

Making Everything Easier!™

PHP, MySQL, JavaScript® & HTML5

ALL-IN-ONE

FOR
DUMMIES®
A Wiley Brand

**7 BOOKS
IN 1**

- Web Technologies
- HTML and CSS
- JavaScript
- PHP
- MySQL
- Web Applications
- PHP and Templates

Steven Suehring
Janet Valade



Chapter 1: Creating a Basic Page with HTML

In This Chapter

- ✓ Getting the 411 on HTML and web pages
- ✓ Putting HTML tags into the correct section
- ✓ Integrating images and links into your page
- ✓ Ensuring that your HTML is valid

HyperText Markup Language (HTML) is the language of the web. When you go to a web page in your web browser such as Internet Explorer, Firefox, or Safari, the browser downloads and displays HTML.

At its heart, HTML is just a document, much the same as a document you'd make in a word processor. A program like Microsoft Word is used to view word processor documents because it knows how to read and display them. Likewise, when it comes to the web, the web browser is the program that knows how to read and display documents created with HTML.

Word processor documents can be created and read with a single program. On the other hand, HTML documents need different programs for creation and reading; you can't create HTML documents with a browser. You create HTML documents using a program called an editor. This editor can be as simple as the Notepad program that comes with Microsoft Windows or as complex as Eclipse or Microsoft Visual Studio. You can typically use the same program to create HTML documents that you use to create PHP programs.

This chapter describes HTML documents and shows how to build an HTML page that you can view through a web browser using the most current version, HTML5.

Understanding the HTML Building Blocks

HTML documents being just documents, they can be stored on any computer. For instance, an HTML document can be stored in the Documents folder on your computer. However, you'd be the only one who could view that HTML document on your computer. To solve that problem, web documents or pages are typically stored on a computer with more resources,



known as a web server. Storing the document on a web server enables other people to view the document.

A web server is a computer that runs special software that knows how to send (or serve) web pages to multiple people at the same time.

HTML documents are set up in a specific order, with certain parts coming before others. They're structured like this so that the web browser knows how to read and display them. When you create an HTML document, it's expected that you'll follow this structure and set up your document so the browser can read it.

Document types

Web browsers can display several types of documents, not just HTML, so when creating a web document the first thing you do is tell the browser what type of document is coming. You declare the type of document with a special line of HTML at the top of the document.

Web browsers can usually read documents in many formats, including HTML, XML, XHTML, SVG, and others. Each of these documents lives by different rules and is set up differently. The document type tells the browser what rules to follow when displaying the document.



In technical terms, the document type is called the Document Type Declaration, or DTD for short.

In prior versions of HTML, developers needed to constantly copy and paste the document type into the document because it was both long and complicated. With the release of the latest version of HTML, called HTML5, the document type has been greatly simplified. The document type for HTML5 is

```
<!doctype html>
```

This will be the first line of every HTML document that you create, before anything else. Any time you need to display HTML, you include a document type, sometimes called a doctype.



You may be tempted to use `<!doctype html5>`, but there is no version number associated with the HTML5 document type. When the next version of HTML comes out, you won't have to go back and update all your document types to HTML6 (unless, of course, they change the document type definition again!).

You may see the other, older document types in your career as an HTML developer. They include:

```
HTML 4.01 Strict      <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML
4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
```

```
HTML 4.01 Transitional<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML
4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```

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

```
XHTML 1.1 DTD <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//
EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
```

Other document types exist as well, and most of them are similarly complex and difficult to remember. If you see these document types on a web page, you'll know that the page may use slightly different syntax to create its HTML document.

HTML documents are made up of letters and words enclosed in angle brackets, sometimes called less-than or greater-than signs:

< >

For example, here's the main element in an HTML document, also called the root element:

```
<html>
```

Typically, HTML elements have both opening and closing tags. Elements are closed with a front-slash in front of the element name. Seeing `<html>` in the document means that later on the document will have `</html>` to close that element. It is said that everything in between the opening `<html>` and closing `</html>` makes up the document and is wrapped inside of those elements.

Sections of an HTML Document

HTML documents use a specific structure. This structure enables the document to be read by a web browser. You'll now see the three main parts of an HTML document.



Up until now you may have been thinking of HTML as creating documents. What's the difference between an HTML document and an HTML page? Nothing. The two terms are interchangeable.

Before going into each section of the document, it'll be useful to see the whole thing, so without further delay, Listing 1-1 shows an entire HTML document.

Listing 1-1: A Basic Web Page

```
<!doctype html>
<html>
<head>
<title>My First Document</title>
</head>
<body>
<div>My Web Page</div>
</body>
</html>
```

If you view this document in a web browser, you receive a page that has a title in the browser's title bar or tab bar and text that states:

My Web Page

Later sections in the chapter explain how to enhance this page with more HTML elements and more text.

The root element

Though not a section of an HTML document, the root element is what wraps around the entire document, appearing as the first thing after the doctype and the last thing in the document.

The root element is opened with:

```
<html>
```

The root element is closed with:

```
</html>
```

The head section and title element

The head section of a document contains information about the document itself. The head section is opened with:

```
<head>
```

The head section is closed with:

```
</head>
```



The head section should not be confused with a header or menu on a page itself. The head section is a behind-the-scenes element of a page.

The head section can contain a lot of information about the page. This information includes things like the title of the page, the language of the page (English, Spanish, French, Swedish, and so on), whether the page contains style information or additional helper programs, and other such things common to the page.

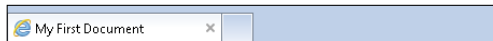


These descriptive elements in the head section are sometimes called *meta elements* because they're common to the entire page or describe the page itself.



You should always have a title element inside the head section. The title element is what shows up in the title bar of the web browser or as the title of the tab in the web browser, shown in Figure 1-1.

Figure 1-1:
The title element shows up in the tab or title bar of the web browser.



What makes a good title element?

Title elements should be descriptive of the page contents but not overly so. Frequently a title tag may have the site name along with something descriptive about the page itself. For example, www.braingia.org has “Steve Suehring – Official Site and Blog,” and then navigating to a given page, say the Books page, results in the title changing to “Books | Steve Suehring – Official Site and Blog.” The title is

therefore both descriptive of the page as well as the site.

Page titles are used by search engines like Google as one factor to help determine whether your page is relevant, therefore placing it higher in the search results. Google, for example, displays up to 66 characters of the title tag. So keeping the title short but sweet is key.

The body section

The body section is the heart of a web page. It's where you place all the text and images for the page. Essentially everything that you see when you view a page (with the exception of the title) is found within the body section.

The body section opens with:

```
<body>
```

The body section closes with:

```
</body>
```

Just like the head section can contain other elements like the title and meta information, the body section can contain several HTML elements as well. For example, inside the body section you find all the link and image elements along with paragraphs, tables, and whatever else is necessary to display the page.

Later in this chapter, you see how to add links and images to a page. Next you learn about the basic page elements found on a web page.

Creating Good HTML

A good web page is structured in a logical order. This means that you place elements in a certain order so that they can be read properly by a web browser and that any time you open an element you also close that element using the corresponding tag that includes angle brackets and a forward slash. Doing so ensures that the page will display like you want it to when viewed in any web browser. Later in this chapter, you see how to check your HTML document to make sure it's structured correctly, but here we tell you how to choose the appropriate elements for your needs.

Using the appropriate elements

Web pages frequently use several page elements, sometimes called tags. Table 1-1 describes some of these elements.

Table 1-1		Common HTML Elements
Element	Description	Typical Use
<a>	Anchor	Creates a link to another page or a section of the same document.
 	Line break	Enters a line break or return character.

<i>Element</i>	<i>Description</i>	<i>Typical Use</i>
<code><div></code>	A section of a page	Creates overall areas or logical divisions on a page, such as a heading/menu section, a content area, or a footer.
<code><form></code>	Web form	Creates a web form to accept user input. Covered in Chapter 3 of this minibook.
<code><h1></code> through <code><h6></code>	Heading	Creates a container for a heading, such as heading text.
<code><hr></code>	Hard rule	Creates a horizontal line.
<code></code>	Image	A container for an image.
<code><input></code>	Input	An element to accept user input. Covered in Chapter 3 of this minibook.
<code><link></code>	Resource link	Links to a resource for the page; not to be confused with an anchor element.
<code><p></code>	A paragraph in a page	Creates textual paragraphs or other areas and containers for text.
<code><script></code>	A script tag	Denotes a web script or program. Also frequently found in the head section.
<code></code>	Span	Creates a container for an element. Frequently used in conjunction with styling information.

Related to the structure or layout of the elements is a concept called *semantic markup*, which is a fancy term to say that you always use the right element at the right time. In other words, you use the right kind of element to hold text and the right kind of element to add line breaks to a page. Consider these benefits of semantic markup:

- ◆ **Improves search results.** A primary benefit of semantic markup is that visitors and search engines alike can find the information they need.
- ◆ **Simplifies maintenance.** A secondary benefit to semantic markup is that it makes maintenance easier later on.

When a page is both semantically correct and valid HTML, it is said to be *well-formed*.

Putting text on a page

There are many ways to insert text into a web page and many elements that are appropriate for holding text. Heading elements such as `<h1>`, `<h2>`, through `<h6>`, are the correct place to put headings, while `<p>`, ``, and `<div>` are appropriate containers for longer form text, such as

paragraphs. Listing 1-2 shows a simple web page with two headings and some paragraphs.

Listing 1-2: A Web Page with Headings and Paragraphs

```
<!doctype html>
<html>
<head>
<title>My First Document</title>
</head>
<body>
<h1>My Web Page</h1>
<p>Welcome to my web page. Here you'll find all sorts of
    information about me.</p>
<h2>My Books</h2>
<p>You can find information on my books here as well.</p>
</body>
</html>
```

When viewed in a web browser, this page appears like Figure 1-2.

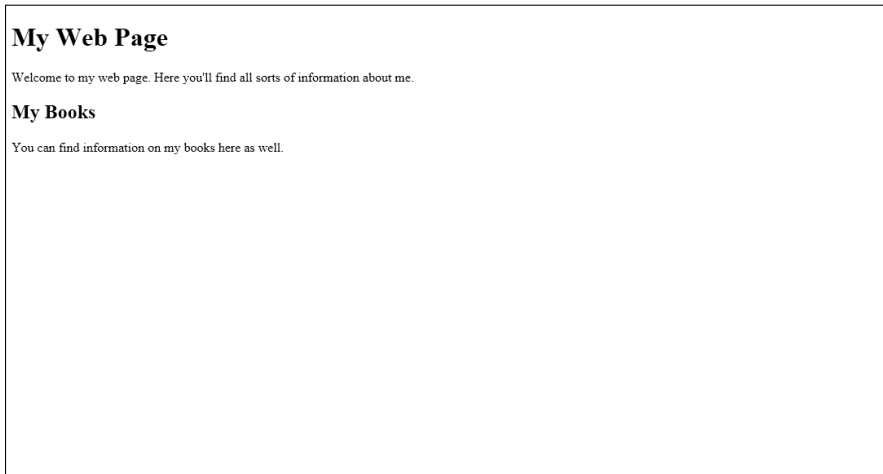


Figure 1-2:
A simple
web
page with
headings
and
paragraphs.



As you can see from Figure 1-2, the information on the page includes an `<h1>` element, followed by a paragraph, `<p>`. When the paragraph is closed with `</p>`, another heading element, this time an `<h2>`, is found. When the second heading is closed, `</h2>`, another paragraph is found.

It would've also been possible to substitute `<div>` elements in place of the paragraph elements on the page.

Creating your first page

Enough of us showing you HTML; it's time for you to build a page. You can create HTML with any text editor; in fact, it often is better to use a plain text editor rather than an expensive HTML creation tool.



It's important to note that you should use a text editor and not a word processor like Microsoft Word. Microsoft Word or a similar program like Pages on a Mac add all sorts of extra formatting information that get in the way of creating good HTML, even in their Save as HTML option.

Therefore, on Windows, use a program like Notepad. Even the Windows program Wordpad can place extra formatting information in it. When it comes to an HTML editor, the simpler the better.

The text editor included with Linux depends on the distribution of Linux you're using. One of your humble book authors' (that would be Steve) personal preference is for the command-line editor call Vi or Vim; a more graphical experience is typically found with a program called gEdit — the default text editor for Ubuntu.

Mac includes a program called TextEdit that can be used for creating plain text documents — but be careful: The TextEdit program will attempt to save files in Rich Text Format (RTF) by default. When creating or saving files with TextEdit, select Plain Text from the File Format drop-down menu.



This chapter focuses on the basics. Don't worry that your web page doesn't look stylish. The next chapter explains how to style your page with Cascading Style Sheets, or CSS.

Follow these steps to create your page:

1. Open your text editor.

See the preceding discussion about text editors. You want a text editor that allows plain text without extra information.

2. In the text editor, enter the following HTML.

```
<!doctype html>
<html>
<head>
<title>My First Web Page</title>
</head>
<body>
<h1>My web page!</h1>
<p>Hello world, welcome to my web site</p>
</body>
</html>
```

3. Save the file as `firstpage.html`.



Save the file exactly as named, using lowercase throughout the name. Later in the chapter, you can practice validating this file.

Apache, the web server used to send the files to your browser, is case sensitive for filenames, so sticking with lowercase will save you lots of headaches. Make sure the extension is `.html` and not `.txt` or another extension. Save the file to your document root, which is discussed in Book I. The document root location depends on how you've installed Apache and on what type of system you're using.

If you're using a hosting provider, then this is the point where you upload the file to their system.

4. Open your web browser to load the page.

In the web browser, point to `http://localhost/firstpage.html`. When you do so, you'll see a page like Figure 1-3.

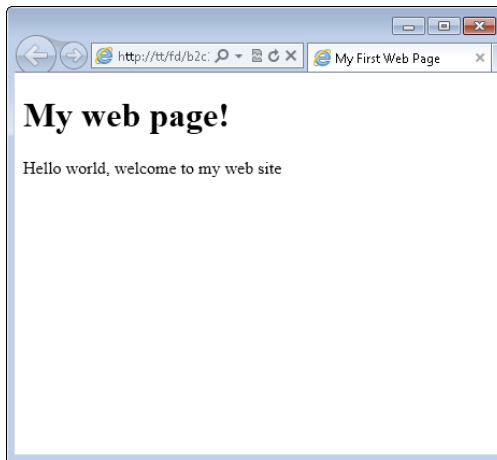


Figure 1-3:
Your first
page,
viewed
through a
browser.

Choosing block-level or inline elements

When you're considering which type of element to add to your page, think about whether you'd like it to extend across the width of the page.

- ◆ **Block-level elements:** Both `<div>` and `<p>` elements are known as *block-level elements*. A block-level element displays across the entire width of the page; nothing can appear next to or alongside a block-level element. Essentially, think of block-level elements as having a carriage return after them.
- ◆ **Inline elements:** Certain elements, primarily the `` element, are considered *inline elements*, which means that other elements can appear next to them. In other words, inline elements don't have a carriage return after them.

Inserting line breaks and spaces

There are times when you create a page and want to insert a line break. To accomplish this action in a word processor, you simply press the Enter or Return key on the keyboard. Things are not so simple in HTML. No matter how many times you press Enter in an HTML document, the text will still display on the same line in the web browser. Consider the code in Listing 1-3. It's the same HTML as Listing 1-2, but has five extra carriage returns inserted.

Listing 1-3: Trying to Insert Carriage Returns into HTML

```
<!doctype html>
<html>
<head>
<title>My First Document</title>
</head>
<body>
<h1>My Web Page</h1>
<p>Welcome to my web page. Here you'll find all sorts of
    information about me.</p>

<h2>My Books</h2>
<p>You can find information on my books here as well.</p>
</body>
</html>
```

When viewed through a web browser, the output is the same as Figure 1-2 earlier in the chapter. You see no blank lines between the first paragraph and the second heading.

The same thing happens to extra spaces in HTML. No matter how many times you press the space bar on the keyboard in a web document, the most you'll ever get is a single space. (We tell you more about how to add spaces at the end of this section.)

The `
` tag is used to insert line breaks into web pages. Look at the code in Listing 1-4. Instead of using the Enter key (or Return on a Mac), the `
` tag is used to add carriage returns:

Listing 1-4: Using `
` for Line Breaks

```
<!doctype html>
<html>
<head>
```

```
<title>My First Document</title>
</head>
<body>
<h1>My Web Page</h1>
<p>Welcome to my web page. Here you'll find all sorts of
    information
about me.</p>
<br>
<br>
<br>
<br>
<br>
<h2>My Books</h2>
<p>You can find information on my books here as well.</p>
</body>
</html>
```

When viewed in a browser, the desired effect is shown, as illustrated in Figure 1-4.

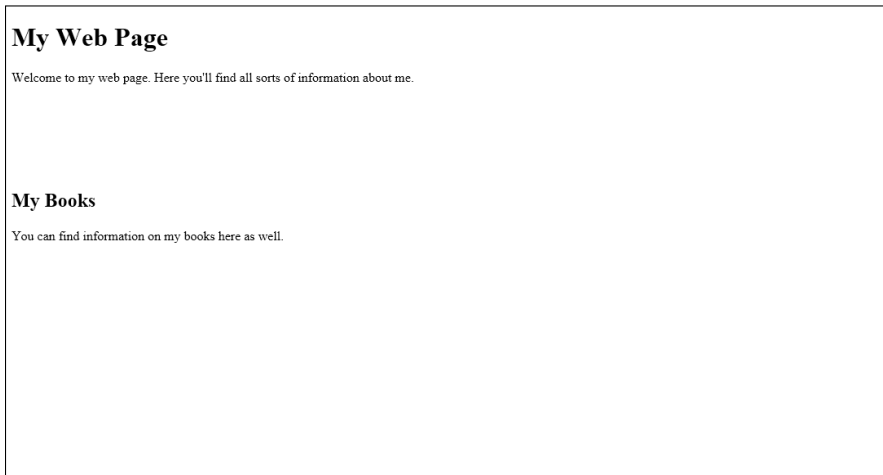


Figure 1-4:
Using `
`
to insert
carriage
returns.



You'll sometimes see an extra slash in some tags like `
` so they'll be written as `
`. This is a holdover from XHTML but is not necessary for HTML5.

While we're on the subject, you'll also notice that `
` doesn't have a closing partner, like a `</br>`. That's ok. You can use `
` as-is, without worrying about having to close it.

Adding spaces to HTML is accomplished with the ` ` entity sometimes written as ` `. However, there are better ways to accomplish spacing

in HTML, chiefly through the use of Cascading Style Sheets (CSS). Therefore, the use of the ` ` entity won't be covered in favor of the more common and more widely supported method through CSS — the topic in Chapter 2 of this minibook.

Making your document easier to maintain

Developers frequently use comments to note behind-the-scenes information about the page or about their code, and comments don't display on the web page. For example, a comment in a web page might be something like "I added this on 10/19/2012" or "Added in support of our sales initiative." If you visit the web page, you can see those comments only by looking at the page's HTML file.

HTML comments are opened with this syntax:

```
<!--
```

HTML comments are closed with this syntax:

```
-->
```

Everything that appears from the beginning `<!--` to the first `-->` is considered part of the comment. Listing 1-5 contains an example HTML document with a comment.

Listing 1-5: Adding an HTML Comment

```
<!doctype html>
<html>
<head>
<title>My First Document</title>
</head>
<body>
<h1>My Web Page</h1>
<p>Welcome to my web page. Here you'll find all sorts of
    information
    about me.</p>
<!-- Adding information about my books 10/1/2012 -->
<h2>My Books</h2>
<p>You can find information on my books here as well.</p>
</body>
</html>
```



HTML comments are visible by the world and should never be used to store any information considered privileged or private.

HTML comments can span multiple lines, as in the example in Listing 1-6:

Listing 1-6: A Multi-line Comment

