a-zA-Z]`. That is much better (and you don't have to worry that you missed one of the letters). Ranges don't have to start with 0 or A, and they don't have to end with 9 or Z. If you want to match only letters in the second half of the alphabet, you can use `[n-z]`, for instance. You can also use a combination of individual characters and ranges: `[12357a-z]`.

A time might come when you want to include a dash as one of the characters in your list, but the dash has super powers within the brackets, so you need to use ... that's right, you need to use an escape character. The pattern `/[a-z\-]/` matches a dash or any lowercase letter. You can also use

brackets and repetition together. The `yearPattern` uses brackets to say that it wants to match digits `0` to `9`; then it uses repetition to say that it wants to match those digits exactly four times.

### Negation

If you want to match anything but a group of characters, regular expressions have got you covered with another character with super powers. Add a caret (^) at the beginning of the group to match anything but the characters in the group. Let's say that I am adamant about being called Steve instead of Steven. In fact, I would take any character at the end of my name, as long as it's not `n` (I actually prefer to be called Steven, but for the sake of the example, let's pretend). My new `namePattern` could look like Listing 6.12.

Listing 6.12 **Don't Call Me Steven**

```
> namePattern = /Steve[^n]/;
  /Steve[^n]/
> namePattern.test('Steve Foote');
  true
> namePattern.test('Steveq Foote');
  true
> namePattern.test('Steven Foote');
  false
```

## A Pattern for Phone Numbers

At this point, you have learned enough to get started on the project of validating phone numbers. First, we need to think of the different valid phone number formats. Here are a few possibilities:

- 1-555-867-5309
- 555-867-5309
- (555)867-5309

We can quickly experiment with these three formats using an online regular expression tool called regexpal: http://regexpal.com/. In the upper box, we enter our regular expression (no need for the / when using this tool), and in the lower box, we enter the strings we want to match, one per line. Add each of the three formats to the lower box. You should also add some strings that should not match the pattern, such as `'Regular Expressions are great'` and `'-0-1-2-3-4-5-6'`. These are clearly not phone numbers, so if our pattern matches them, we need to do some work.

In each of these formats, we see some digits and some dashes. In one of these formats, we see parentheses. The formats have between 12 and 14 characters. We can turn that into a rudimentary regular expression using brackets and repetition. Listing 6.13 shows the regular expression, and Figure 6.2 shows a visual representation of the regular expression.

Listing 6.13 **Phone Number Regex, Round 1**

```
var phoneNumberPattern = /[0-9\-\(\)]{12,14}/;
```

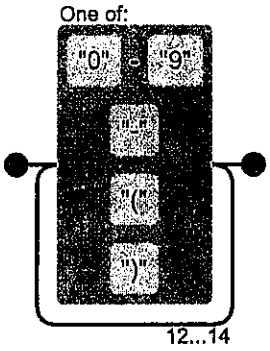



Figure 6.2 A visualization of `phoneNumberPattern`

Our first attempt is great; it matches all three formats. But it also matches one of the non–phone number strings (see Figure 6.3). It's good, but we can do better.

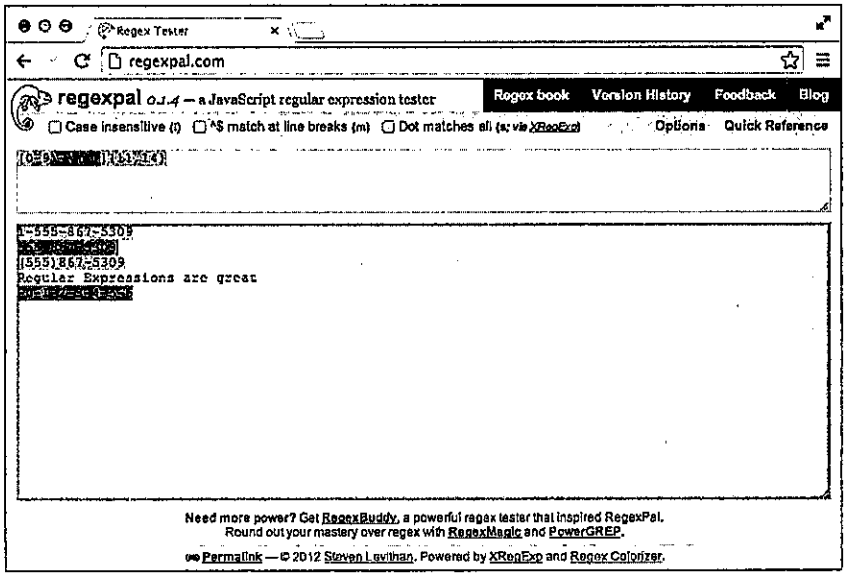



Figure 6.3 A valiant first attempt, but we have a way to go.

To make our pattern even better, we need to break down the phone number into its parts: 1) An optional prefix: `1-`. 2) An optional opening parentheses. 3) A three-digit area code. 4) Either a dash or a closing parentheses. 5) Three digits. 6) A dash. 7) Four digits. Each of these parts is a relatively simple pattern:

1. `1?-?`
2. `\(?`
3. `[0-9]{3}`
4. `[\-\)]`
5. `[0-9]{3}`
6. `-`
7. [0-9]{4}

Putting them all together (see Listing 6.14), we get a not-so-simple pattern, but we know what each part does, so it's not impossible to understand. With the help of the railroad diagram in Figure 6.4, things get even easier. The real test, though, is whether our new pattern matches everything it should match and rejects everything it should not match (see Figure 6.5).

Listing 6.14 **Phone Number Regex, Round 2**

```
var phoneNumberPattern = /1?-?\(?[0-9]{3}[\-\)][0-9]{3}-[0-9]{4}/;
```

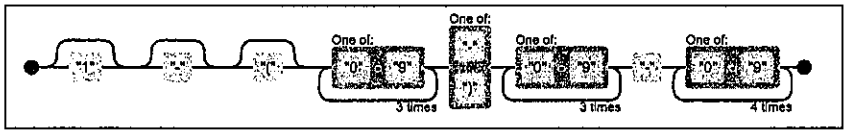



Figure 6.4 A visualization of a better phone number pattern

The phone number pattern is looking great. We will make it even better as we learn more about regular expression features.

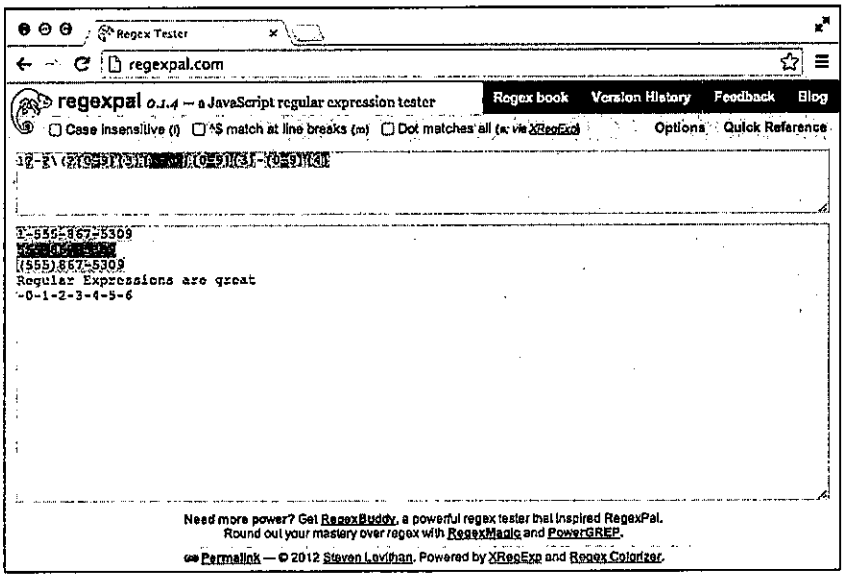



Figure 6.5 A much better phone number pattern

# I Need My \s

We have improved our patterns a lot from the first regular expression we wrote (`/Steven/`), and we haven't even gotten into one of the most prolific features regular expressions have to offer: the backslash. So far, we've used the backslash as an escape character, a way to take the super powers away from a special character. But the backslash doesn't just take; the backslash can give super powers to ordinary characters, too.