```
<!doctype html>
<html>
<head>
<title>My First Document</title>
</head>
<body>
<h1>My Web Page</h1>
<p>Welcome to my web page. Here you'll find all sorts of
    information
    about me.</p>
<!--
    Adding information about my books
    Date: 10/1/2012
-->
<h2>My Books</h2>
<p>You can find information on my books here as well.</p>
</body>
</html>
```

In this comment, you can see that the actual text of the comment is indented, which brings up another important point: It's helpful to use indentation when creating documents. Documents are easier to read and maintain later when elements are indented, so that way you can clearly see visually which elements are “inside” of which other elements.

Adding lists and tables

Lists and tables help to represent certain types of information. For example, a list of trees in Steve's yard is best represented with a list like this:

```
Pine
Oak
Elm
```

But if he wants to include more information about the trees, a table is a better format:

<i>Tree Type</i>	<i>Description</i>
Pine	A common tree in my yard.
Oak	There are a few oaks in my yard.
Elm	I have one Elm in my yard but it's too close to the house.

HTML has tags to create both lists and tables. Table 1-2 describes a variety of such elements.

Table 1-2 Common List and Table Elements in HTML		
<i>Element</i>	<i>Type</i>	<i>Description</i>
	List Item	Used in conjunction with or to create lists of information.
	Order List	An ordered list of information, used in conjunction with .
<table>	Table	Used with <tr>, <td>, and other elements to create a table for presenting information.
<td>	Table Cell	Creates a cell in a table row.
<th>	Table Header	A table cell that's a heading.
<tr>	Table Row	Creates a row of a table.
	Unordered List	Related to and to create lists of information.

When building a list, you have two choices of the type of list to create: an ordered list or an unordered list. Ordered lists are used for things like making an outline, while unordered lists make up pretty much every other kind of list.

Listing 1-7 shows the HTML used to create a standard unordered list.

Listing 1-7: Creating an Unordered List

```
<!doctype html>
<html>
<head>
<title>An unordered list</title>
</head>
<body>
<ul>
  <li>Pine</li>
  <li>Oak</li>
  <li>Elm</li>
</ul>
</body>
</html>
```


When viewed in a browser, this HTML results in a page like that in Figure 1-5.

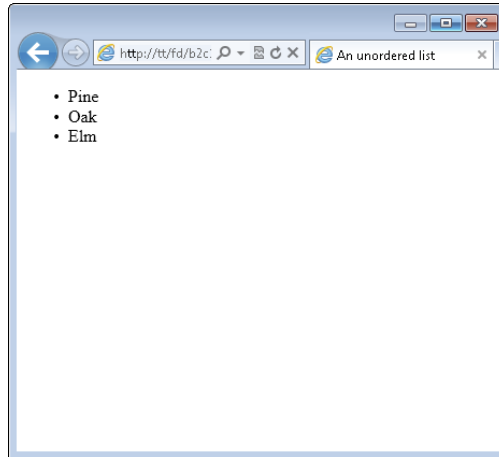


Figure 1-5:
An
unordered
list.

The unordered list created in Listing 1-7 uses the default styling for the list, which adds bullets next to each item. You can also change the style of this bullet or not include one at all using CSS. You learn more about CSS in the next chapter.

Creating an ordered list means simply changing the `` element to ``. Doing so looks like this:

```
<ol>
  <li>Pine</li>
  <li>Oak</li>
  <li>Elm</li>
</ol>
```



When viewed in a browser, the bullets from the preceding example are replaced with numbers, as in Figure 1-6.

Other types of lists, such as definition lists, exist but aren't covered here.

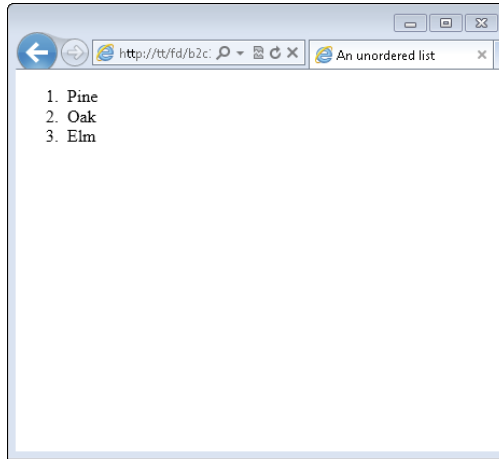


Figure 1-6:
An ordered
list.

Practicing Creating a Table

It's time to create a page with a table. Follow these steps:

1. Open your text editor.

See the preceding exercise for more information on text editors.

2. In the text editor, create a new text document.

Most text editors will open with a blank or empty document to begin with. If you have anything in the document, clear it out before continuing.

3. Enter the following HTML:

```
<!doctype html>
<html>
<head>
<title>My First Web Page</title>
</head>
<body>
<h1>My Table</h1>
<table>
  <tr>
    <th>Airport Code</th>
    <th>Common Name/City</th>
  </tr>
```

```
<tr>
  <td>CWA</td>
  <td>Central Wisconsin Airport</td>
</tr>
<tr>
  <td>ORD</td>
  <td>Chicago O'Hare</td>
</tr>
<tr>
  <td>LHR</td>
  <td>London Heathrow</td>
</tr>
</table>
</body>
</html>
```

4. Save the file as **table.html**.

Save the file, as you did for the preceding exercise, with a `.html` extension. The file should be saved in your document root. Refer to the preceding exercise or Book I for more information on finding your document root if you haven't already found it for that exercise.

5. View the file in your browser.

Open your web browser and type **`http://localhost/table.html`** into the address bar. Doing so will show a page like the one in Figure 1-7.

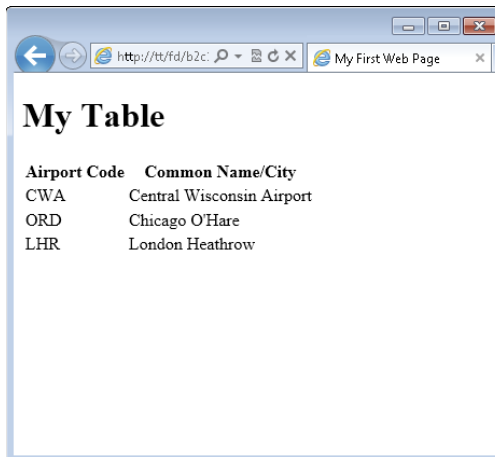


Figure 1-7:
The table
you created
for this
exercise.

Notice that the table doesn't have any borders around it. If you'd like to add borders, keep working through this exercise. Otherwise, continue to the next section.

6. Open `table.html` in your text editor.

If you closed your text editor, open it again and load `table.html`.

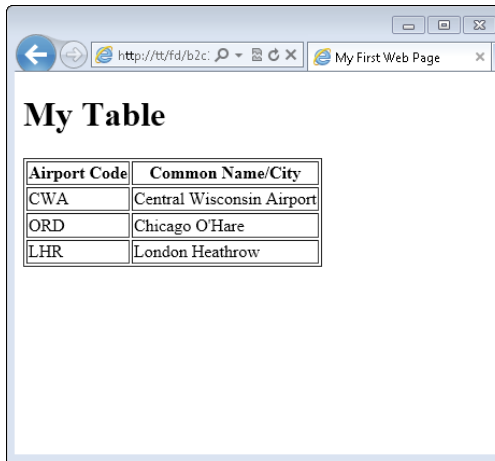
7. Change the code in `table.html` to the following:

```
<!doctype html>
<html>
<head>
<title>My First Web Page</title>
</head>
<body>
<h1>My Table</h1>
<table border="1">
  <tr>
    <th>Airport Code</th>
    <th>Common Name/City</th>
  </tr>
  <tr>
    <td>CWA</td>
    <td>Central Wisconsin Airport</td>
  </tr>
  <tr>
    <td>ORD</td>
    <td>Chicago O'Hare</td>
  </tr>
  <tr>
    <td>LHR</td>
    <td>London Heathrow</td>
  </tr>
</table>
</body>
</html>
```

Note that the only change is to add a space and then `border="1"` within the `<table>` element.

8. Reload `table.html` in your browser.

If you closed your browser, reopen it and go to `http://localhost/table.html`. If your browser is still open, press `Ctrl+R` to refresh the page (`Command+R` on a Mac). You now see a border around the table, as in Figure 1-8.



The screenshot shows a web browser window with the address bar displaying 'http://tt/fd/b2c:'. The browser title is 'My First Web Page'. The main content area displays a table titled 'My Table'.

Airport Code	Common Name/City
CWA	Central Wisconsin Airport
ORD	Chicago O'Hare
LHR	London Heathrow

Figure 1-8:
A table with
borders
around each
cell.

This is a rather primitive way to add a border to a table. A better way to accomplish this task is by using CSS, which you learned about briefly in Book I. Chapter 2 of this minibook covers CSS in much more detail, too.



When you added `border="1"` to the `<table>` element, you added something called an *attribute*. An attribute helps to further describe or define the element or provides additional details about how that element should behave.

Including Links and Images on Your Web Page

What would the web be without links — and images too? Not much of web at all. Links are the items that you click on inside of web pages to connect to or load other pages, and when we talk about images, we mean both illustrations and photos. This section looks at how to add links and also images to your web page.

Adding links

Links are added with the `<a>`, or anchor element. The `href` attribute tells the anchor element the destination for the link. The destination can be just about anything, from another web page on the same site, to a different site, to a document or file, to another location within the same web page. The link itself can be added to just about anything on the page. For example, you might link each of the trees mentioned in the previous section to articles about each of those types of trees.

When something is linked, the browser typically gives visual feedback that there's a link by highlighting and underlining the linked area. You'll see an

example of this shortly. Like other HTML elements, the `<a>` element has a corresponding closing `` tag that is used to tell the browser when to stop highlighting and underlining the link.

Linking to other pages

Linking to other pages, whether on the same site or at a different site, is accomplished in the same way. For example, look at the following HTML:

```
<p>Here's a link to <a href="http://www.braingia.org">Steve  
Suehring's site</a></p>
```

This line uses a paragraph element `<p>` to create a sentence, “Here’s a link to Steve Suehring’s site.” This being the web, you decide to actually provide a link so that visitors can click on certain words and be transported to that page. You do so with the `<a>` element along with the `href` attribute. In this case, the `<a>` element looks like this:

```
<a href="http://www.braingia.org">
```

The `href` attribute points to the URL `http://www.braingia.org` and is enclosed in quotation marks. The text that will be highlighted then appears, followed by the closing `` tag.

Here’s an exercise for implementing this link.

1. Open your text editor.

You use your text editor to create a new file, so there should be nothing in the text editor except a blank document or file.

2. In the text editor, place the following HTML:

```
<!doctype html>  
<html>  
<head>  
<title>Link</title>  
</head>  
<body>  
<p>Here's a link to <a href="http://www.braingia.  
org">Steve Suehring's site</a></p>  
</body>  
</html>
```

3. Save the file as `link.html`.

The file should be saved to your document root with the name `link.html`.

4. Open your browser and view the page.

Open your web browser and point to `http://localhost/link.html` by entering that URL into the address bar. You’ll see a page like that in Figure 1-9.

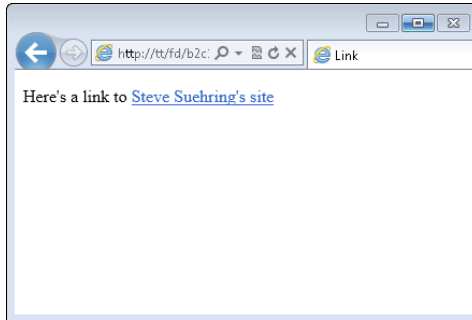


Figure 1-9:
A page with
a link.



Always close `<a>` elements with a corresponding closing `` tag. A frequent mistake is to leave the `<a>` element open, resulting in all the text that follows to be highlighted as a link.

The example and exercise show how to link to a page on a different website. Creating a link to a page on the same site is accomplished in the same manner, but rather than including the Uniform Resource Identifier (URI) scheme and the hostname (the `http://www.braingia.org` part from this example), you can just link to the page itself.

If you've been following along with previous exercises, then you should have a page called `table.html`. Here's HTML to create a link to `table.html`. The preceding exercise's HTML is included so that you can see the overall context for the link:

```
<!doctype html>
<html>
<head>
<title>Link</title>
</head>
<body>
<p>A link to <a href="table.html">the table example</a></p>
</body>
</html>
```

Like before, the link is contained within a `<p>` element but note the `href` attribute now points merely to `table.html`.



Avoid spaces in filenames and in web URLs in general. Spaces are generally not friendly to URLs, in filenames, or in images. Though they can be worked around, you'll have much more success if you always simply avoid spaces when naming things for the web.

Understanding absolute versus relative links

The link shown in the preceding example is called a *relative link* because it does not begin with either the Uniform Resource Identifier (URI) scheme (`http://`) or a beginning front slash (`/`). A relative link assumes that the target (`table.html` in the example) is in the same directory or folder as the document or page from which it's linked. In the case of the example, a relative link works because the current page, `link.html`, and the page being linked, `table.html`, both exist in the document root.

If both pages were not in the same directory (in other words, if `table.html` was in a folder called *tables* in the document root and the `link.html` file was in a folder called *links*

in the document root), then you would need to create an *absolute link*. An absolute link tells the server exactly where to look to find the target. For example, an absolute link might look like `/tables/table.html`. This link tells the server that it needs to begin looking in its document root for a directory called `tables` and that it should then find a file called `table.html` in the `tables` directory.

Use absolute links when you need to provide exact or absolute references to the target being linked. Use relative links when the resource being linked will always be found in the same place relative to the page linking to it. If the location of the page or the target changes, then relative links will stop working.

Linking within a page

Sometimes you want to link within the same page. You might do this on a particularly long page, where you have a table of contents at the top and then the full article lower down in the page.

Creating withinpage links uses the same `<a>` element that you've seen, this time with the `name` attribute. Listing 1-8 shows HTML to create a within-page anchor.

Listing 1-8: An In-Page Anchor

```
<!doctype html>
<html>
<head>
<title>Link</title>
</head>
<body>
<ul>
  <li><a href="#pine">Pine</a></li>
  <li><a href="#oak">Oak</a></li>
  <li><a href="#elm">Elm</a></li>
```



```
</ul>
<p><a name="pine">Pine trees are abundant in my yard.</a><p>
<p><a name="oak">There are a few oak trees in my yard.</a><p>
<p><a name="elm">There's one elm in my yard.</a><p>
</body>
</html>
```

In Listing 1-8, the href tags added to each of the list items use a pound or hash sign (#). This is the key used to tell the browser that the resource will be found on the same page. Then later on in the HTML you see another <a> element, this time using the name attribute. That name attribute corresponds to each of the href attributes from earlier in the page.

That's it! There's nothing more to adding in-page links. You merely need to use the pound sign to indicate that the resource is found later on the page and then use the name attribute to make another element match that.

Opening links in a new window

Sometimes you want to make a link open in a new tab or a new window. When a visitor clicks a link that's defined in such a way, the browser will open a new tab and load the linked resource in that new tab. The existing site will still be open in the visitor's browser, too.



Don't make every link open in a new window. You should do so only where it makes sense, as might be the case where a visitor is in the middle of a long process on your website and needs to link to reference another resource or site, like a directory of ZIP codes or a terms of service agreement. Also, whether the link opens in a new tab or a new window is dependent on the browser; you can't control it.

This can be done by adding the target attribute to your <a> element with a special value, `_blank`. For example, an earlier example showed how to create a link to Steve's website, www.braingia.org. Recall that the link looked like this:

```
<a href="http://www.braingia.org">Steve Suehring's site</a>
```

To make this link open in a new window, you add the `target="_blank"` attribute/value pair to the element, so it looks like this:

```
<a href="http://www.braingia.org" target="_blank">Steve
  Suehring's site</a>
```

You can try this out by opening the `link.html` file from the earlier exercise and adding `target="_blank"` as shown. Note the use of the underscore preceding the word *blank*. When you save the file and reload that page (Ctrl+R or Command+R), the link won't look any different. However, clicking the link will open a new tab (or new window, depending on your browser and configuration).

Adding images

Images, such as photos or graphics, enhance the visual appeal of a web page. Images are usually embedded in a page, such as shown in Figure 1-10, where a photo of the cover of another of Steve's books, *MySQL Bible* (John Wiley & Sons, Inc.), is shown.

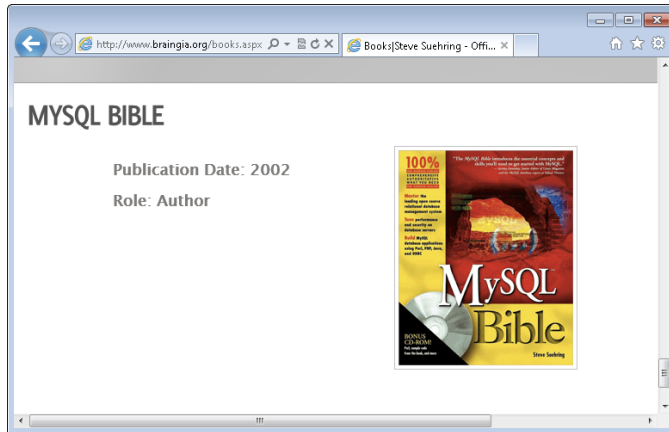


Figure 1-10:
An image on
a web page.

You can include images from anywhere, assuming that you have the legal rights to do so. In other words, you can store the image on your web server or you can include an image stored on someone else's web server. (But we repeat: First, make sure you don't violate any copyright!)

There's also another, special type of image, called a background image. Background images provide the background for the page itself. Chapter 2 of this minibook covers background images.

Referencing the image location

Images are added with the `` element. Just as with the `<a>` element, the `` element uses an attribute to tell the browser more information about itself. The `src` attribute is used to tell the browser where to find the image. Earlier, in Figure 1-10, you see an image of a book cover. The HTML to bring that image into the page looks like this:

```

```

As you can see, the `` element adds the `src` attribute, which then references where to find the image on the web server.



You might notice that the `` element doesn't have a closing `` tag. That's because this element doesn't have its own content, unlike the `<p>` and `<a>` elements — which both need content to go within them and therefore need to be closed. You may sometimes see an element like `` closed with `>/>` instead of just `>`, as in the example. Both are acceptable and valid ways to close this type of element.

The `` element should always have an `alt` attribute. The `alt` attribute tells search engines and assistive technologies about the image being used. When used with an `` element, the `alt` attribute looks like this:

```

```

You should use a short description as the contents of the `alt` attribute. Using something like “MySQL Bible was a great book and everyone should’ve purchased one” doesn’t describe the image, but “MySQL Bible” does.

Choosing good web images

When choosing an image for the web you need to look at more than just making sure no one blinked when the photo was taken. You should also consider the image's height and width, the size of the file, and its format. Web browsers can view images formatted in numerous formats, including JPG, GIF, and PNG, as well as several others.

The height and width of the image are up to you and depend on the needs of your page. For example, Steve needed a special sized file in order to display the *MySQL Bible* book cover. He used image manipulation software in order to resize the image for his needs. Many image manipulation and image processing software programs are available. Adobe Photoshop and Gimp are among the most popular ones.

File size is arguably one of the most important aspects for your consideration when choosing an image. When you include large images, such as those taken at the high-quality setting with your digital camera, visitors have to download the file, which can take an extraordinarily long time depending on the speed of the visitors' connection. If they're visiting from a dial-up modem or slower connection, then an image that's 4 megabytes (MB) may take 20 minutes to load! This is also true with today's mobile devices, on which the speeds may be slower and a visitor using such a device may have to pay data download fees.

To get around this, you can resize your images using the aforementioned software. Resizing images to under 100 kilobytes (KB) is important. Another important aspect to consider is the sum of all images on the page. For example, if you have 15 images at 100KB each, then you're requiring the visitor to download 1.5MB worth of images — which is likely too much for many visitors. If the page seems slow to load, they may go elsewhere rather than wait.



When you're choosing an image format, know that if you choose one of the three formats mentioned earlier (JPG, GIF, and PNG), you ensure that the widest possible audience can view the image without needing special software to do so.

Keep the sum of all images in mind when sizing the images for your page so that the page downloads faster for the visitor.

Creating a page with an image

It's time to create a page with an image so that you can see how and where an image fits within the larger whole of an HTML page. Follow these steps.

1. Open your text editor.

See the previous discussion about text editors.

2. In the text editor, enter the following HTML.