## Shortcuts for Brackets

Most commonly, the backslash turns letters into groups. For instance, `\s` matches any type of whitespace character, such as a space, a tab, and a newline (among others). The backslash can also represent a character that cannot be used literally in a regular expression. For instance, a regular expression generally has to be all on one line, so to represent a new line, you need to use `\n` because you can't just press Enter. The following table describes all the characters that take on special meaning with the backslash.

| Symbol | Meaning |
|---|---|
| `\b` | A word boundary. This matches a place where a word character (see `\w`) is next to a nonword character (see `\W`). |
| `\B` | A nonword boundary. This matches a place where two word characters are next to each other or two nonword characters are next to each other. |

| Symbol | Meaning |
|---|---|
| `\d` | A digit. `\d` is shorthand for `[0-9]`. |
| `\D` | Anything but a digit. `\D` is shorthand for `[^0-9]`. |
| `\n` | A newline character. |
| `\r` | A carriage return character. Used on Windows to create a new line. |
| `\s` | A whitespace character. Whitespace characters include a space (the character you get when you press the spacebar), a tab, a newline (`\n`), a carriage return (`\r`), and a few others. |
| `\t` | A tab character. |
| `\w` | A word character, which is any alphanumeric character or underscore. `\w` is shorthand for `[0-9a-zA-Z_]`. |
| `\W` | A nonword character, equivalent to `[^0-9a-zA-Z_]`. |

With this knowledge, we can make our phone number pattern a little more readable. Our pattern currently repeats `[0-9]` three times. We can replace `[0-9]` with `\d` for a slightly more concise regular expression without changing what it does (see Listing 6.15).

Listing 6.15 **Phone Number Pattern, Round 3**

```
var phoneNumberPattern = /1?-?\(?\d{3}[\-\)]\d{3}-\d{4}/;
```

Let's add some code to our kittenbook project. In `prompt.js`, we'll ask users for their phone numbers instead of their names. Then we'll validate the phone numbers using our regular expression (see Listing 6.16). As you learned in Chapter 2, "How Software Works," we could send an SMS to that number to further our validation. However, that is beyond the scope of this book. Let's just stick to validation via regular expressions for now.

Listing 6.16 **`prompt.js` Now Asks for and Validates Phone Numbers**

```
// Get the user's name.
var userName = prompt('Hello, what\'s your name?');
// Get the user's phone number.
var phoneNumber = prompt('Hello ' + userName + ', what\'s your phone number?');
// Create the phone number pattern.
var phoneNumberPattern = /1?-?\(?\d{3}[\-\)]\d{3}-\d{4}/;
// Create a variable to store the output.
var output = '<h1>Hello, ' + userName + '!</h1>';

// Is the phone number valid?
if (phoneNumberPattern.test(phoneNumber)) {

  // Yes, the phone number is valid! Add the success message to the output.
```

```
  output = output + '<p>' + kbValues.projectName + ' ' + kbValues.versionNumber +
           ' viewed on: ' + kbValues.currentTime + '</p>';

} else {
  // No, the phone number is not valid. Tell the user about the problem.
  output = output + '<h2>That phone number is invalid: ' + phoneNumber;
}
// Insert the output into the web page.
document.body.innerHTML = output;
```

## Limitations

Some of these normal characters endowed with super powers are absolutely necessary because they represent a single character (such as `\n` and `\t`) that cannot be represented in any other way in a regular expression. However, some of these characters are just shortcuts to allow you to write less code (such as `\w` and `\s`). The shortcuts are useful, as long as you recognize their limitations. The `\w` matches only the letters from the English alphabet. Throw in just a little bit of Spanish flair to your string (as with `olé`), and your regular expression will fail. Also, `\s` matches a lot of space characters that you might not actually want to match. In both cases, writing out the exact set of characters you want to match (within brackets) is sometimes necessary. The shortcuts are useful—until they're not.

### (?:Groups)

I will admit that, up to this point, regular expressions might seem a little boring. Although I do think it's pretty cool to be able to create a pattern that can match any phone number in a whole bunch of different formats, groups (and especially capturing groups) is what made me really love regular expressions. Groups are like subpatterns within the larger regular expression, and the super characters can operate on the entire group together. A few examples should help make this clearer. Let's you have a dog named fifi, and you want to create a regular expression that matches any string with your dog's name in it. To complicate matters, let's say you sometimes call your dog "fi" for short. You need a group, which is a set of characters surrounded by parentheses. A noncapturing group, which is the type we want to use to find `fifi`, starts with `?:`. See Listing 6.17.

Listing 6.17 **Finding** `fifi`

```
> var fifiPattern = /(?:fi){1,2}/;
  /(?:fi){1,2}/
> var fifi = 'I have a dog named fifi';
  "I have a dog named fifi"
> var fi = 'Sometimes I call my dog fi';
  fi = "Sometimes I call my dog fi"
> fifiPattern.test(fifi);
  true
```

```
> fifiPattern.test(fi);
  true
```

What about the `namePattern` to match `'Steven'` and `'Steve'`—what if we want to match `'Stephen'`, too? We can make it happen with a group, but we'll also need to introduce another super character: `|`. The `|`, or pipe (found on the Backslash key, between the Backspace and Enter keys), means "or" in regular expressions. The pipe splits the regular expression and matches either what is on the left or what is on the right. For example, `/apple|broccoli/` matches `'apple juice'` and `'baked broccoli'`, but not `'carrot cake'`. We can use a pipe inside a group to match either `v` for `'Steven'` or `ph` for `'Stephen'`. See Listing 6.18.

Listing 6.18 **Searching for `'Stephen'`**

```
> var namePattern = /Ste(?:v|ph)en?/;
  /Ste(?:v|ph)en?/
> var myName = 'Stephen';
  "Stephen"
> namePattern.test(myName);
  true
> myName = 'Steven';
  "Steven"
> namePattern.test(myName);
  true
```

As a final example, let's consider the prefix in our `phoneNumberPattern`. It's almost right, but it has a slight problem. When we divided the phone number into parts, the first part was an optional prefix of `1-`, which we achieved with `1?-?`. The problem is that `1?-?` matches phone numbers that look like `-877-555-1234` because the pattern matches the `1` zero or one times, and then matches the `-` zero or one times. We really want to match them together zero or one times, and we can do that with a group. See Listing 6.19.

Listing 6.19 **Fixing the Prefix**

```
var phoneNumberPattern = /(?:1-)?\(?\d{3}[\-\)]\d{3}-\d{4}/;
```

That's much better. Another point to note in the `phoneNumberPattern` is the backslashes in front of the nongroup parentheses. Now that you know that parentheses are special characters, those backslashes should make a little more sense.

## (Capture)

Capturing groups are (for me, anyway) the most interesting and useful feature of regular expressions. They allow you to extract data from your matched string. For instance, we can use capturing on the `phoneNumberPattern` to extract just the area code from the string. Capturing groups are surrounded by just parentheses (with no special characters such as `?:` at

the beginning). You need to use `match` or `exec` to be able to access what the group captured. See Listing 6.20.

Listing 6.20 **Capture the Area Code**

```
var phoneNumberPattern = /(?:1-)?\(?(\d{3})[\-\)]\d{3}-\d{4}/;
```

Now that we have access to the area code, we can create a more personalized message for our users. We'll assume that the user lives in the location related to the area code (cellphones have kind of ruined this assumption, but we'll go with it anyway). You can add an `areaCodes` object to the `kbValues` object in `values.js`. See Listing 6.21.