```
<!doctype html>
<html>
<head>
<title>A snowy picture</title>
</head>
<body>
<h1>A snowy picture</h1>
<p></p>
</body>
</html>
```

When you create this HTML you need to use a photo or other picture of your own or you can use the `snow.jpg` file included in the companion content of this book. Regardless of the picture you choose, you need to place the file in the document root of the web server (discussed in Book I). Also, make sure that case (uppercase and lowercase) for the filename matches what you put in the `src` attribute. In other words, if your picture is called `TheKids.JPG`, then the `src` attribute should be `"TheKids.JPG"`.

3. Save the file as `image.html`.

Save the file exactly as named, using lowercase throughout the name. The file should be saved to your document root, which is discussed in Book I. The document root location depends on how you've installed Apache and on what type of system you're using. If you're using a hosting provider, then this is the point where you upload the file to that host provider's system.

4. Open your web browser to load the page.

In the web browser, point to `http://localhost/image.html`. When you do so, you'll see a page like Figure 1-11.



Figure 1-11:
Adding an
image to a
page.

This HTML used an `` element to load a photo called `snow.jpg` from the current directory. In other words, `snow.jpg` was in the same directory as the `image.html` page on the web server.



Avoid spaces in image filenames, just as you would for regular files and other URLs. Remember also that URLs, files, and images are case sensitive.

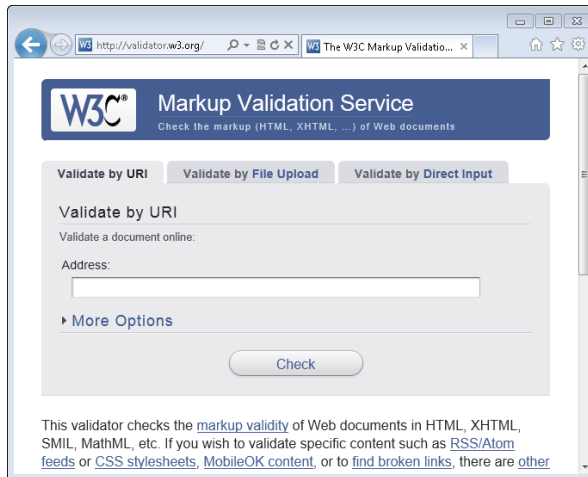
Writing Valid HTML

When you create a web page with HTML, there are certain rules to follow in order to make sure that web browsers can read and display the page correctly. HTML and its rules are discussed in the first minibook included in this guide. The current version of the HTML specification is HTML version 5, known simply as HTML5.

The process of validating a page means that a specialized website examines the HTML code that you write and compares it to the specification for that version of HTML. In the case of the HTML that you're writing for this book you are using HTML5.

The website used to validate HTML is called the W3C Markup Validation Service (frequently called the W3C Validator) and is operated by the World Wide Web Consortium (W3C). The W3C Validator is found at <http://validator.w3.org> and is free to use. Figure 1-12 shows the W3C Validator.

Figure 1-12:
The W3C
Markup
Validation
Service,
sometimes
simply
called the
Validator.



Validate your HTML in one of three ways:

- ◆ **Providing a URL:** You can enter a URL into the Validator and it will automatically retrieve the HTML at that URL and attempt to validate it. In order for the Validator to retrieve your HTML using this method, the page needs to be available to the public. This is usually not the case when you've installed a web server on your computer, as discussed in this book. If you're using an external hosting provider, then your site and pages may be available to the Internet. In that case, you can enter the URL in the "Validate by URI" address box.
- ◆ **Uploading a file:** You can upload a file using the "Validate by File Upload" option. Using this method, you choose a file on your computer. That file is then uploaded to the Validator.
- ◆ **Pasting HTML into the Validator:** This means copying the HTML from your text editor and pasting it into the "Validate by Direct Input" tab in the Validator. This option is typically the fastest and easiest method and it's the one that we show in this section.

Validating Your HTML

If you've followed the exercises in this chapter, then you've built some HTML. The next exercise uses the W3C Validator to make sure that the HTML you've written is valid according to the HTML5 specification. Follow these steps:

1. Open `firstpage.html` using your text editor.

This page was the first one you created in this chapter. However, if you skipped that exercise, open any one of the HTML files that you created in this chapter.

2. Highlight/select all the HTML in the open file.

Use your mouse or pointing device to highlight all the HTML or press Ctrl+A on Windows or Command+A on Mac.

3. Copy the HTML to your clipboard.

Select Copy (found in the Edit menu in most text editors) or press Ctrl+C on Windows or Command+C on Mac to copy the highlighted HTML to the clipboard.

4. Open your web browser and navigate to the W3C Validator.

With the browser open, type **http://validator.w3.org** in the address or location bar in the browser and press Enter to go to the Validator.

5. Select Validate by Direct Input.

The Validate by Direct Input tab will be used to paste in the code in your clipboard.

6. Paste the HTML into the Validator.

Press Ctrl+V on Windows or Command+V on Mac to paste the HTML from the clipboard into the Enter the Markup to Validate box on the Validator page. If you're using the HTML from firstpage.html, your screen should look similar to that in Figure 1-13.

7. Click Check.

Click the Check button on the Validator page to run the validation of your HTML. You should receive a page similar to that in Figure 1-14.

Figure 1-13:
Pasting
HTML
into the
Validator.

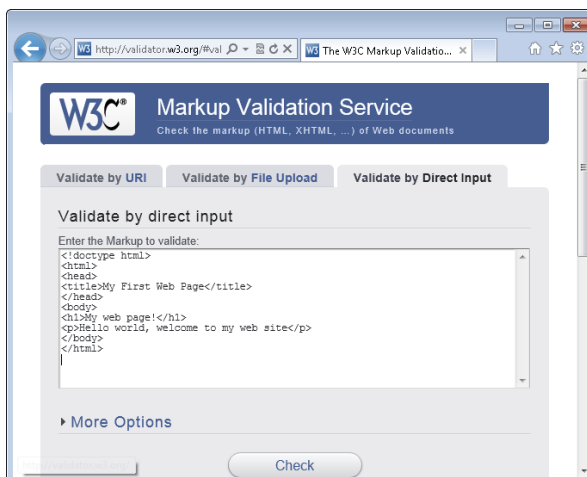
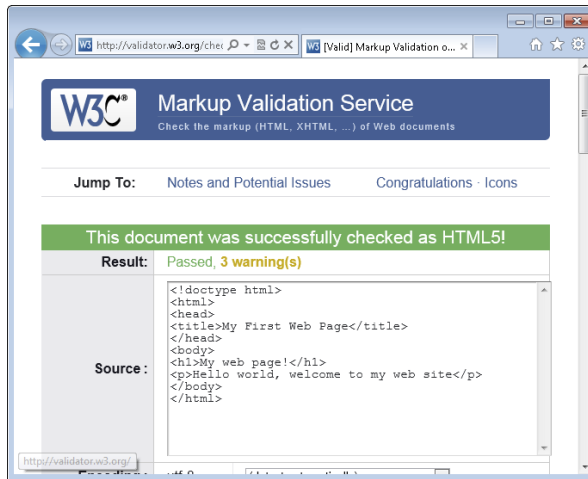


Figure 1-14:
A valid
HTML
document
passed
through the
Validator.



Notice the three warnings in this validation. Scrolling down reveals that one of the warnings is that the HTML5 validator is actually experimental at this time, though that may change by the time you read this. The other two warnings are related to language settings.



It's good practice to include the character encoding, which helps the browser determine how to read the document, including what language is used for the HTML and the page. See <http://www.w3.org/International/tutorials/tutorial-char-enc/#Slide0250> for more information on character encoding.

Chapter 2: Adding Style with CSS

In This Chapter

- ✓ Finding out what styling the page means
- ✓ Exploring different methods of using CSS
- ✓ Selecting certain elements for styling
- ✓ Changing fonts and adding borders
- ✓ Adding list styles
- ✓ Modifying backgrounds
- ✓ Working with layout
- ✓ Adding a header and footer

In the preceding chapters, you learn a little about a lot and a lot about a few things. Namely, you learn how to install a web server and a database system and you learn a little about HTML. Although HTML is used to add text to a page, that text is pretty boring; it needs some style. Enter CSS.

In this chapter, you learn what Cascading Style Sheets (CSS) is and how to use it for various layout and style purposes. We recommend that you work through the chapter from beginning to end, because some exercises build on previous exercises.

Discovering What CSS Can and Can't Do for Your Web Page

This section looks at CSS from a high level to give you a foundation on which you'll learn how to use CSS on your website.

What is CSS?

CSS complements HTML by providing a look and feel to web pages. The HTML pages you created in the preceding chapter looked fairly plain, with a default font and font size. Using CSS, you can spice up that look, adding color and background images, changing fonts and font sizes, drawing borders around areas, and even changing the layout of the page itself.

CSS has its own language, separate from HTML, but you wouldn't use CSS without the HTML page. In other words, although HTML can stand on its own and present a page to a browser, CSS can't. You wouldn't write a CSS page. Rather, you write HTML and then use CSS to help style that page to get it to look like you want it to.



Like HTML, CSS is defined by specifications, with the latest being CSS version 3, known as CSS3.

Why use CSS?

Before CSS, an HTML developer changed fonts and colors by changing attributes on each element. If the developer wanted all the headings to look a certain way, he had to change each of those headings. Imagine doing this on a page with ten headings, and then imagine doing it on 50 pages. The task quickly becomes tedious. And then think of what happens when the site owner decides she wants all the headings changed back to the original way.

CSS alleviates this burden of individually updating elements and makes it so that you can apply one single style across one or more elements. You can apply multiple styles to the same element, and you can target a certain style down to the individual element. For example, if you want all headings to be bold font but a certain heading should have italic, you can do that with CSS.



Use CSS to make changes to the layout, look, and feel of a web page. CSS makes managing these changes easy.

Limitations of CSS

CSS isn't without limitations. The primary limitation of CSS is that not all web browsers support CSS in exactly the same way. One browser might interpret your layout in a slightly different manner, placing items higher or lower or in a different place entirely.

Also, older browsers don't support newer versions of CSS, specifically the CSS3 specification. This means that those browsers can't use some of the features of the CSS3 specification. To get around this, you can use older versions of the specification that are more widely supported by those older browsers.

The key when using CSS and, as you see later, when using JavaScript, is to test across multiple browsers. Web browsers such as Firefox, Chrome, and Safari are all free downloads, and Microsoft offers software called the Virtual PC for Application Compatibility, which are free, time-limited, versions of Windows that include older versions of Internet Explorer. You can run them inside of Microsoft's free Virtual PC emulation software. By testing in other browsers, you can see how the site will look in those browsers and correct layout issues prior to deploying the site to the Internet.



Always test your pages in multiple browsers to ensure that they look and act like you intended.

Connecting CSS to a Page

You can add CSS to a page in a few different ways:

- ◆ Directly to an HTML element
- ◆ With an internal style sheet
- ◆ With an external style sheet

The most reusable way to add CSS to a page is by using an external style sheet, but the simplest is to add styling information directly on an element. We show each of these methods.

Adding styling to an HTML element

You add style to just about any HTML element with the style attribute, as in this example that makes all the text in the first paragraph into bold font:

```
<p style="font-weight: bold;">All of this text will be  
bold.</p>
```

When viewed in a browser the text is bold, as shown in Figure 2-1.

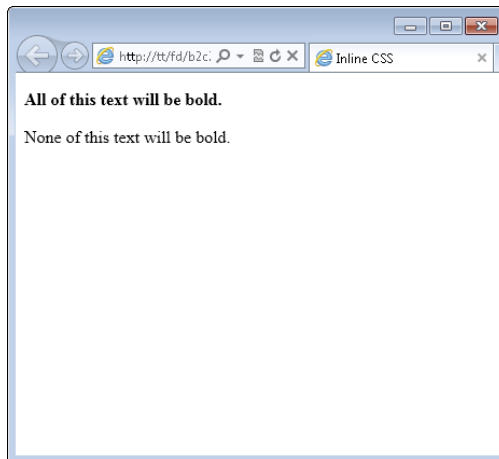


Figure 2-1:
Bold text
styled
with CSS.

In Figure 2-1, the paragraph with bold text appears above a normal paragraph. That normal paragraph doesn't use CSS for styling.



When a style is applied within an HTML element, it's called an *inline style* or *inline CSS*.

Here's an example that you can try. You build some HTML first and then begin to add styling to it.

1. Open your text editor.

Create a new blank file. See Chapter 1 of this minibook for more information on text editors and creating a new text document.

2. Within the blank text document, place the following HTML:

```
<!doctype html>
<html>
<head>
<title>A CSS Exercise</title>
</head>
<body>
<div>This is my web page.</div>

<div>
  This is the <span>nicest</span> page I've made yet.
</div>

<div>Here is some text on my page.</div>

</body>
</html>
```

3. Save the file as `css.html`.

Within the text editor, save the file using the name `css.html`, making sure there are no spaces or other characters in the filename. The file should be saved within your document root.

4. Open your web browser and view the page.

Within the web browser's address bar, type `http://localhost/css.html` and you'll see a page similar to that shown in Figure 2-2.

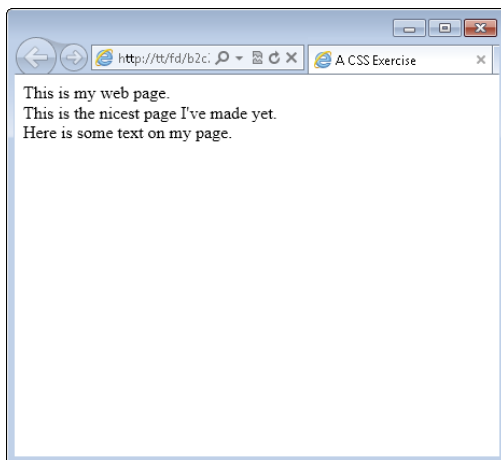
5. Close the browser.

Now that you've verified that the page is working, close the browser.

6. Switch to the text editor to edit the HTML.

Within the text editor, edit the HTML from Step 2 to add CSS. If you closed the file, reopen it in your text editor.

Figure 2-2:
Creating a
simple web
page.



7. Change the HTML to add two different style attributes, as shown here:

```
<!doctype html>
<html>
<head>
<title>A CSS Exercise</title>
</head>
<body>
<div style="font-weight: bold;">This is my web page.</div>

<div>
  This is the <span style="font-style:
    italic;">nicest</span> page I've made yet.
</div>

<div style="font-weight: bold;">Here is some text on my
  page.</div>

</body>
</html>
```

8. Save the file.

You can save it as `css.html` or save it as `css2.html` if you don't want to overwrite your original `css.html` file. The file should be saved in your document root.

9. Open your web browser and view the page.

Typing in **http://localhost/css.html** (or **css2.html** if you saved it as **css2.html**) reveals the file, now with inline styles applied to two areas. This is illustrated in Figure 2-3.

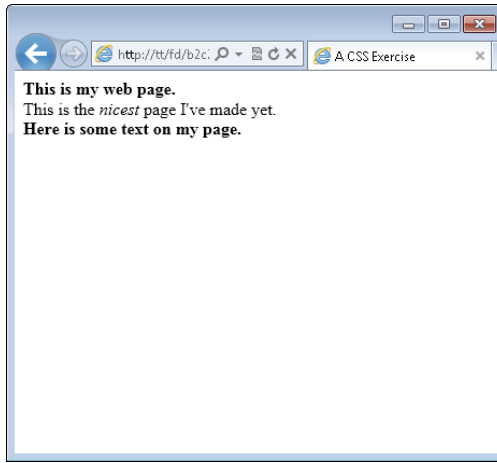


Figure 2-3:
Adding
inline styles
to the
HTML.

This exercise created an HTML file that used both `<div>` and `` elements. The HTML was then styled using inline styles. The inline styles adjusted both the `font-weight` and the `font-style` to create bold text for two elements and italic text for one element.

When used with CSS, `font-weight` and `font-style` are known as properties. These properties are then given values, such as `bold` and `italic`. When you see terminology that a CSS property was changed, you know that the property is the name and that the value is what to change that property to.

Using an internal style sheet

Applying styles to individual elements quickly becomes cumbersome when you have a large web page. As you see in the preceding exercise, in order to make the text of the two `<div>` elements bold you needed to add a style attribute to each of the `<div>` elements. Luckily, there's a better way.

You can create a special area of the web page to store styling information. This styling information is then applied to the appropriate elements within the HTML. This alleviates the need to add a style attribute to each element.

You add internal styles within the `<head>` portion of a web page using the `<style>` element. Listing 2-1 shows HTML with a `<style>` element.

Listing 2-1: Using an Internal Style Sheet

```
<!doctype html>
<html>
<head>
  <title>A CSS Exercise</title>
  <style type="text/css">
    div {
      font-weight: bold;
    }

    span {
      font-style: italic;
    }
  </style>
</head>
<body>
<div>This is my web page.</div>

<div>
  This is the <span>nicest</span> page I've made yet.
</div>

<div>Here is some text on my page.</div>

</body>
</html>
```

The page adds an internal style sheet to add a bold font to `<div>` elements and an italic styled font to all `` elements in the page.

```
<style type="text/css">
  div {
    font-weight: bold;
  }

  span {
    font-style: italic;
  }
</style>
```

The `<style>` element uses a type attribute to tell the browser what type of styling information to expect. In this case, we're using `text/css` type styling. Notice also the closing tag, which is required.

When this page is viewed in a browser, it displays like that in Figure 2-4.

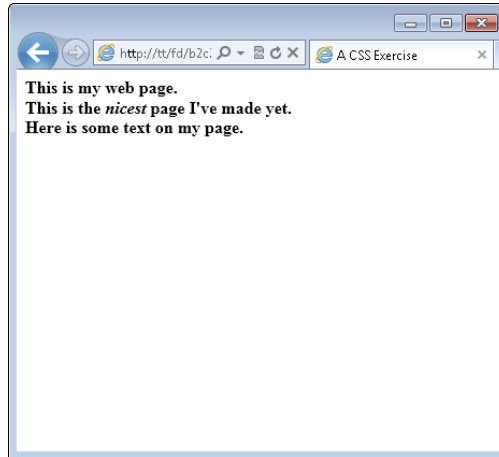


Figure 2-4:
Using an
internal
style sheet.

Look closely at Figure 2-4; notice the slight difference with the display from Figure 2-3. In Figure 2-3, the second line (“This is the nicest page I’ve made yet.”) is not bold, but the line appears in a bold font in Figure 2-4.

This difference is present because the internal style sheet targets all `<div>` elements in the page rather than just the specific ones that were changed with the inline style method shown earlier. The next section, “Targeting Styles,” shows how to fix this.

Using an external style sheet

You’ve seen how inline styles, adding styling information to each element individually, can become tedious. You then saw how to use an internal style sheet to create styling information for the page as a whole. But what happens when you have 10 pages or 100 pages, all needing styling?

You can use external style sheets to share CSS among multiple pages. An external style sheet, just another type of text document, can be included on every page. The browser reads this external style sheet just as it would read styles applied within the page itself, and applies those styles accordingly.

You add or include an external style sheet with the `<link>` element, which goes in the `<head>` area of an HTML page.

A typical `<link>` element to add CSS looks like this:

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

That's it. That line includes a file called `style.css` in the current directory and incorporates it into the page. All the `<style>` information and inline styling can be removed in place of that one single line in the `<head>` section of the page.

Inside the external style sheet are the rules to apply — and only the rules to apply. You don't need to include the `style` attribute or even an opening or closing `<style>` element within an external style sheet. Looking back at the example in Listing 2-1, the external style sheet would contain only this information:

```
div {  
    font-weight: bold;  
}  
  
span {  
    font-style: italic;  
}
```

Now that external style sheet can be shared among multiple HTML files. If you need to make a change to styling, you need to edit only the one CSS file, and it automatically applies the styles to all the pages that use that CSS file. As you can see, external CSS files make maintenance of websites much easier.



External style sheets are the recommended method for using CSS, and with only a few exceptions, the remainder of the book uses CSS included from an external style sheet.

Targeting Styles

Recall the problem identified earlier, where the bold font was applied to all the `<div>` elements on the page, when you might not necessarily want to apply it to all those elements. You can fix that problem by targeting or narrowing down the scope of the CSS rule using a more specific selector.

CSS uses *selectors* to determine the element or elements to which a rule will be applied. In the internal style sheet example earlier in this chapter, the selector was the `<div>` element, or all `<div>` elements on the page. In this section, we tell you how to select specific elements, and groups of elements, so that you can apply CSS styles to them.

Selecting HTML elements

Most any HTML element can be the target of a selector, even things like the `<body>` element. In fact, the `<body>` element is frequently used as a selector in order to target page-wide styles, such as what set of fonts to use for the page. You see an example of this in the next section, “Changing Fonts.”

You’ve already seen examples using HTML elements as selectors. You simply use the element name, with no brackets around it. Instead of `<div>` as it would be in HTML, you use `div` when using it as a CSS selector. Here’s what that looks like:

```
div {  
    font-weight: bold;  
}
```

As you can see, the name of the element, `div`, is followed by a brace. This indicates that the rule is beginning. Within the opening and corresponding closing brace, the property, `font-weight`, is selected, followed by a colon (`:`). The value is then set to `bold`. The line is terminated with a semicolon (`;`). This semicolon tells the browser that the line is done; in other words, the property/value pair are closed.

Multiple properties can be set in the same selector. Taking the preceding example, you could change the font’s style to be both bold and italic, like this:

```
div {  
    font-weight: bold;  
    font-style: italic;  
}
```

Each line is ended with a semicolon, and the entire rule is enclosed in opening and closing curly braces.

Selecting individual elements

What you’ve seen so far in this section is that you can target all HTML elements by simply using their names. You’ve been seeing examples of that throughout the chapter. But what happens when you want to target one, and only one, element on a page? That’s where the `id` selector comes into play.

The `id` (short for identifier) enables you to select one and only one element within a page. To do so, you need to modify the HTML to add an `id` attribute and provide a name for that element. For example, consider an HTML like this:

```
<div>Steve Suehring</div>
```

If you want to apply a bold font to that element, you could select all `<div>` elements but that would likely also apply a bold font to other `<div>` elements on the page, as you've already seen. Instead, the solution is to add an `id` to that particular `<div>`, like so:

```
<div id="myName">Steve Suehring</div>
```

The `id`'s value is set to `myName`. Note the case used in this example, with an uppercase `N`. This case should be matched in the CSS.

To select this `id` within the CSS, you use a pound sign or hash character (`#`), like so:

```
#myName
```

With that in mind, making the `#myName` `id` bold looks exactly like the examples you've already seen, just substituting `#myName` for `div`:

```
#myName {  
    font-weight: bold;  
}
```



Always match the case that you use in the HTML with the case that you use in the CSS. If you use all uppercase to name the ID in the HTML, then use all uppercase in the CSS. If you use all lowercase in the HTML, use lowercase in the CSS. If you use a combination, like the example, then match that combination in the CSS.

When using IDs in HTML, it's important to realize that the ID should be used once and only once across an entire page. It's fine to use the same ID in different pages, but the ID should appear only once within a page.

We can hear your protest now: "But what if I need to apply the same style to more than one element?" That's where a CSS class comes in.

Selecting a group of elements

You've learned how to target HTML elements across a page and you've learned how to target just one individual element. A CSS class is used to select multiple elements and apply a style to them.

Unlike the selection that occurs when you select all `<div>` elements, a CSS class is applied only to the specific elements that you choose. The HTML elements don't even need to be of the same type; you can apply the same CSS class to a `<div>`, to an `` tag, and to a `<p>` element alike.

CSS comments

Within the CSS rule shown nearby, there's a comment: `/* CSS Goes Here */`. Just like in HTML where you can use comments to help explain a certain piece of code, so too can you use comments in CSS to help explain the CSS. Like HTML comments, comments in CSS are not visible in the output of the page but,

also like HTML comments, CSS comments are viewable by viewing the source of the HTML or CSS document itself. This means that visitors can see the comments too!

Comments in CSS are opened with `/*` and closed with `*/`. Everything appearing between `/*` and `*/` is treated as a comment.

Like an id, a class is applied first to the HTML elements with an attribute. The attribute is the aptly titled `class`, as in this example:



```
<div class="boldText">This text has a class.</div>
```

As in the id example, the class is also case sensitive. The case used in the HTML should match that in the CSS.

Whereas an ID selector uses a pound sign (`#`) in the CSS, a class uses a single period or dot. In the preceding example, where the class is named `boldText` in the HTML, it would be referenced like this in the CSS:

```
.boldText {  
  /* CSS Goes Here */  
}
```

In this example, the class `boldText` is selected.

Classes can be used to solve the problem discovered earlier in Figure 2-4 (in the “Using an internal style sheet” section), where the bold font was applied to all the `<div>` elements because the CSS used the `div` selector. You can use a class in the HTML to target only those elements that you want to target.

It's time to test that theory. Follow these steps.

1. **Open a text editor.**
2. **Open `css.html`.**

Open the file that you created in a previous exercise. You may have named it `css2.html`.

3. Make changes to `css.html` to remove the CSS.

The page should look like this:

```
<!doctype html>
<html>
<head>
<title>A CSS Exercise</title>
<link rel="stylesheet" type="text/css" href="style.
    css">
</head>
<body>
<div class="boldText">This is my web page.</div>

<div>
    This is the <span>nicest</span> page I've made yet.
</div>

<div class="boldText">Here is some text on my page.</
    div>

</body>
</html>
```

4. Save the file.

You can save it as `css.html` or rename it to `css3.html`. Save the file in your document root.

5. Create a new empty text file.

Using your text editor, create a new empty file.

6. Place the following CSS in the file.

```
.boldText {
    font-weight: bold;
}

span {
    font-style: italic;
}
```

7. Save the file.

Save the file as `style.css` within your document root. Note that you should ensure that the file is named with all lowercase and has the correct file extension, `.css`.

8. Open your browser and view the `css.html` file.

Type **`http://localhost/css.html`** in the browser's address bar. If you save the file as `css3.html`, then use that instead of `css.html`. The output should look like that in Figure 2-5.

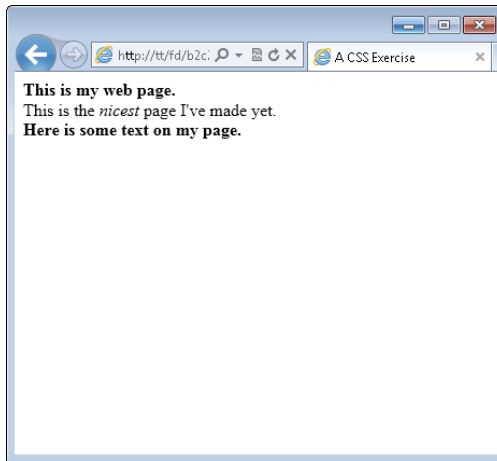


Figure 2-5:
A page with
an external
style sheet.

Notice that the page in Figure 2-5 looks exactly like Figure 2-3. That's what we hoped would happen! This exercise implemented an external style sheet and used a CSS class to target the bold `font-weight` to only those elements that we wanted to be bold.

Changing Fonts

So far you've seen a good amount of changing font weight to make fonts appear bold and a little about font styling to make the font appear in italics. However, you can do a lot more with fonts on the web using CSS, including choose a font family and select font sizes and color.

Setting the font family

The term *font family* describes the typeface or look of the font used for the text. The font family can be changed using CSS but there's a huge limitation: The fonts you use need to also be available on the visitor's computer. In practical terms, this means that you have to use certain "web friendly" fonts that appear on most visitors' computers. It also means that you can't always guarantee what font the visitor will see. If a visitor doesn't have the font that you specify, that visitor's browser chooses a substitute.

The CSS property for the font is called `font-family`. When setting a font, the best practice is to provide a list of fonts from which the browser can choose, as in this example:

```
font-family: arial, helvetica, sans-serif;
```

You can set the recommended fonts for the entire HTML page by using the selector for the `<body>` element, as in this example:

```
body {  
    font-family: arial, helvetica, sans-serif;  
}
```

Any page that uses that CSS rule will attempt to display its text first with the Arial font. If that font isn't available, the Helvetica font is used next. If that font isn't available, then a sans-serif font is used. If none of those are available, then the browser chooses a font to use all on its own.

Common values for `font-family` are

`arial, helvetica, sans-serif`

`"Arial Black", Gadget, sans-serif`

`Georgia, serif`

`"Times New Roman", Times, serif`



A concept called Web Fonts enables the use of additional fonts by allowing the browser to download the preferred fonts as part of the page. This concept is discussed at www.html5rocks.com/en/tutorials/webfonts/quick.

Listing 2-2 shows the CSS that you saw in an earlier example. This listing adds the `font-family` CSS property to the body of the page, meaning that this `font-family` setting will be applied across the entire page.

Listing 2-2: Setting the Font-Family Value with CSS

```
body {  
    font-family: arial, helvetica, sans-serif;  
}  
  
.boldText {  
    font-weight: bold;  
}  
  
span {  
    font-style: italic;  
}
```

When viewed in a browser using the same HTML from the preceding exercise, the result looks like Figure 2-6.

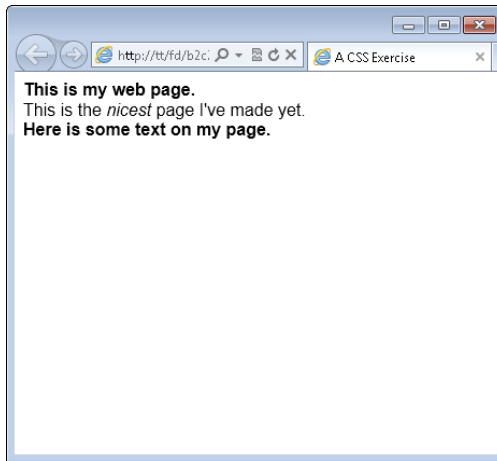


Figure 2-6:
Changing
the font
family
with CSS.

Setting font size

How large the text appears on a web page is its *font size*. You can set font sizes using the `font-size` CSS property. Font sizes can be set in one of four units:

- ◆ Percentage
- ◆ Pixels
- ◆ Points
- ◆ Em's

Which of those you should use depends largely on whom you ask. If you ask four web developers which one to use, you'll probably get four different answers. You can read the sidebar for more information, but this book uses either percentage or em's. If you're asking why, the short answer is that both of those methods work well for mobile devices and other scenarios where visitors may want to scale the text size according to their needs.

Em's are a unit of sizing for fonts, much like points that you see in a word processor.

Font sizes are set like any other CSS property; for example, this sets the font size to 150% of its normal size:

```
font-size: 150%;
```

Choosing a font sizing method

When choosing a font sizing method, you can use percentage, em's, points, and pixels. Points and pixels are fixed sizes and some browsers can have trouble resizing them, or more

appropriately, the browsers don't allow the visitor to resize the text without using a zoom tool. Percentages and em's allow resizing.

It's quite common to set a font size for the entire page and then change font sizes for individual elements in the page. For example, setting the font size for the body element — in other words, the entire page — looks like this:

```
body {  
    font-size: 90%;  
}
```

With that CSS setting, the fonts across all elements on the page would be set to 90% of their default value. You could then change individual areas of the page to have a different font size. Using em's for the other fonts allows you to change the font sizes relative to that initial setting of 90%. This allows for greater control over the page's font sizes.



Like other CSS settings, visitors can override your CSS with their own settings. They may change the font sizes according to their needs.

Listing 2-3 shows a CSS file that depicts this functionality.

Listing 2-3: CSS to Change the Font Size

```
body {  
    font-size: 90%;  
}  
  
span {  
    font-size: 1.7em;  
}
```

When combined with the HTML from the previous exercise, you get a page like that in Figure 2-7. Note the increased font size for the word *nice*st, thanks to the increased size set with an em.

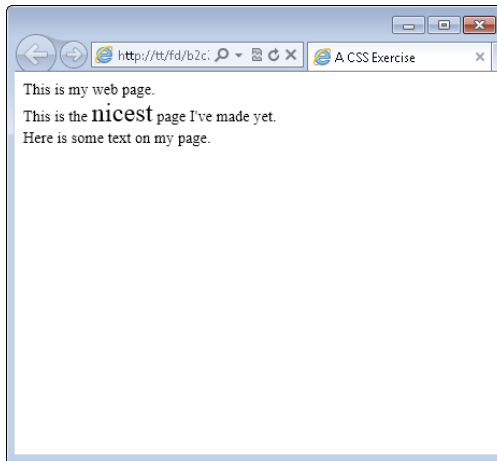


Figure 2-7:
Changing
font sizes
with CSS.



When using em's for font sizes, an em value of 1.0 corresponds to 100%. Therefore, 0.9em would be about 90%, while 1.7em (as in the example) is essentially 170%.

Fonts set with pixels or points use their abbreviations, as in these examples:

```
font-size: 12px;  
font-size: 12pt;
```

Setting the font color

Just as font sizes can be set, so too can the colors used for fonts. Care should be taken when choosing font colors so as to make the text readable. For example, using white text on a white background makes it impossible for the reader to see the text!

Just as there are multiple options for how to change the font size, there are also multiple ways to change the font color. You can use a friendly name for common colors, like red, blue, green, and so on, or you can use a hexadecimal code, or hex code for short.



Hex codes are three- to six-character codes that correspond to the Red, Green, and Blue (RGB) color mix appropriate to obtain the desired color.

Table 2-1 shows some common hex codes and their corresponding color.

Table 2-1	Hex codes for colors
<i>Code</i>	<i>Color</i>
#FF0000	Red
#00FF00	Green
#0000FF	Blue
#666666	Dark Gray
#000000	Black
#FFFFFF	White
#FFFF00	Yellow
#FFA500	Orange

Hex codes are the more accurate and preferred way to set colors in HTML but they're hard to remember. A tool like Visibone's Color Lab at www.visibone.com/colorlab is essential to obtaining the hex code corresponding to the color that you want to use.

Font color is set using the color CSS property, as in this example (which is the code for red):

```
color: #FF0000;
```

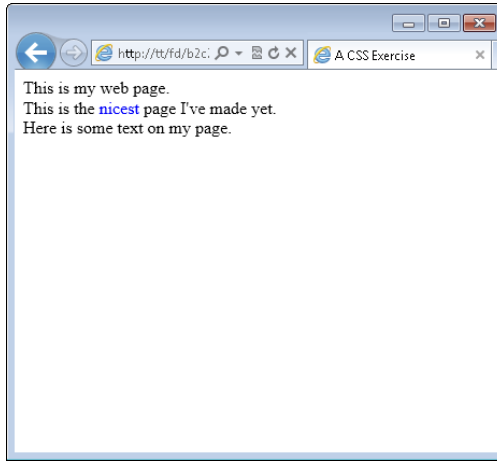
Listing 2-4 shows CSS to change colors of a `` element to blue using a hex code:

Listing 2-4: Coloring a Font Using CSS

```
span {  
    color: #0000FF;  
}
```

When viewed in a browser with the HTML created earlier in this chapter, the output looks like Figure 2-8. Note the blue coloring (which may be a bit difficult to read in this black-and-white book) for the word “nicest” on the page.

Figure 2-8:
Changing
the font
color to
blue.



Adding Borders

Borders can help provide visual separation between elements on a page. You can add borders around just about anything in HTML and there are a few border styles to choose from. Borders are added with the border CSS property.

When creating a border with CSS, you set three things:

- ◆ **Border thickness**
- ◆ **Border style**
- ◆ **Border color**

These three items are set in a list, separated by a space, as in this example:

```
border: 1px solid black;
```

In this example, a border would be created and would be 1 pixel thick. The border would be solid and would be black in color.

Some common border styles are shown in Table 2-2.

Table 2-2	Border Styles in CSS
Style	Description
Solid	A solid line
Dotted	A dotted line
Dashed	A line with a dash effect
Double	Two solid lines

It's time for an exercise to create a border around some elements. Follow these steps.

1. Open your text editor.
2. Verify the HTML file from the preceding exercise.

The HTML from the preceding exercise is the starting point for this exercise. If yours doesn't look like this, change it to look like this HTML. For those of you who had this file exactly as in the preceding exercise, the only thing you need to do is add a class called `addBorder` in the first `<div>` element.