Listing 6.21 **Add an `areaCodes` Object to `kbValues`**

```
var kbValues = {
  projectName: 'kittenbook',
  versionNumber: '0.0.1',
  areaCodes: {
    '408': 'Silicon Valley',
    '702': 'Las Vegas',
    '801': 'Northern Utah',
    '765': 'West Lafayette',
    '901': 'Memphis',
    '507': 'Rochester, MN'
  }
};
```

You can add as many area codes as you want to the `areaCodes` object. Now you can update `prompt.js` to use the area code from the user's phone number to create a more personalized greeting. The first step is to update the regular expression to include the capturing group for the area code (see Listing 6.20). Then you need to capture the matches using `exec` or `match`, which will return an array with the area code in index position 1. See Listing 6.22.

Listing 6.22 **Capture the Area Code**

```
// Create the phone number pattern.
var phoneNumberPattern = /(?:1-)?\(?(\d{3})[\-\)]\d{3}-\d{4}/;
// Get matches from phoneNumber
var phoneMatches = phoneNumberPattern.exec(phoneNumber);
// If the phone number is 901-555-5309, then phoneMatches will be
// ['901-555-5309', '901']
var areaCode = phoneMatches[1];
```

Finally, you need to add the personalized message to the output. This brings us to an interesting problem. You know how to access the different attributes of an object using the object

name, then a dot, and then the attribute name, as in `kbValues.projectName`. So we can access the area codes object with `kbValues.areaCodes`, and we have the area code we want stored in the `areaCode` variable. But how can we access the value related to that area code? We could try `kbValues.areaCodes.areaCode`, but that would look for an attribute whose name is `areaCode` inside the `areaCodes` object. In JavaScript, you can use the bracket syntax to access attributes of an object (see Listing 6.23). It looks similar to the syntax for accessing items in an array. Listing 6.24 shows how to use the bracket syntax to get the location based on area code.

Listing 6.23 **Accessing Object Attributes with the Bracket Syntax**

```
> var states = {'AL': 'Alabama',
                'AK': 'Alaska',
                'AZ': 'Arizona',
                'AK': 'Arkansas'};
> states.AL;
  'Alabama'
> var stateName = 'AL';
  'AL'
> states.stateName;  // This won't work :(
  undefined
> states[stateName];
  'Alabama'
> states['AL'];
  'Alabama'
```

Listing 6.24 **Getting the Location Based on the `areaCode`**

```
// Get the location using bracket syntax
var userLocation = kbValues.areaCodes[areaCode];
```

Now you can add the `location` to the output in whatever way you want. Maybe you could ask how the weather in `location` is these days. Regardless, your users are going to be impressed that you figured out where they live based on their phone number. They would be even more impressed if they knew all the regular expression magic you had to do to figure out their location.

## Capture the Tag

Now that you've seen a bit about how using regular expressions and capturing groups work, it's time to try some out on your own. Open the Chrome Dev Tools on any page (I recommend the Mozilla Developer Network, but any page will do). Find an HTML tag in the Elements tab, copy the HTML of that element (see Figure 6.6), and save the string to a variable in the console (I recommend an element that has no children so that you don't have to deal with newlines when saving the string to a variable). Now write a regular expression that will capture the tag

name (for example, the tag name of <h2> is h2). You should be able to use the same regular expression to capture the tag name of any HTML tag you copy.

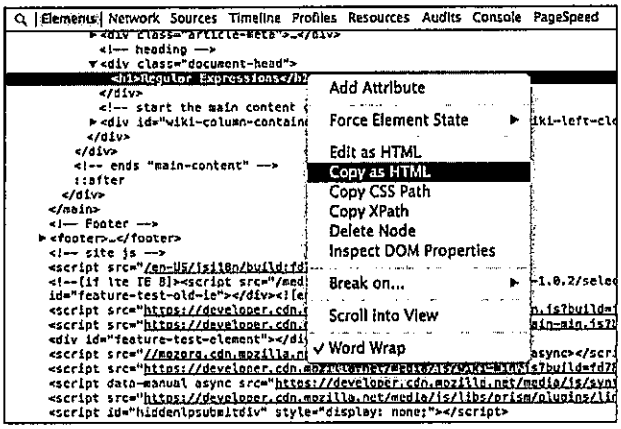



Figure 6.6 Copy the HTML of a given tag.

## Advanced Find and Replace

To see another way capturing groups can be useful, let's think back to Listing 5.9 of the last chapter. That was the one where we updated the age attribute of the `author` string with the long, complex code. We can improve that long, complex code a lot by using a regular expression with a capturing group and also using advanced find and replace (see Listing 6.25).

Listing 6.25 **Improving the String Data Structure with a Capturing Group**

```
var author = 'firstName=Steven&lastName=Foote&age=27&favoriteFoods=waffles,Thai
curry';

// Use the String's built-in replace method.
author.replace(/(age=)(\d+)/, function(fullMatch, group1, group2) {
  /**
    * fullMatch contains "age=27"
    * group1 contains "age="
    * group2 contains "27"
    */

  // Whatever we return will replace the entire match in the original string
  // Add 1 to group2, then put the two groups back together.
  // parseInt turns the string "27" into the number 27, so we can add them together.
  // Without parseInt, "27" + 1 becomes "271", not 28.
  return group1 + (parseInt(group2, 10) + 1);
```

```
});
// author is now 'firstName=Steven&lastName=Foote&age=28&favoriteFoods=waffles,Thai
curry'
```

## The Beginning and the End (of a Line)

Find and replace with regular expressions is pretty amazing, but one part of the previous regular expression is not quite right. The regular expression matches `'age='` and then some numbers. What if the author string also contained `'page=23'`? Then our regular expression would match `'age=27'` and `'page=23'`. We could change our regular expression to be `/(&age=)(\d+)/`, but we would still have a problem: What if `'age'` is at the beginning of the string and there is no preceding `&`? Fortunately, regular expressions have a way of identifying the beginning (and end) of a line. The beginning of a line is represented by a `^` (remember, the `^` has a different meaning inside brackets—regular expressions sure can be confusing), and the end of the line is represented by a `$`. We can change our find-and-replace regular expression to match either `&` or the beginning of the line, then `'age': /((?:^|&)age=)(\d+)/`.

# Flags

Regular expressions have a few default behaviors that you can turn on or off using flags. These default behaviors usually work just fine, but when they don't, it's nice to be able to turn them off.

## Global

By default, as soon as a regular expression finds a match within a string, it stops looking. The global flag tells the regular expression to find all matches in the string, not just the first. To set the global flag, you place a `g` outside the closing forward slash of your regular expression: `/[0-9]{4}/g` finds all years in a string, not just the first one.

## Ignore Case

Regular expressions are case sensitive by default, but sometimes you don't care whether text is upper or lower case. You can keep your regular expressions from being case sensitive using the ignore case flag (set with an `i`, so `/ste(?:ph|v)en/gi` finds every instance of `'Steven'`, `'steven'`, `'Stephen'`, and `'stephen'` in a string). Note how the global flag and the ignore case flag can be (but don't have to be) used together.

### Multiline

By default, `^` and `$` match only the beginning of the string and the end of the string, respectively. In other words, if your string contains multiple lines, the beginning and ends of the lines are not actually matched by `^` and `$` unless you turn on the multiline flag. The multiline flag is set with an `m`. `/.*\.html$/gm` matches every line in a string that ends with `'.html'`.

## When Will You Ever Use Regex?

By now you are probably thinking, "I never want to see another regular expression again. My head hurts." Or maybe you're thinking, "Wow! I <3 regular expressions!" I'm guessing it's not the latter. Either way, you've learned a lot about regular expressions, and you might be wondering about when and how you will actually use them. The following examples are a few of the ways that I have actually used regular expressions to make my work easier (for real, regular expressions can actually make life easier, not harder).