```
<!doctype html>
<html>
<head>
<title>A CSS Exercise</title>
<link rel="stylesheet" type="text/css" href="style.
css">
</head>
<body>
<div class="boldText addBorder">This is my web page.</
div>

<div>
  This is the <span>nicest</span> page I've made yet.
</div>

<div class="boldText">Here is some text on my page.</
div>

</body>
</html>
```

3. Save the HTML file.

Save it as `css-border.html` and place it in your document root.

4. Open your CSS file.

You should have a CSS file from the preceding exercise. The CSS file from that exercise should contain a class called `boldText` and a CSS rule changing all `` elements to italic. Within your CSS file, add and change your CSS so that it looks like the following:

```
.boldText {  
    font-weight: bold;  
}  
  
span {  
    font-style: italic;  
}  
  
.addBorder {  
    border: 3px double black;  
}
```

5. Save the CSS file.

Save the file as `style.css` in your document root.

6. View the page in a browser.

Open your web browser and point to `http://localhost/css-border.html` to view the page. You should see a page like Figure 2-9.

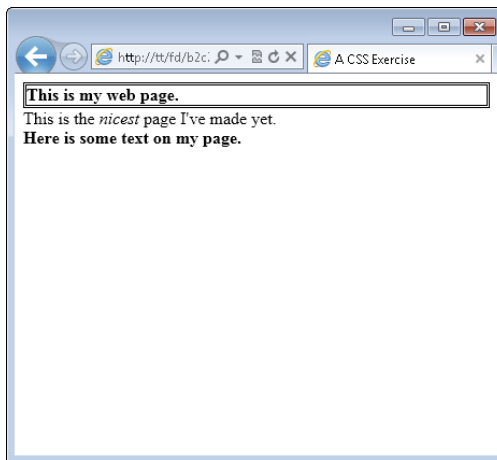


Figure 2-9:
Adding a
border to a
div element.

You may have noticed in this exercise that you now have two classes on the first `<div>` in the page. That's a great feature of classes because you can use more than one on an element to combine them.

You can experiment with the CSS from this exercise to add different styles of borders to different elements in the page.

You may not like how close the text is to the border in Figure 2-9. We sure don't. You can change this with CSS. The CSS padding property changes how close the text will come to the inside edge of the border. For example, you could change the CSS for the `addBorder` class to look like this:

```
.addBorder {  
    border: 3px double black;  
    padding: 5px;  
}
```

When you do so, the resulting page will look like that in Figure 2-10.

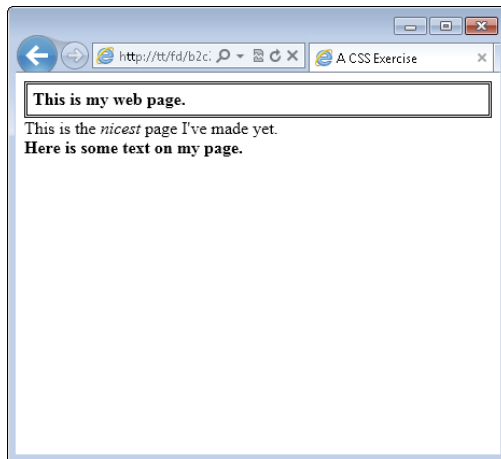


Figure 2-10:
Adding
padding
within the
`addBorder`
class.

Padding can be added to move the text farther away from its borders. Padding can be applied to any element, regardless of whether it has borders, in order to move that element's contents.

When you add padding, the contents of the element move away from all the edges. However, you can also add padding so that the contents move away from the top, bottom, right, or left, or any combination therein. This is accomplished with the `padding-top`, `padding-bottom`, `padding-right`, and `padding-left` properties, respectively.



There's a shortcut method for setting padding that sees all the padding defined on one line. That shortcut isn't used here, but you'll see it in other people's CSS.

Where padding moves elements from the inside, there's also a property to move or shift elements around from the outside. This element is called margin, and we discuss it later in the chapter when we talk about creating page layouts.

Changing List Styles

Recall the example from Chapter 1 of this minibook that created a bulleted list. That section indicated that you can change or even remove the bullets from the list using CSS. Well, it's true. You can. The bullet style for a list is determined by the `list-style-type` CSS property.

There are numerous values for the `list-style-type` property. Table 2-3 shows some common ones.

Table 2-3 Common List Styles	
<i>Style</i>	<i>Description</i>
<code>circle</code>	Provides a circle type bullet.
<code>decimal</code>	The default style for <code></code> lists, a simple number.
<code>disc</code>	The default style for <code></code> lists, a filled in circle style.
<code>none</code>	Removes styling completely for the list.
<code>square</code>	A square bullet.
<code>upper-roman</code>	An uppercase Roman numeral, as in an outline.

Changing bullet styles

The best way to see these styles in action is by trying them out. This exercise uses Listing 1-7 from the preceding chapter, and we show you all that code here in Step 3.

1. **Open your text editor.**
2. **Change or create `ul.html`.**

If you have a file called `ul.html` from the previous chapter, open it now. If you don't, you can create one now by creating a new empty text document.

Inside the file, use the following HTML. If you're using `ul.html`, then you merely need to add the `<link>` element to incorporate a CSS file.

```
<!doctype html>
<html>
<head>
<title>An unordered list</title>
<link rel="stylesheet" type="text/css" href="ul.css">
</head>
<body>
<ul>
  <li>Pine</li>
  <li>Oak</li>
  <li>Elm</li>
</ul>
</body>
</html>
```

3. **Save the file.**

Save the file as `ul.html` in your document root.

4. **Create a new file.**

Create a new empty text document using your text editor.

5. **Place the following CSS in the new document:**

```
ul {
  list-style-type: square;
}
```

6. **Save the CSS file.**

Save the file as `ul.css` in your document root.

7. Open your web browser and view the page.

In your web browser, type **http://localhost/ul.html** into the address bar and press Enter. You should see a page like the one in Figure 2-11.

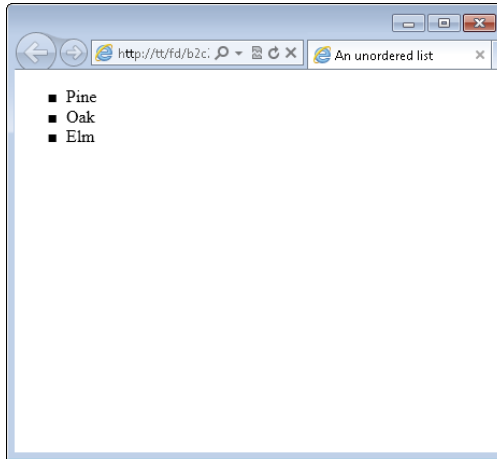


Figure 2-11:
Changing
the list style.

You can experiment with the `list-style-type` property to add or change bullet style.

Removing bullets

A common look for lists on web pages uses no bullets at all. This effect is created by setting the value of the `list-style-type` to `none`, as in this example, which can be used in the `ul.css` file you just created.

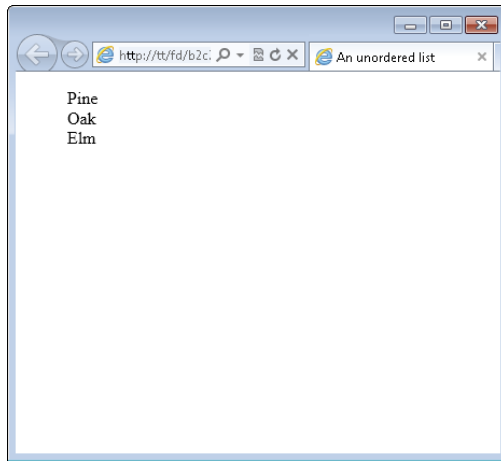
```
ul {  
    list-style-type: none;  
}
```

When applied to the page you created in the preceding exercise, the result looks like Figure 2-12.



You apply the `list-style-type` property to the `` or `` and not to the individual list items (the `` element).

Figure 2-12:
Removing
the bullets
from an
HTML list.



Adding a Background

The pages you've created so far have a white background, or more exactly, they have the default background chosen by the browser. In old versions of web browsers, that background color was gray. You can change the color of the background using CSS, or use a background image.

Background colors and background images can be applied to the entire page or to individual elements. Changing background colors on individual elements helps to add highlight and color to certain areas of the page.

Changing the background color

The background color of an HTML element is changed with the `background-color` CSS property. The background color uses the same syntax (hex code) as font colors; refer to the discussion of font colors earlier in this chapter to see hex codes for common colors.

Here's an example that changes the background color of the entire page:

```
body {  
    background-color: #FFFF00;  
}
```

Figure 2-13 shows the resulting page. Note that the yellow color won't come through very well in the book, but it's there!

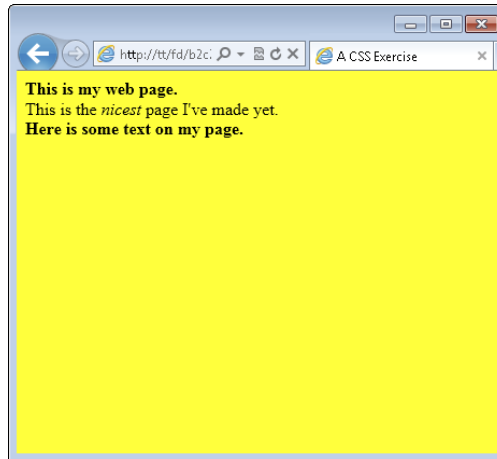


Figure 2-13:
Adding
a yellow
background
color to a
page.

As previously stated, individual elements can also be changed and you can use all the different CSS selectors to focus that color change to a class, to an individual element (using an id), or to all elements by using the element name. For example, changing all the `<div>` elements to yellow looks like this:

```
div {  
    background-color: #FFFF00;  
}
```

You can also use CSS to target elements by their hierarchy; in other words, you can target the elements when they appear as children of other elements. This calls for an example. Many of the examples in this book use HTML similar to that shown in Listing 2-5, so we use Listing 2-5 to show you how to target certain HTML elements.

Listing 2-5: HTML Used in Some Examples

```
<!doctype html>  
<html>  
<head>  
<title>A CSS Exercise</title>  
<link rel="stylesheet" type="text/css" href="style8.css">  
</head>  
<body>  
<div class="boldText">This is my web page.</div>
```

```
<div>
  This is the <span>nicest</span> page I've made yet.
</div>

<div class="boldText">Here is some text on my page.</div>

</body>
</html>
```

Focus on the `` element inside the second `<div>` in this HTML. You could say that the `` element is a child of the `<div>`. Using CSS, you can target this span by its position as a child of the `<div>`. This is helpful if you want to apply certain styling to all elements of a certain type but you don't (or can't) add a class to those elements. For example, if you wanted to make all `` elements that appear within a `<div>` to have a red background, you could do so with this CSS:

```
div span {
  background-color: #FF0000;
}
```

Applying this CSS to the CSS previously seen, including that for Figure 2-13, you get a result like Figure 2-14, which (trust us) shows the word *nicest* highlighted in red.

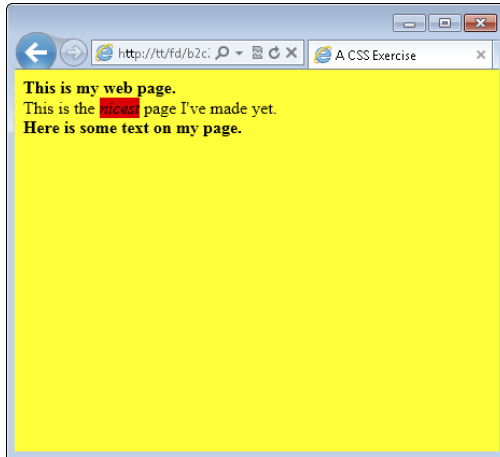


Figure 2-14:
Targeting
an element
in order to
apply a CSS
rule.

This CSS targeting can be applied in any way that you'd like, whether that's targeting a certain ID, a certain class, or certain elements, like the example does. You can create powerful (and sometimes confusing) combinations of CSS hierarchies in order to apply CSS rules.



You can use this CSS targeting to apply any CSS rule, not just background colors.

Adding a background image

Background images are a good way to create a nice looking HTML page. Using a background image, you can create a gradient effect, where one part of the page is a solid color and the color fades out or gets lighter as it stretches to the other side.

Background images appear behind other elements. This means that you can overlay all your HTML, including other images, on top of a background image.



You can find many free images through the Creative Commons. See <http://search.creativecommons.org> for more information. Be sure to choose an image that still allows for the text to be readable on the page; black text on a dark picture is not a good match.

Background images are added with the `background-image` CSS property, as described here and in the following sections.

```
background-image:url ("myImage.jpg") ;
```

Adding a single background image

One of the features of background images is that you can tile or repeat them within a page. This means that no matter how large the visitor's screen, the background image will always appear. Conversely, you can also choose to not repeat the background image. This section shows how to add a single, non-repeating image.

In order to complete this exercise, you need an image. The image will preferably be at least 800 pixels by 600 pixels. You can find out the image resolution by right-clicking the image and selecting Properties in Windows or choosing Get Info from the Finder window on a Mac.

1. Open your text editor.

Create a new empty text document in your text editor.

2. In the text editor, enter the following HTML:

```
<!doctype html>
<html>
<head>
<title>Background Image</title>
<link rel="stylesheet" type="text/css"
```

```
        href="image-style.css">
</head>
<body>
</body>
</html>
```

3. Save the file.

Save the file as `backimage.html` in your document root.

4. Create a new text document.

Create a new empty text document with your text editor.

5. Place the following CSS in the new document.

Be sure to use the name of your image. In this example, we're using an image called `large-snow.jpg`. The image should be saved within your document root.

```
body {
    background-image:url("large-snow.jpg");
    background-repeat: no-repeat;
}
```

6. Save the CSS file.

Save the file as `image-style.css` and make sure it's saved within your document root.

7. Open your web browser and view the page.

Open your web browser and navigate to the page at `http://localhost/backimage.html`. You'll see the page with a background image. You can see a screenshot of our page, with the `large-snow.jpg` image, in Figure 2-15.

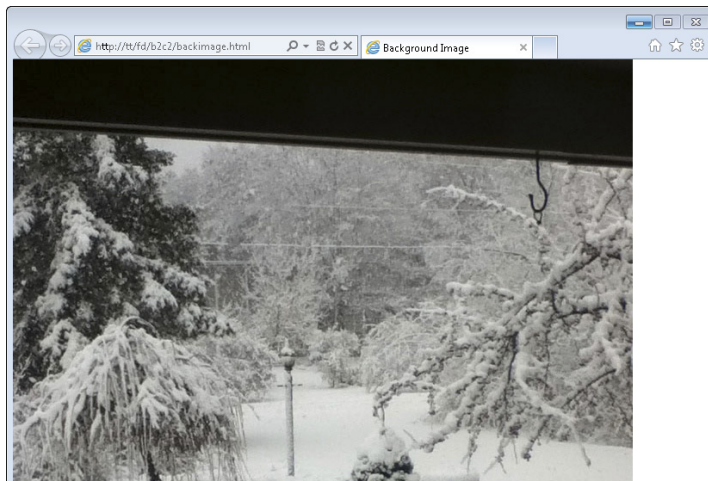


Figure 2-15:
A single
background
image.

Depending on the size of your image and your screen, you may notice that the image ends, as it does along the right side of Figure 2-15. Additionally, you may notice that the image isn't centered. Keep reading for a solution.

Improving the single background image page

A common approach used to create a better looking page is to add a background color that matches the edges of the image. In the case of our image, the top and bottom are black. Therefore, we could add a rule to the CSS to make the default background color black. This won't have any effect where the image is located — the image takes precedence — but it will matter along the bottom where the image ends.

The CSS for this look is as follows:

```
body {  
    background-image:url("large-snow.jpg");  
    background-repeat: no-repeat;  
    background-color: #000000;  
}
```

With that rule in place, the image will still end but the appearance won't be quite as shocking or noticeable because it matches the color of the edge of the image, as shown in Figure 2-16.

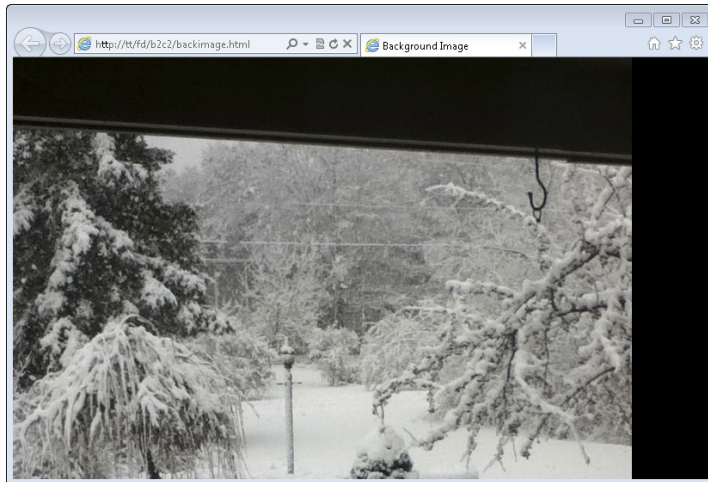


Figure 2-16:
Adding a
background
color and a
background
image.

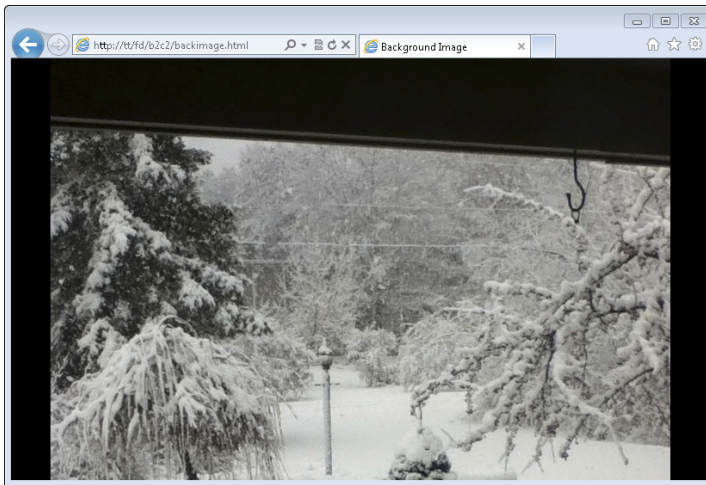
While the background color trick solves the problem with the edge of the image, it doesn't solve the centering issue. The current background image is applied to the body — in other words, the entire page. In order to center the background image, another CSS property needs to be added, as shown in this CSS:

```
body {  
    background-image:url("large-snow.jpg");  
    background-repeat: no-repeat;  
    background-color: #000000;  
    background-position: center top;  
}
```

This CSS adds the `background-position` rule and places it at the center at the top of the page. Other values include left, right, and bottom, and you can combine them so that the background image would appear at the bottom right, for example.

The CSS shown here places the image at the center of the page and at the top. This results in the page shown in Figure 2-17.

Figure 2-17:
A centered
background
image on
the top,
with a
background
color.



With that image in place, you can then add any HTML to the page that you see fit. Note with an image like this (a dark top and light middle) you need to adjust the font colors so that the text is visible on the page.

Adding a repeating image

You can add an image that repeats. This is a common scenario for web pages because then the image doesn't end along the sides, no matter how large your resolution is. This also alleviates the need for a background position because the background image applies to the entire element.



When applied to the entire page, as in the examples shown, you can also forego the `background-repeat` rule and the background color because the image continues throughout the entire page.

An ideal repeating image is one that doesn't have noticeable borders because those borders will show up when the image is tiled or repeated on the page.

Figure 2-18 shows a small image (15 pixels x 15 pixels) used as a repeating image with the following CSS:

```
body {  
    background-image:url("back.jpg");  
}
```

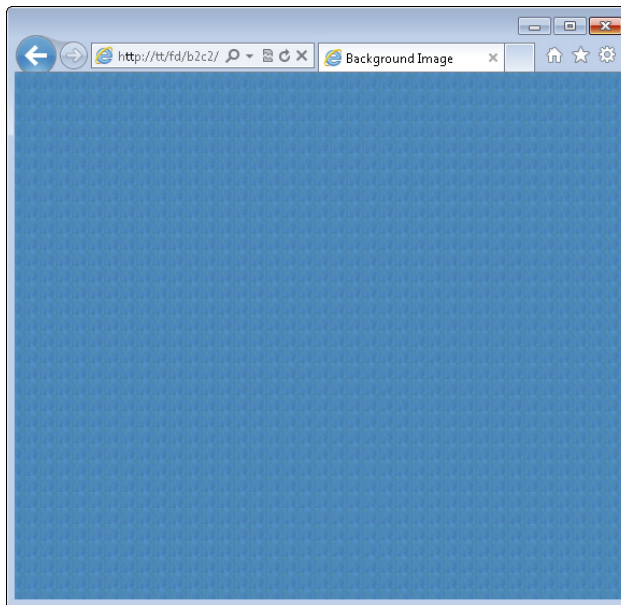


Figure 2-18:
A repeating
background
image.

As in the example for a single image background, you can now add HTML atop the background, again choosing a font color that offsets the image so that visitors can easily read the text.

Creating Page Layouts

You've now learned a good amount of CSS to change the behavior and appearance of individual items, add background colors, style lists, and so on. All of this leads to creating pages by using CSS. CSS is used to create more complex appearances for web pages than you've seen so far. For example, you can create column effects, where there's a menu on the left or right side and content in the other column, and we tell you how to do that here.



When working with alignment and column layouts, it's sometimes helpful to add a border to the element to see where it begins and ends so that you can see how the layout looks.

Creating a single-column layout

Everything you've seen so far has been a single-column layout. There's only one column, aligned on the left of the page. You can, however, control that alignment with CSS. Doing so means creating more complex HTML than you've seen so far but nothing in the HTML will be new; there'll just be more HTML than before.

1. Open a text editor.

Open your text editor and create a new empty document.

2. Within the empty document, enter the following HTML:

```
<!doctype html>
<html>
<head>
<title>Single Column</title>
<link rel="stylesheet" type="text/css" href="single.
css">
</head>
<body>
<div id="container">
  <div id="content">
    <h2>Here's some content</h2>
    <p>This is where a story would go</p>
    <h2>Here's more content</h2>
    <p>This is another story</p>
  </div> <!-- end content -->
</div> <!-- end container -->
</body>
</html>
```

3. Save the file.

Save the file as `single.html` in your document root.

4. Open your browser and view the page.

View the page by going to `http://localhost/single.html` in your browser. You'll see a page similar to that in Figure 2-19.

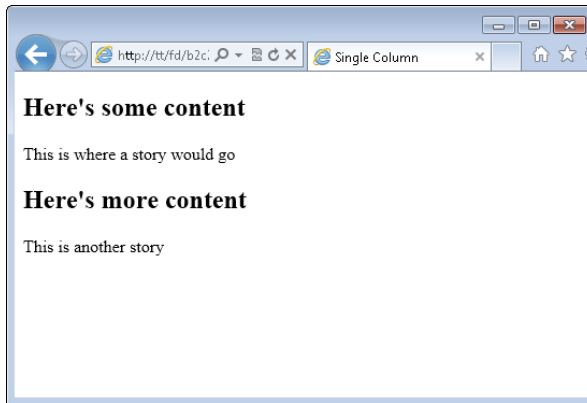


Figure 2-19:
A basic
page before
adding CSS.

5. Create a new text document.

Create a new empty text document in your text editor.

6. In the document, place the following CSS:

```
body {
    font-family: arial,helvetica,sans-serif;
}

#container {
    margin: 0 auto;
    width: 600px;
    background: #FFFFFF;
}

#content {
    clear: left;
    padding: 20px;
}
```

7. Save the CSS file.

Save the file as `single.css` in your document root.

8. Open your web browser.

Navigate to `http://localhost/single.html` in your browser. If your browser is still open, reload the page with `Ctrl+R` on Windows or `Command+R` on Mac. You'll see a page like that in Figure 2-20. See the paragraphs that follow for more information on what specific modifications you made in Step 6.

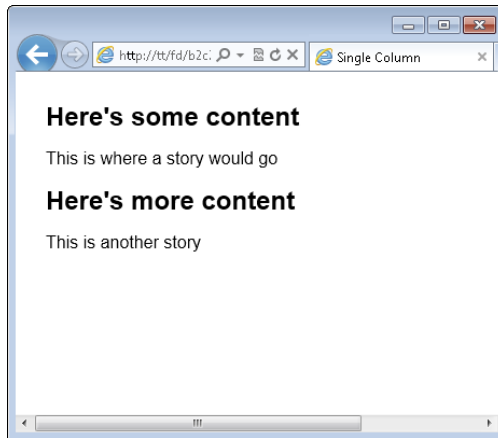


Figure 2-20:
A single-
column
layout.

Later in this chapter, you see how to add a header and footer onto this layout in order to improve its look and functionality.

The HTML for this layout uses a `<div>` element as a container. The container helps to create the layout and doesn't hold any text content of its own. The CSS for this exercise uses three CSS properties that might be new to you: `width`, `margin`, and `clear`. Here's how they work:

- ◆ **width:** Sets the horizontal width of an element. In this case, the container is set to 600px (pixels) wide. No matter how small the browser window is, your HTML will never get smaller than 600px.
- ◆ **margin:** This is the complement to the `padding` property shown earlier in this chapter, in the “Adding Borders” section. The `margin` property defines the spacing on the outside of the element. In the case shown here (`margin: 0 auto;`), the shortcut method is used. See the sidebar for more information. The value “auto” means that the browser will choose the value.
- ◆ **clear:** Makes it so that no elements can appear on the side of the element to which the rule applies. In the example, `clear left` was used on the `<div>` with the id of `#content`. This means that nothing could appear on the left side of that element. Other values for `clear` include “both,” “none,” “right,” and “inherit.”

You can experiment with the margins of your browser window to see how the layout created in the exercise reacts or moves along with the browser.

The layout created in this section is called a single-column fixed-width layout. Another option is a single-column liquid layout. A liquid layout can work better in certain devices. The fixed-width layout shown can sometimes result in a horizontal scroll bar at the bottom of the page.

To change the layout to a liquid layout, you only need to change a small amount of CSS in the #container, as shown here:

```
body {  
    font-family: arial,helvetica,sans-serif;  
}  
  
#container {  
    margin: 0 30px;  
    background: #FFFFFF;  
}  
  
#content {  
    clear: left;  
    padding: 20px;  
}
```

Note the only changes are to remove the width property within the #container and also change the margin from “0 auto” to “0 30px.” With that, the layout becomes a liquid layout and works better, especially in mobile devices.

Shortcuts for margin and padding

Rather than defining a rule for each of the top, bottom, right, and left elements of margin or padding, you can use a shortcut method that defines all of them on one line. For example:

margin: 0px 50px 200px 300px;
is equivalent to this:

```
margin-top: 0px;  
margin-right: 50px;  
margin-bottom: 200px;  
margin-left: 300px;
```

When four numbers appear in the rule, the order is top, right, bottom, and left. To help remember the order, use the mnemonic “TrouBLE,” which takes the first letter of each of the Top, Right, Bottom, Left, and makes them

into a word to remind you how much trouble it is remembering the order.

Instead of all four values, you sometimes see one, two, or three of the values present for margin or padding, as in the example shown earlier:

```
margin: 0 auto;
```

When two values are used, the first value corresponds to the top and bottom and the second value corresponds to the right and left. When three values are used, the first is the top, the second is the left and right, and the last is the bottom. Finally, when one value is used, it applies equally to the top, right, bottom, and left.

Creating a two-column layout

A two-column layout uses a bit more HTML to achieve the effect of multiple columns. This is frequently done to add a menu along the side of a page or links to other stories or content.

Listing 2-6 shows the HTML involved for a two-column fixed-width layout.

Listing 2-6: A Two-Column Fixed-Width Layout

```
<!doctype html>
<html>
<head>
<title>Two Column</title>
<link rel="stylesheet" type="text/css" href="double.css">
</head>
<body>
<div id="container">
  <div id="mainContainer">
    <div id="content">
      <h2>Here's some content</h2>
      <p>This is where a story would go</p>
      <h2>Here's more content</h2>
      <p>This is another story</p>
    </div> <!-- end content -->
    <div id="sidebar">
      <h3>Menu</h3>
      <ul>
        <li>Menu item 1</li>
        <li>Menu item 2</li>
        <li>Menu item 3</li>
      </ul>
    </div> <!-- end sidebar -->
  </div> <!-- end mainContainer -->
</div> <!-- end container -->
</body>
</html>
```

This HTML uses the container `<div>` from the single-column layout and adds another container `<div>` to hold the content. That `<div>`, called `mainContainer`, holds both the content and the sidebar. The other addition is the sidebar itself, aptly titled `sidebar`. That sidebar holds a menu with an unordered list (``) in it.

The CSS for the two-column layout is shown in Listing 2-7.

Listing 2-7: CSS for a Two-Column Fixed-Width Layout

```
#container {
    margin: 0 auto;
    width: 900px;
}

#mainContainer {
    float: left;
    width: 900px;
}

#content {
    clear: left;
    float: left;
    width: 500px;
    padding: 20px 0;
    margin: 0 0 0 30px;
    display: inline;
}

#sidebar {
    float: right;
    width: 260px;
    padding: 20px 0;
    margin: 0 20px 0 0;
    display: inline;
    background-color: #CCCCCC;
}
```

This CSS uses several of the items that you've seen already, including margin, padding, clear, and background-color, among others. New to this CSS are the float and the display properties.

The float property defines whether an element will move or float within a layout, either to the left or to the right or whether it won't float at all (none), as is the default. However, because you want to create two columns next to each other, you need to float the content container to the left and the sidebar to the right. Therefore, if you want the sidebar to appear on the right, you simply need to swap float: left in the #content CSS with the float: right found in the #sidebar's CSS.

The display property sets how the element should be displayed. Certain elements are known as *block-level* elements and display the entire width of the page. The <div> element is a good example of this. Because you want to make the columns appear next to each other, you need to change this block display behavior to inline (we introduce inline elements in the preceding chapter), so that the element doesn't extend the full width of the page.

Hiding elements

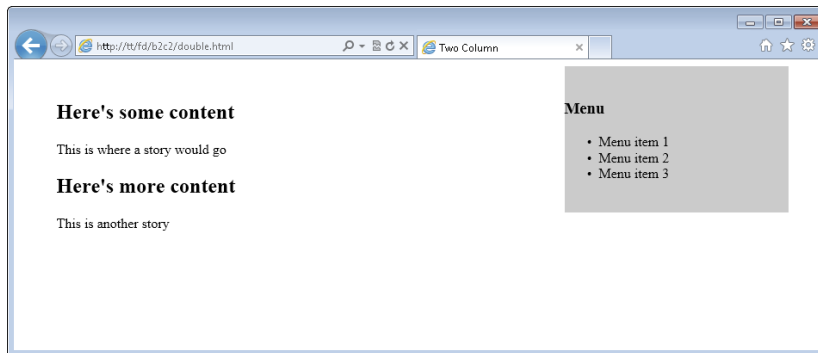
Setting the CSS display property to `none` hides an element from a page. When you do so, the element is removed entirely from the page. You can also use another CSS property, `visibility`, to hide elements. When hiding an element with the `visibility` property

(`visibility: hidden;`), the box or area on the page still remains in place but the element becomes invisible. Making the element visible again (`visibility: visible;`) shows the element.

Three frequently used values for the display property are `block` (to extend the element the full width), `inline` (to make the element use only its own width for display), and `none` (which removes the element from display entirely).

When viewed in a browser, the layout shown in Listings 2-6 and 2-7 produces a page like that in Figure 2-21.

Figure 2-21:
A two-column fixed-width layout.



The layout shown in Figure 2-21 is a fixed-width layout. Converting this to a liquid layout means changing the `width` and `margin` values in the CSS from pixels (px) to percentages (%). The CSS to convert into a liquid layout is shown in Listing 2-8.

Listing 2-8: Converting to a Two-Column Liquid Layout

```
#container {
    margin: 0 auto;
```

```

        width: 100%;
    }

    #mainContainer {
        float: left;
        width: 100%;
    }

    #content {
        clear: left;
        float: left;
        width: 65%;
        padding: 20px 0;
        margin: 0 0 0 5%;
        display: inline;
    }

    #sidebar {
        float: right;
        width: 20%;
        padding: 20px 0;
        margin: 0 2% 0 0;
        display: inline;
        background-color: #CCCCCC;
    }

```

The changes occur in the `#container`, `#mainContainer`, `#content`, and `#sidebar` sections, to change the previous values that used pixels to percentages. This layout now changes with the width of the browser, as shown in Figure 2-22, where you'll notice that the width of the browser is much smaller.

Figure 2-22:
Creating a
liquid layout
with two
columns.



Adding Headers and Footers to a Page

The layouts you've seen so far provide a good base from which you can build a more complex web page and indeed website. However, the page is missing two things: a header and a footer.

Headers are typically used to convey information such as the name of the site or to provide a menu; footers are used to provide additional information such as copyright and are also being used to provide a map of links within a site, known as a *site map*. Additionally, we tell you how to create a menu within the header.

Creating a header, header menu, and footer

You've seen how to create a multi-column layout with a main content area and a sidebar. To create this layout, you add a `<div>` element to hold the sidebar's content. You then apply CSS rules to the `<div>` to set its width and position. Creating a header and footer is accomplished in largely the same manner. An additional `<div>` is created to hold the content for each and then rules are applied to those `<div>` elements to position them.

This being the last example in the chapter, it serves as a capstone exercise.

1. Open your text editor.

Create a new blank text document.

2. Enter the following HTML in the text document:

```
<!doctype html>
<html>
<head>
<title>Two Column With Header and Footer</title>
<link rel="stylesheet" type="text/css" href="final.
css">
</head>
<body>
<div id="container">
  <div id="header">
    <h1>This is my site!</h1>
  </div> <!-- end header -->
  <div id="menu">
    <ul>
      <li><a href="#">Home</a></li>
      <li><A href="#">Services</a></li>
```

```

        <li><a href="#">About Me</a></li>
        <li><a href="#">Contact Me</a></li>
    </ul>
</div> <!-- end menu -->
<div id="mainContainer">
    <div id="content">
        <h2>Here's some content</h2>
        <p>This is where a story would go</p>
        <h2>Here's more content</h2>
        <p>This is another story</p>
    </div> <!-- end content -->
    <div id="sidebar">
        <h3>Menu</h3>
        <ul>
            <li>Menu item 1</li>
            <li>Menu item 2</li>
            <li>Menu item 3</li>
        </ul>
    </div> <!-- end sidebar -->
    <div id="footer">
        <p>Copyright (c) 2012 Steve Suehring</p>
    </div> <!-- end footer -->
</div> <!-- end mainContainer -->
</div> <!-- end container -->
</body>
</html>

```

3. Save the file.

Save the file as `final.html` in your document root.

4. Create a new text document.

Create a new empty text document. This one should hold the following CSS:

```

body {
    font-family: arial,helvetica,sans-serif;
}

#container {
    margin: 0 auto;
    width: 100%;
}

#header {
    background-color: #abacab;
    padding: 10px;
}

#menu {
    float: left;
    width: 100%;
}

```

```
        background-color: #0c0c0c;
    }

    #menu ul li {
        list-style-type: none;
        display: inline;
    }

    #menu li a {
        display: block;
        text-decoration: none;
        border-right: 2px solid #FFFFFF;
        padding: 3px 10px;
        float: left;
        color: #FFFFFF;
    }

    #menu li a:hover {
        background-color: #CCCCCC;
    }

    #mainContainer {
        float: left;
        width: 100%;
    }

    #content {
        clear: left;
        float: left;
        width: 65%;
        padding: 20px 0;
        margin: 0 0 0 5%;
        display: inline;
    }

    #sidebar {
        float: right;
        width: 30%;
        padding: 20px 0;
        margin: 0;
        display: inline;
        background-color: #CCCCCC;
    }

    #footer {
        clear: left;
        background-color: #CCCCCC;
        text-align: center;
        padding: 20px;
        height: 1%;
    }
```

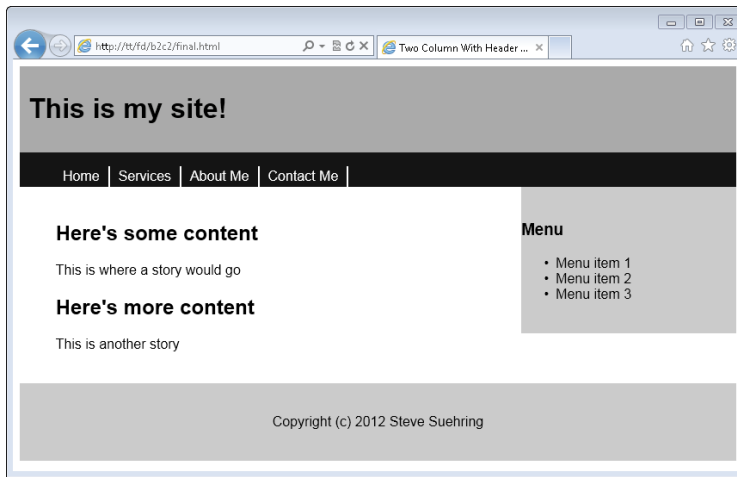
5. Save the file.

Save the CSS file as `final.css` in your document root.

6. Open your browser and view the page.

Open your web browser, navigate to `http://localhost/final.html`, and you'll see the page, like the one shown in Figure 2-23.

Figure 2-23:
A two-column liquid layout with a header and footer.



Examining the HTML and CSS files

To create this layout, you use a more complex HTML file than you've used before but there isn't anything in that file that you haven't already seen. It's just longer in order to create the additional HTML and content for the page!

The CSS does use some additional items, specifically to create the menu or links across the top. Note that this is separate from the contextual menu that appears on the right. The menu created for this page appears in the header and provides links to the areas of the site, such as Home, Services, About Me, and Contact Me.

The CSS for that section looks like this:

```
#menu ul li {
    list-style-type: none;
    display: inline;
}
```

That section uses a hierarchical structure to target only the `` elements within the `#menu` area. The `list-style-type` was set to `none`, which you saw earlier in the chapter. However, the `display` was set to `inline`. When used with lists, it makes the lists flow horizontally rather than vertically, so you get the desired effect here.

The next section of CSS changed the behavior of the `<a>` elements within that menu and was again targeted using `#menu li a` so that the CSS rule applied only to those specific `<a>` elements.