### `grep`

In Chapter 3, "Getting to Know Your Computer," you learned about the command-line tool `grep`. At the time, I made `grep` seem like Find for searching multiple files at once. But `grep` is actually all about regular expressions (in fact, *grep* stands for global regular expression print). Using `grep`, you can search for patterns within multiple files at once. When working on a project that has a lot of files (LinkedIn has a *lot* of files), `grep` is a necessity. For example, I can search all my CSS files for places where I am using `float: left;` or `float: right;` using one `grep` command.

### Code Refactoring

I am writing this book using HTML, and I have used regular expressions (and especially advanced find and replace) to help me work faster. As I start a new chapter, I copy the portion of my outline that relates to that chapter into a new HTML file. The items in the outline use HTML lists (`<ul>` and `<li>` tags), but these items become headers in the actual chapter (`<h1>`, `<h2>`, and so on). I could switch each of these tags by hand, but instead I write a couple regular expressions. As shown in Figure 6.7, Sublime Text, Vim, and many other editors support regular expressions (and capturing groups) in their find-and-replace features.

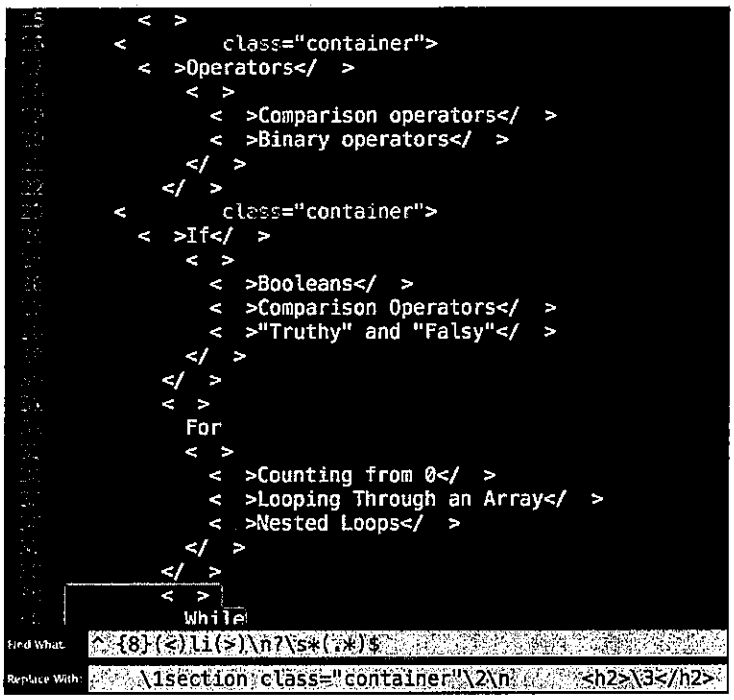



Figure 6.7 Change list items to headers with regular expressions

## Validation

In this chapter, you have written a pretty good phone number validation (for U.S. phone numbers). Nearly every time I have to do validation, I use regular expressions. Emails are especially difficult to validate; do a search for "email regular expression," and you will see what I mean. Again, just because a phone number or email address matches a regular expression does not mean that it is valid, but regular expressions can tell you whether the string at least *looks* reasonable.

## Data Extraction

My very first programming project was to use regular expressions to extract data about books from a very large (1MB) text file. That project could be the reason I am particularly fond of capturing groups. When I saw regular expressions turn an unusable 30,000-page report into a useful spreadsheet, I was hooked.

## Summing Up

You have learned a lot in this chapter. What once looked like random characters (ahem, `/(?:1-)?\(?(\d{3})[\-\)]\d{3}-\d{4}/`, ahem) now makes sense to you. Sure, it might take some time to figure out what it means (and I think it always will), but you can figure it out. Regular expressions are a great tool, but remember that just because you're holding a hammer doesn't make everything a nail. In this chapter, you learned about:

- The regular expression syntax
- Quantifiers such as `?`, `+`, and `*`
- Character sets in brackets
- Groups and capturing groups
- Advanced find and replace and other applications of regular expressions

In the next chapter, you will learn about:

- Operators
- Programming with flow control
- How to use `if`, `for`, `while`, and `switch` to control flow
- Another way to code defensively with `try`
- Triggers and events