```
#menu li a {
    display: block;
    text-decoration: none;
    border-right: 2px solid #FFFFFF;
    padding: 3px 10px;
    float: left;
    color: #FFFFFF;
}
```

This CSS rule uses the standard `float`, `display`, and `border` properties explained earlier in this chapter. Added here is a `text-decoration` CSS property, which changes the default behavior of the `<a>` link. Rather than being underlined and colored, changing the `text-decoration` to `none` removes that effect, giving the menu a cleaner look.

The final piece of the menu's CSS is this:

```
#menu li a:hover {
    background-color: #CCCCCC;
}
```

This CSS rule targets the hover behavior of the `<a>` element. When the visitor hovers over the link, it will change color, in this case to `#CCCCCC`, which is a shade of gray.

Chapter 3: Creating and Styling Web Forms

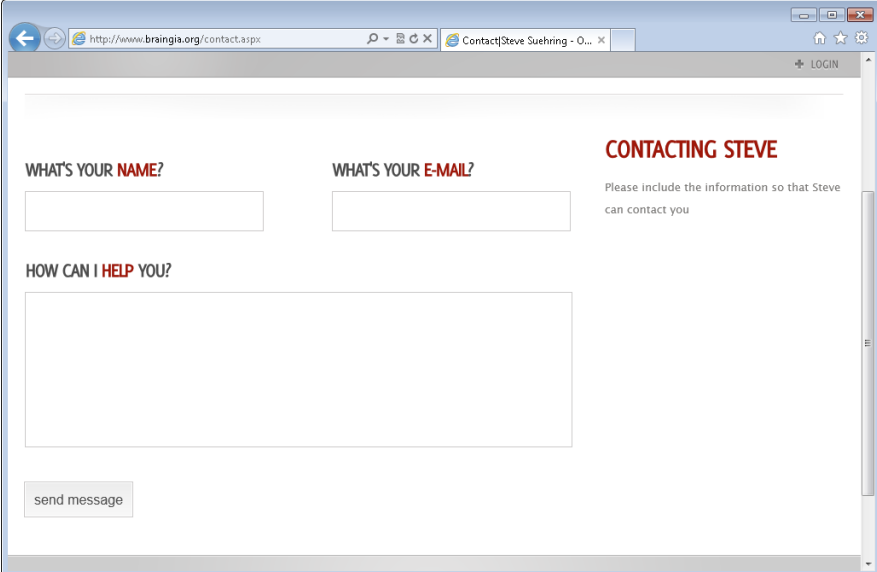
In This Chapter

- ✓ Using web forms to get information
- ✓ Creating a form
- ✓ Using CSS to style a form

Web forms enable your site to gather information from users. This chapter discusses web forms in all their glory and shows you how to both create a form and how to style it with CSS.

Using Web Forms to Get Information

With web forms, like the one shown in Figure 3-1, you can gather information from users.



The screenshot shows a web browser window with the address bar displaying `http://www.braingia.org/contact.aspx`. The page content includes a form with the following elements:

- A header section titled **CONTACTING STEVE** in red text, followed by the instruction: "Please include the information so that Steve can contact you".
- A label **WHAT'S YOUR NAME?** in red text, followed by a single-line text input field.
- A label **WHAT'S YOUR E-MAIL?** in red text, followed by a single-line text input field.
- A label **HOW CAN I HELP YOU?** in red text, followed by a large multi-line text area.
- A "send message" button located at the bottom left of the form.
- A "LOGIN" link in the top right corner of the page.

Figure 3-1:
A basic
web form.

Web forms can collect anything from name and e-mail address and a message, like the one shown in Figure 3-1, to images and files from your computer. For instance, when you log in to your web-based e-mail account like Gmail, you're filling out a form with your username and your password. Here's a look at how you can use HTML to create web forms.

Understanding web forms

When you fill out a form, the information is sent to the web server. What exactly the web server does with the information is up to the programs running on the server. For example, when you fill out the contact form on my website, the server e-mails the information e-mailed to me, but when you fill out a form to find hotel rooms on a hotel's website, the server looks in its database for matching rooms based on the dates that you fill out. In Book VI, you work with server-side programs to process web forms. For now, focus on the forms themselves.

In HTML terms, forms are created with the `<form>` element. Forms open with `<form>` and close with `</form>`, as in this example:

```
<form action="#">
<input type="text" name="emailaddress">
<input type="submit" name="submit">
</form>
```

You see how to create your own form in the next section.

Looking at form elements

There are many ways to get input through a form, each with its own specific name or type of input. The code example in the preceding section includes two `input` types: a `text` type and a `submit` type. The `text` type creates a box where the users can enter information. The `submit` type creates a button that users use to send the information to the server.

There are many other types of input elements in a form, including these:

- ◆ **Drop-down or select:** Creates a drop-down box with multiple choices from which the user can pick one.
- ◆ **Check boxes:** Creates one or more boxes that the user can select.
- ◆ **Radio buttons:** Creates one or more small buttons, of which the user can select only one.
- ◆ **Others:** There are other specialty types — including password, text area, and file — that enable you to gather other types of input from the user.

You've already seen the basic form elements, but there's more to creating forms than just adding elements. Forms need to be integrated with other HTML in order to display like you want them to. Beyond that, as you see later in the chapter, you can also style forms with Cascading Style Sheets (CSS). But for now, work on building a simple form.

Figure 3-2 shows a web form using two text input types.

The screenshot shows a web browser window with the address bar displaying 'http://tt/fd/b2c/'. The page title is 'A Basic Form'. The form itself has a title 'A Basic Form' followed by a horizontal line. Below the line is a legend 'Form Information'. Under the legend, there are two text input fields: 'Name:' and 'E-mail Address'.

Figure 3-2:
A basic web
form with
two inputs.

The HTML used to create this form is shown here:

```
<!doctype html>
<html>
<head>
<title>A Basic Form</title>
</head>
<body>
<h1>A Basic Form</h1>
<hr>
<form action="#">
<fieldset>
  <legend>Form Information</legend>
  <div>
    <label for="username">Name:</label>
    <input id="username" type="text" name="username">
  </div>
  <div>
    <label for="email">E-mail Address:</label>
    <input id="email" type="text" name="email">
  </div>
</fieldset>
</form>
</body>
</html>
```

Up until the first `<form>` tag, the HTML is all stuff you've already seen earlier in this book. The form begins with that opening `<form>` tag. When you create a form, you use two attributes fairly often, one of which is the action attribute. The action attribute tells the form where to go or what to do when the user clicks Submit. You see another attribute, method, a little later.

The next element found in the form is `<fieldset>`, which is optional for a form. The `<fieldset>` element is used primarily for layout and accessibility. The next element found is the `<legend>` element. This element creates the Form Information legend and the box that (though difficult to see in the screenshot) surrounds the inputs in the form. Like `<fieldset>`, the `<legend>` element is entirely optional.

Next in the form are the `<div>` elements used to create each row of inputs. The `<label>` element ties the friendly name — what you see on the screen, in this case, Name — to the actual input. The `<label>` element is optional but recommended because it helps with assistive technologies. Below the `<label>` element you see an `<input>` element. This `<div>`, `<label>`, `<input>` structure is repeated for the E-mail Address field.

Creating a Form

With some understanding of how forms are structured, it's time to look at creating one with some of the elements already discussed. In this section, you find out more about the `<form>` element and how to create text boxes, drop-down boxes, check boxes, and radio buttons that visitors to your website can use to enter information. You also find out how to create a Submit button, which lets visitors indicate that they're ready to transmit that information to you.

All about the form element

You already saw that the `<form>` element commonly uses a couple different attributes, `action` and `method`. The action of a form typically points to the server program that will handle the input from the form. It's where the form sends its data.

If the action tells the form where to send the data, then the `method` attribute tells the form how to send the data to the server. There are two primary methods that you'll encounter: GET and POST. The GET method is appropriate for small forms, whereas the POST method is appropriate for larger forms or ones that need to send a lot of information.

Knowing the difference in the GET and POST methods

When you use a `GET` method, the form's contents are sent as part of the URL. In the sample form that you saw earlier, the URL would end up being something like:

```
http://localhost/form1.html?
username=Steve&email=
steve@example.com
```

The first thing you notice is that a user can easily see all the form elements, including their names and values, right in their browser's address bar. Beyond that, though, there's a practical limitation in just how long that URL

can get. Many browsers, like Internet Explorer, only allow a certain number of characters in the URL, so if your form or the data being sent is too long, then it won't work.

When you use a `POST`, there's no such length restriction set by the browser. It's important to note, though, that the user can still see the form's data and how it will be sent to the server; you can't hide that from the user no matter which method you use.

For most forms, I use `POST` unless there's a specific reason to use the `GET` method.

When the action is set, as you've seen, to the pound sign or hash mark (`#`), the form essentially goes nowhere and does nothing, which for now is exactly what you want because you haven't built a server program to work with the incoming data yet!

Adding a text input

You've already seen text inputs in this chapter. Adding one is as simple as using the type of `"text"`. You can also add a couple more handy attributes, `size` and `maxlength`, which tell the browser how large to make the text box on the screen and the maximum amount of characters that are allowed in the field.

For example:

```
<input type="text" name="username" size="20" maxlength="30">
```

This HTML creates a 20-characters-wide input box, and the most that someone could enter into the box is 30 characters.

Another attribute that you might see is the `value` attribute, which prepopulates the field with the value you provide. Consider this example HTML:

```
<input type="text" name="username" value="Username Here">
```

Adding that to the form from Figure 3-2 results in a form like the one shown in Figure 3-3. Notice the value in the Name field is now set according to the value property in the `<input>` definition.

Figure 3-3:
Adding a
value to a
field.

Adding a drop-down box

A drop-down box, also known as a select box, presents many options, from which the user can select one. An example is a list of states, such as Alaska, California, Wisconsin, and so on, where the user typically chooses one from among the list. The drop-down box provides a good way to display that information. You create a drop-down using the `<select>` element along with `<option>` elements, like this:

```
<select name="state">
  <option value="CA">California</option>
  <option value="WI">Wisconsin</option>
</select>
```

Here's a full form with a drop-down added to it:

```
<!doctype html>
<html>
<head>
<title>A Basic Select</title>
</head>
<body>
<h1>A Basic Select</h1>
<hr>
<form action="#">
<fieldset>
  <legend>Form Information</legend>
```

```
<div>
  <label for="state">State:</label>
  <select id="state" name="state">
    <option value="CA">California</option>
    <option value="WI">Wisconsin</option>
  </select>
</div>
</fieldset>
</form>
</body>
</html>
```

When it's viewed in a browser, you get a page like that shown in Figure 3-4.

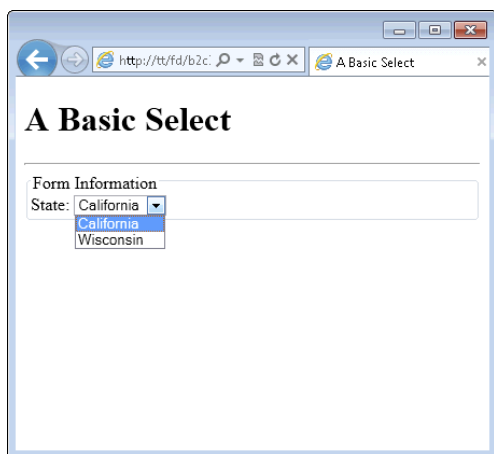


Figure 3-4:
Creating a
select drop-
down box.

When a drop-down box is displayed, the first element is the one that shows up as the default. In the example shown in Figure 3-4, California is displayed as the default option. You can, however, change the default value in two different ways, as discussed here.

Like text boxes, you can set a default value for a drop-down box. This is accomplished using the `selected` attribute. Though not always required, it's a good idea to set a value for the `selected` attribute, as in this example that would change the default value to Wisconsin:

```
<select name="state">
  <option value="CA">California</option>
  <option selected="selected" value="WI">Wisconsin</option>
</select>
```

Another way to set a default value of sorts is to set a blank option as the first option in the list. While this isn't technically a default value, it shows

up first on the list so it'll show as the default option when a user loads the page. A common way you'll see this is to use "Select a value" or similar wording as the first option, indicating to the user that there's some action required, as shown here and in Figure 3-5.

```
<select name="state">
  <option value="">Select a value...</option>
  <option value="CA">California</option>
  <option value="WI">Wisconsin</option>
</select>
```

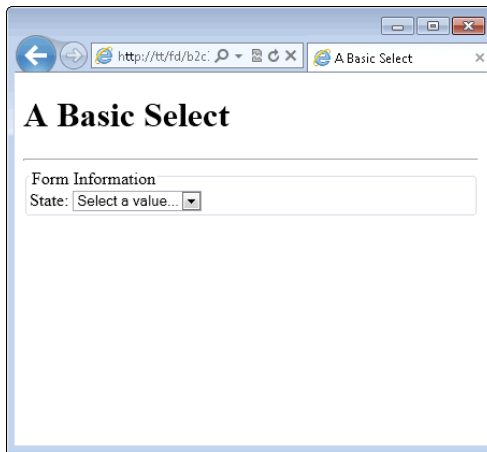


Figure 3-5:
Setting the
first value
for a
drop-down.



Using the `selected` attribute overrides the first value trick shown in this example.

Creating check boxes

Another way to represent multiple values is by using check boxes. Where drop-downs are good to represent multiple values when there are a lot of options, check boxes are good to represent multiple values when there are just a few options, as might be the case when building a form for choosing pizza toppings. When someone adds pizza toppings, she can choose more than one on her pizza, but there usually aren't too many toppings.

```
<!doctype html>
<html>
<head>
<title>Checkboxes</title>
</head>
<body>
<h1>Checkboxes</h1>
<hr>
<form action="#">
```

```
<fieldset>
  <legend>Pizza Information</legend>
  <div>Toppings: <br />
    <input type="checkbox" id="sausage"
      name="toppings" value="sausage">
    <label for="sausage">Sausage</label><br />
    <input type="checkbox" id="pep"
      name="toppings" value="pep">
    <label for="pep">Pepperoni</label><br />
    <input type="checkbox" id="mush"
      name="toppings" value="mush">
    <label for="mush">Mushrooms</label><br />
  </div>
</fieldset>
</form>
</body>
</html>
```

This HTML creates three check boxes in a group called "toppings". The resulting page is shown in Figure 3-6.

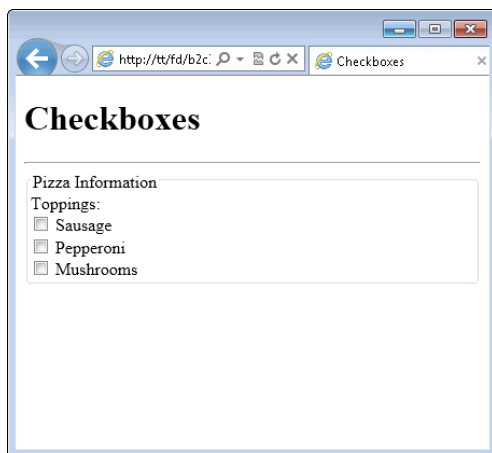


Figure 3-6:
Using check
boxes for
input.

Notice in the HTML that each check box has the same `name` attribute but uses different `value` attributes and different `id` attributes. The `id` attributes need to be unique in order for the HTML to be valid (and for the labels to work correctly). The name is the same because the check boxes are actually grouped together; they represent one type of information: pizza toppings.

In practice, you may see check boxes without `name` attributes or with a different `name` attribute for each check box. The example you see here is one that keeps the information logically grouped, which makes it easier to maintain later and also makes it easier to work with in a server program, as you see later in this book.

Using radio buttons

Radio buttons are used where there are multiple values but the user can choose only one from among those options, as would be the case with a type of crust for a pizza. The crust can be thin or deep dish — but not both or the pizza would be a complete mess.

Here's the HTML to create radio buttons. Notice that the HTML isn't really all that much different than the check box example:

```

<!doctype html>
<html>
<head>
<title>Radio</title>
</head>
<body>
<h1>Radio</h1>
<hr>
<form action="#">
<fieldset>
  <legend>Pizza Information</legend>
  <div>Crust: <br />
    <input type="radio" id="deep"
      name="crust" value="deep">
    <label for="deep">Deep Dish</label><br />
    <input type="radio" id="thin"
      name="crust" value="thin">
    <label for="thin">Thin</label><br />
  </div>
</fieldset>
</form>
</body>
</html>

```

When viewed in a browser, the result is like that in Figure 3-7.

Like check boxes, radio buttons have the same name but use different value and id attributes. Like check boxes, radio buttons use these values for the same reasons. With radio buttons, the name attribute is even more crucial. Radio buttons that share the same name attribute are in the same group, meaning the user can choose only one of the options in that group. If you want the user to be able to choose more than one option, then you should probably be using a check box.



However, you can use more than one radio button group on a page. Just use a different name for the new radio button group and the user will be able to select from that group too.

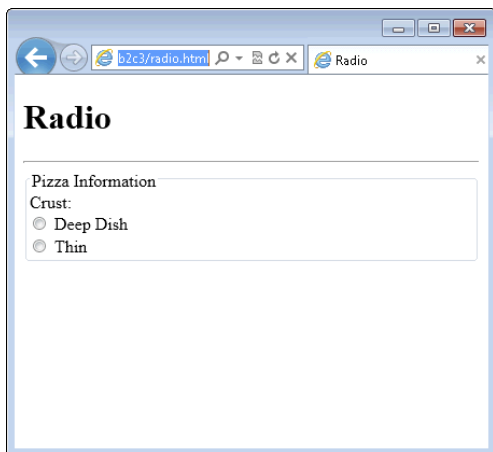


Figure 3-7:
Radio
buttons on a
web page.

Submitting and clearing the form

Thus far, you've seen some of the input types that you can use on a web page to gather information. The really big glaring piece missing from your knowledge is how to actually submit the form or send it to the server for processing. That's accomplished with another input type called submit.

```
<input type="submit" name="submit"
        value="Process Request">
```

For example, consider this example, where a Submit button is added to a form that you saw earlier in the chapter:

```
<!doctype html>
<html>
<head>
<title>A Basic Form</title>
</head>
<body>
<h1>A Basic Form</h1>
<hr>
<form action="#">
<fieldset>
  <legend>Form Information</legend>
  <div>
    <label for="username">Name:</label>
    <input type="text" id="username" name="username">
  </div>
  <div>
```

```
<label for="email">E-mail Address:</label>
<input type="text" id="username" name="email">
</div>
<div>
  <input type="submit" name="submit"
        value="Send Form">
</div>
</fieldset>
</form>
</body>
</html>
```

This HTML results in a page like that in Figure 3-8.

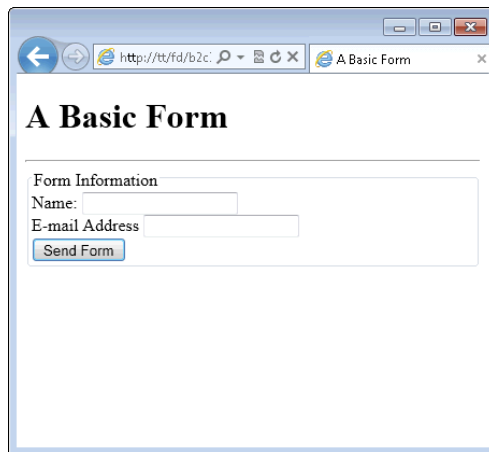


Figure 3-8:
Adding
a Submit
button.

Another button that you see on forms is a Clear or Reset button. The Reset button clears the input and resets the form, removing anything the user has placed into the form. Adding a Reset button is as simple as adding an input type of "reset":

```
<input type="reset" name="reset" value="Clear Form">
```

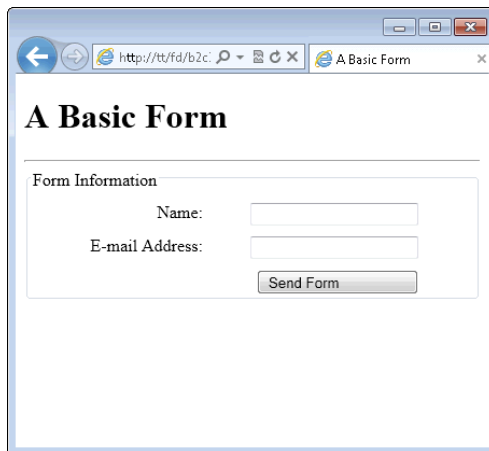
Using CSS to Align Form Fields

The form examples you've seen so far have been pretty boring, just plain HTML with no alignment or visual appeal. Forms are just standard HTML, so they can be styled using CSS. This section looks at how to do just that. The example you'll see in this section uses CSS right within the HTML file. This is done for simplicity.

When aligning form fields, the key is to use well-structured HTML. The HTML that you've seen so far in this chapter fits the bill and so aligning the form fields will be rather easy. In fact, using the HTML from the final example as a guide, merely adding this style information to the <head> section aligns the fields:

```
<style type="text/css">
.form-field {
    clear: both;
    padding: 10px;
    width: 350px;
}
.form-field label {
    float: left;
    width: 150px;
    text-align: right;
}
.form-field input {
    float: right;
    width: 150px;
    text-align: left;
}
</style>
```

The result is shown in Figure 3-9. Each of the style rules match using a CSS class and, in the case of the label and input, a child selector is further used to narrow the application of the CSS rule.



A Basic Form

Form Information

Name:

E-mail Address:

Figure 3-9:
Aligning
form fields
with CSS.

But wait! The Send Form button is now stretched to 150px wide and the text (“Send Form”) is aligned to the left side of the button. Oops, looks like that’s exactly what you asked for:

```
.form-field input {  
    float: right;  
    width: 150px;  
    text-align: left;  
}
```

You need a way to either make that button smaller or at the very least to align the text in the center of it. Steve personally likes bigger buttons. They make it easier for users to click or tap, if they’re using a mobile device. So we’re choosing to align the text in the center but leave the button the same size.

Aligning it in the center means adding something to the Submit button’s HTML in order to be able to access it within the CSS. The easiest way to do that is by adding an `id` attribute to the Submit button, like so:

```
<input id="submit" type="submit" name="submit"  
    value="Send Form">
```

Here’s the CSS to add:

```
#submit {  
    text-align: center;  
}
```

The result is shown in Figure 3-10.

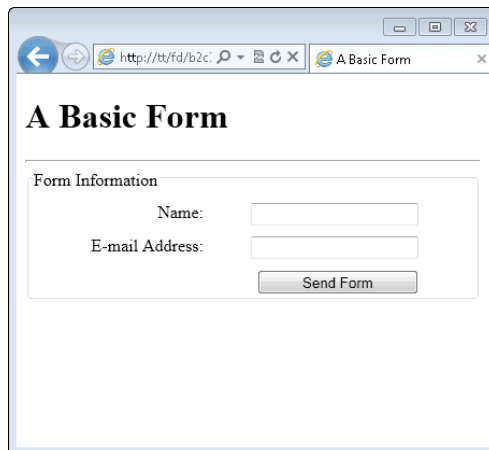


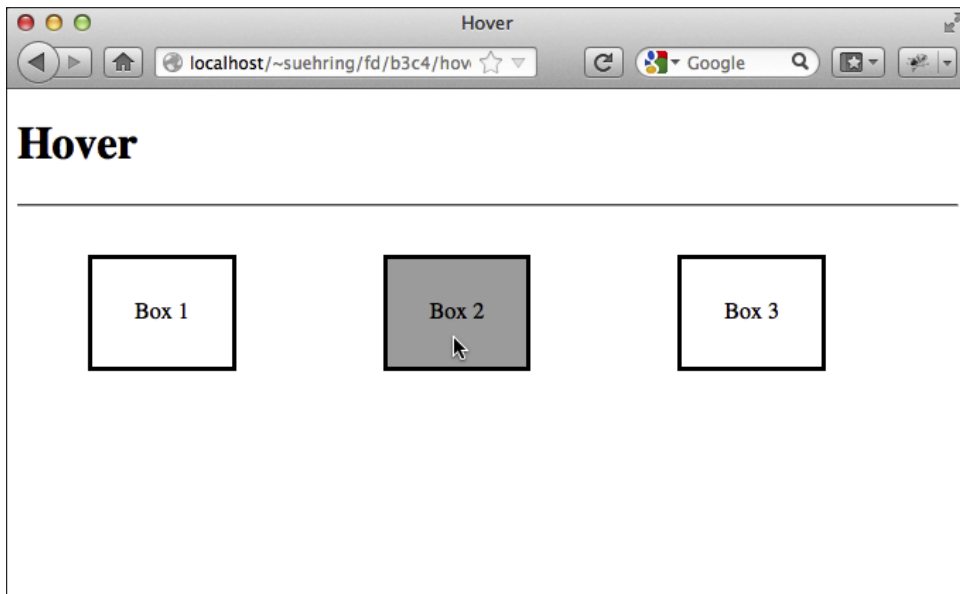
Figure 3-10:
Aligning
the text of
the Submit
button.

The full HTML and CSS are shown here:

```
<!doctype html>
<html>
<head>
<title>A Basic Form</title>
<style type="text/css">
.form-field {
    clear: both;
    padding: 10px;
    width: 350px;
}
.form-field label {
    float: left;
    width: 150px;
    text-align: right;
}
.form-field input {
    float: right;
    width: 150px;
    text-align: left;
}
#submit {
    text-align: center;
}
</style>
</head>
<body>
<h1>A Basic Form</h1>
<hr>
<form action="#">
<fieldset>
    <legend>Form Information</legend>
    <div class="form-field">
        <label for="username">Name:</label>
        <input type="text" id="username" name="username">
    </div>
    <div class="form-field">
        <label for="email">E-mail Address:</label>
        <input type="text" id="username" name="email">
    </div>
    <div class="form-field">
        <input id="submit" type="submit" name="submit"
        value="Send Form">
    </div>
</fieldset>
</form>
</body>
</html>
```


Book III

JavaScript



For more info on JavaScript, go to www.dummies.com/extras/phpmysqljavascripthtml5aio.

Contents at a Glance

Chapter 1: Understanding JavaScript Basics	187
Viewing the World of JavaScript.....	187
Examining the Ways to Add JavaScript to a Page	188
Chapter 2: Building a JavaScript Program	191
Getting Started with JavaScript Programming.....	191
Testing Things with Conditionals.....	197
Performing Actions Multiple Times with Loops	200
Using Functions to Avoid Repeating Yourself	203
Objects in Brief	208
Working with HTML Documents.....	210
Working with Web Browsers.....	214
Chapter 3: Adding jQuery	219
jQuery Introduced	219
Installing jQuery.....	220
Adding jQuery to a Page	221
Incorporating the jQuery ready() Function	223
Selecting Elements with jQuery	225
Working with HTML Using jQuery.....	227
Changing Attributes and Styles	232
Chapter 4: Reacting to Events with JavaScript and jQuery	241
Understanding Events.....	241
Working with Forms	242
Monitoring Mouse Events.....	247
Reacting to Keyboard Events.....	254
Chapter 5: Troubleshooting JavaScript Programs	261
Employing Basic JavaScript Troubleshooting Techniques.....	261
Identifying JavaScript Problems with Firebug	264

Chapter 1: Understanding JavaScript Basics

In This Chapter

- ✓ Understanding JavaScript's role in web programming
- ✓ Adding JavaScript to a page

This minibook is all about JavaScript and its place in building web applications. JavaScript is a very powerful language, and you can use it to add great features to enhance the user experience. In this chapter, we tell you a little bit about the types of interactivity that you can add to a web page with JavaScript and then show you how to add JavaScript to a page.

In the next chapter, we show you how to use JavaScript to perform some very basic programming functions, and then we follow that with a look at more practical items with JavaScript.

Viewing the World of JavaScript

JavaScript is used for web programming to enhance or add to the user experience when using a web page. This section looks at some of the aspects of JavaScript that will help you understand the language and give you a good foundation upon which you'll be able to really make your web pages stand out.

JavaScript isn't Java

Don't be confused by the name. JavaScript has absolutely nothing to do with Java — the coffee or the programming language. JavaScript's name came about because marketing folks wanted to latch onto the “cool” factor back when the Java programming language was shiny and new.

Java is a heavy language that doesn't necessarily run on everyone's computer; people have to install extra software to get it to run. Although powerful, Java is not meant for the types of web programming that you usually need to do. JavaScript, on the other hand, is included with just about every web browser and doesn't need anything else installed. You use JavaScript to make the pages come alive, with auto-populating form fields, and all kinds of bells and whistles that enhance the user experience.

One of the most common things that we hear from nontechnical folks is confusing or calling JavaScript, “Java.” Now that you know that the two are completely different, you won’t do the same! You will, however, need to resist the urge to correct people when you hear them confuse the two languages.



JavaScript is defined by the specification known as ECMA-262. Web browsers have varying degrees of support for the ECMA-262 specification, so the exact version of JavaScript that’s available in the browser varies according to the version of the browser being used.

Knowing what JavaScript can do

JavaScript is an integral part of web pages today. When you see something like Google Maps, where you can scroll left and right by simply dragging the map, that’s JavaScript behind the scenes. When you go to a site to look up flight details, and the site automatically suggests airports as you type into the field, that’s JavaScript. Countless widgets and usability enhancements that you take for granted when you use the web are actually JavaScript programs.

JavaScript programs run in the user’s web browser. This is both a blessing and a curse. On the one hand, by running on the user’s web browser it means that your server doesn’t need to run the program. On the other hand, by running in the user’s browser it means that your program runs slightly differently depending on the version of browser that the user is using on your site. In fact, the user may have JavaScript turned off completely!

While theoretically all JavaScript should run the same, in practice it doesn’t. Internet Explorer, especially older versions like 6 and 7, interpret JavaScript in entirely different ways than other browsers like Firefox and Chrome. This means that you need to create two different programs or two different ways to make the same thing work on your web pages. Luckily, there are ways around this, which you discover in this minibook.

Examining the Ways to Add JavaScript to a Page

Although JavaScript is included in everyone’s web browser, you still need to program the actions that you want to happen on your page. You might recall from Book II, Chapter 2, if you’ve read it, that you can style your page with Cascading Style Sheets (CSS) added directly to the HTML or reference a separate CSS file. Similarly, this section shows the various ways to incorporate JavaScript into a page. You can add the JavaScript directly to the HTML file, reference a separate JavaScript file, or do both — and we help you understand when each option is appropriate.

Adding the JavaScript tag

You add JavaScript to a page with the `<script>` tag, like this:

```
<script type="text/javascript">
// JavaScript goes here
</script>
```

You may see various ways to include JavaScript in a page, like `"text/ecmascript"` or without the `type` attribute at all, just an empty `<script>` tag. These methods work, sort of, and some of them are technically correct. But the one that you see most often and the one that we've had the best luck with is the one shown, with a `type` of `"text/javascript"`.

If you're wondering, the sets of double slashes you see in this example start a comment, which we tell you more about in the next chapter.

Adding JavaScript to a page's HTML

Always position the JavaScript code after the opening `<script type="text/javascript">` tag and before the closing `</script>` tag. You can include those tags in both the `<head>` section and the `<body>` section of a page.

Here's an example showing JavaScript in two different locations in a page:

```
<!doctype html>
<html>
<head>
<title>Another Basic Page</title>
<script type="text/javascript">
    // JavaScript goes here
</script>
</head>
<body>
<h1>Here's another basic page</h1>
<script type="text/javascript">
    // JavaScript can also go here
</script>
</body>
</html>
```

You could actually place as many of those separate `script` elements as you want on a page but there's usually no reason to do so.

Using external JavaScript

The example you just saw shows JavaScript within the page, in much the same way that you can add CSS inside of a page. Although that method works for small scripts and certainly comes in handy for showing examples in this book, a better way to add JavaScript is by using external JavaScript files.



Using external JavaScript files is the same concept as using external files for CSS. Doing so promotes reusability and makes troubleshooting and changes easier.

You can add external JavaScript by using the `src` attribute, like this:

```
<script type="text/javascript"
      src="externalfile.js"></script>
```

This example loads the file "externalfile.js" from the same directory on the web server. The contents of that file are expected to be JavaScript.



Notice in this example that there's nothing between the opening `<script>` and closing `</script>` tags. When using an external JavaScript file, you can't put JavaScript within that same set of tags.

You could add a reference, like the one shown, anywhere in the page, but the traditional spot for that is in the `<head>` section of the page. Also note there's nothing preventing you from using an external JavaScript file along with in-page JavaScript, so this is perfectly valid:

```
<!doctype html>
<html>
<head>
<title>Another Basic Page</title>
<script type="text/javascript" src="externalfile.js"></
  script>
<script type="text/javascript">
  // JavaScript goes here
</script>
</head>
<body>
<h1>Here's another basic page</h1>
</body>
</html>
```

This example loads an external JavaScript file and then runs some JavaScript right within the page.

Chapter 2: Building a JavaScript Program

In This Chapter

- ✓ Understanding the basic syntax of JavaScript
- ✓ Implementing JavaScript functions
- ✓ Working with JavaScript and HTML
- ✓ Using JavaScript with a web browser

The preceding chapter shows how to add the JavaScript tag to a page, and this chapter concentrates on what you can do after that. Key to understanding a programming language is learning its syntax. Just like when you learn a foreign language and you need to learn the words and grammar of the language, the syntax of a programming language is just that: the words and grammar that make up the language.

JavaScript is viewed through a web browser and programmed in a text editor, just like HTML and CSS. The examples you see throughout this chapter can be programmed just like any of the other examples you see throughout the book.

In this chapter, you'll see how to build a JavaScript program, including some of the ins and outs of programming in JavaScript.

Getting Started with JavaScript Programming

Since this might be your first exposure to programming of any kind, this section starts with some basic information to get you up to speed.

Sending an alert to the screen

You can use JavaScript to send an alert to the screen. Although this isn't used much on web pages, you do use it for troubleshooting your programs, and it's a quick way to see JavaScript in action too.

Begin by opening your text editor with a new or blank document. In the text editor, place the following HTML and JavaScript:

```
<!doctype html>
<html>
<head>
<title>Another Basic Page</title>
</head>
<body>
<h1>Here's another basic page</h1>
<script type="text/javascript">
    alert("hello");
</script>
</body>
</html>
```

Save the file as `basic.html` in your document root.

View the page by opening your web browser and navigating to `http://localhost/basic.html`. You should see a page like that in Figure 2-1.

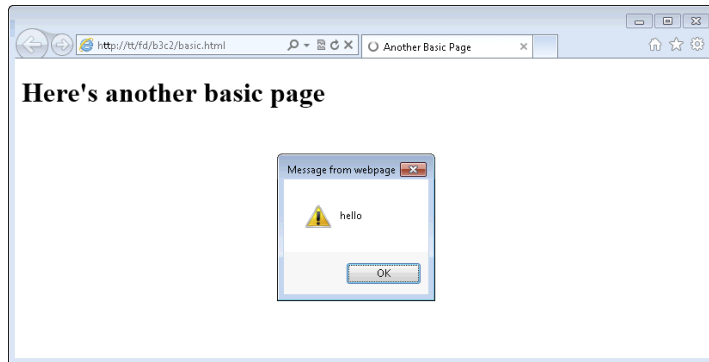


Figure 2-1:
Loading a
page with
an alert.

Click OK to dismiss the alert.

Congratulations, you're now a JavaScript programmer!

Looking at that program, contained in a single line between the opening and closing `<script>` tags, there's just the word `alert` with the word `"hello"` enclosed in quotes and parentheses. The word `alert` is actually a built-in function (more on functions later in the "Using Functions to Avoid Repeating Yourself" section).



The line of JavaScript ends with a semicolon. That's an important concept and should be a primary takeaway from this exercise: You end almost every line of JavaScript with a semicolon.

Adding comments

Just like the `alert` function is useful, so too are comments, which are like sticky notes for your code. A comment can be used so that you remember what a certain piece of code is supposed to do or can be used to skip over parts of the code that you don't want to run.

A common form of comment begins with two slashes, like this:

```
// This is a comment
```

You see that form of comment in the preceding chapter. The words that follow the two slashes won't be read by the web browser, but they can be read by people viewing your JavaScript so keep it clean!

Another type of comment begins with a front slash and an asterisk, like this `/*`, and closes with an asterisk and a front slash, like this `*/`. With that style of comment, everything in between the opening and closing comment isn't read.

```
/*  
This won't be read, it's in a comment  
*/
```

Holding data for later in variables

When you work with a programming language like JavaScript, you frequently need to hold data for later use. You'll get a value, such as input from a form that the user fills out, and then you'll need to use it later.



To hold data, you use a variable, which keeps track of the data that you tell it to store for the lifetime of your program. That's an important concept: The contents of a variable only live as long as your program. Unlike data in a database, there's no persistence for variable data.

Variables are defined in JavaScript with the `var` keyword, short for *variable*.

```
var myVariable;
```



JavaScript is case sensitive. You see in the example that the `var` keyword is lowercase and the variable `myVariable` uses mixed case. It's important to use the same case for variable names and always be aware that, for instance, `MYVARIABLE` is not the same as `myVariable` or `myvariable`. Always follow case sensitivity for JavaScript, and you'll never have a problem with it!

When you create a variable, it's common to give it some data to hold onto. You do this with the equals sign:

```
var myVariable = 4;
```

That bit of code sets the variable named `myVariable` equal to the number 4. If you don't set the variable right when you create it, like in that example, you can set it any time merely by setting it equal to the value that you want. Here's an example:

```
var myVariable;  
myVariable = 4;
```

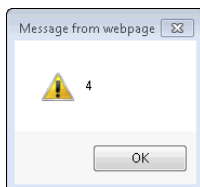
You can take a spin with variables by modifying the JavaScript you created in the preceding exercise to look like that in Listing 2-1.

Listing 2-1: Trying Out a Variable

```
<!doctype html>  
<html>  
<head>  
<title>JavaScript Chapter 2</title>  
</head>  
<body>  
<h1>Here's another basic page</h1>  
<script type="text/javascript">  
    var myVariable = 4;  
    alert(myVariable);  
</script>  
</body>  
</html>
```

If you view that code in a browser, you'll see an alert like the one shown in Figure 2-2.

Figure 2-2:
Displaying
the contents
of a
variable.



JavaScript variables can hold *strings*, which are essentially words enclosed in quotes, or numbers, like you saw in the example.



Variables need to be named in a certain way. Variables need to begin with a letter and can't begin with a number. Though certain special characters are fine, in general variables should contain only letters and numbers. Variable names should be descriptive of what they contain or what they do.

Holding multiple values in an array

Variables hold one thing and they do it well, but there are times when you want to hold multiple things. Sure, you could just create multiple variables, one for each thing. You could also create an array. An *array* is a special type of variable used to hold multiple values. Here's an example:

```
var myArray = ["Steve", "Jakob", "Rebecca", "Owen"];
```

This array contains four things, known as *elements*. You see more about arrays later, when we tell you about loops.

Creating strings to keep track of words

When you place words in quotes in JavaScript you create what's called a *string*. It's typical to place the contents of strings into variables, like this:

```
var aString = "This is a string.";
```

Strings can contain numbers, and when you put a number in quotes it will be a string. The key is the quotes, as shown here:

```
var anotherString = "This is more than 5 letters long!";
```

Strings can be enclosed in single quotes or double quotes.



Strings can be put together using the plus sign (+), as in the exercise you're about to work through.

Joining strings is called *concatenation*, and we talk a little more about that when we tell you about joining strings in Book IV, Chapter 1.

To practice creating a concatenated string, begin by opening your text editor with a new or blank document.

In the text editor, place the following HTML and JavaScript:

```
<!doctype html>
<html>
<head>
<title>JavaScript Chapter 2</title>
</head>
<body>
<h1>Here's another basic page</h1>
<script type="text/javascript">
    var myString = "Partly" + "Cloudy";
    alert(myString);
</script>
</body>
</html>
```

Save the file as `string.html` in your document root.

Open your browser and view the page by going to `http://localhost/string.html`. You should see an alert like the one in Figure 2-3.

Figure 2-3:
A concatenated string.



Look closely at Figure 2-3. Notice that there's no space between the words *Partly* and *Cloudy*. In order to have a space there it needs to be added either on the end of the word *Partly* or at the beginning of the word *Cloudy*.

Working with numbers

You already saw that JavaScript variables can hold numbers. You can also do math with JavaScript, either directly on the numbers or through variables. For example, adding two numbers:

```
var myNumber = 4 + 4;
```

Subtraction is accomplished with the minus sign (-), division with the front slash (/), and multiplication with the asterisk (*).

```
//Subtraction
var subtraction = 5 - 3;
//Division
var division = 20 / 5;
//Multiplication
var multiply = 2 * 2;
```

Testing Things with Conditionals

With a few pages of JavaScript primer done, it's time to look at a way to make decisions with JavaScript. These decisions are called *conditionals*. A good way to explain them is by explaining Steve's thought process around mowing the lawn: If it's greater than 75 degrees, then it's too hot to mow. If it's raining, then he can't mow. Otherwise, he can mow the lawn. This can be set up in JavaScript something like this:

```
if (temperature > 75) {
    alert("It's too hot to mow");
} else if (weather == "raining") {
    alert("It's raining, can't mow");
} else {
    alert("Gotta mow");
}
```

That little bit of code reveals all you need to know about conditionals in JavaScript! You test a condition and then do something based on the results of that condition.

When you set up a condition, you use parentheses to contain the test and everything that you want to happen then appears between the opening and closing braces.



Conditionals are one of the cases where you don't end each line with a semicolon.

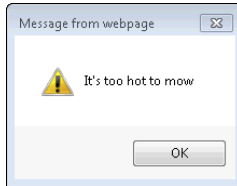
Here's an exercise that you can experiment with to work with conditionals. Begin by opening your text editor with a new or blank document. In the text editor, place the following HTML and JavaScript:

```
<!doctype html>
<html>
<head>
<title>JavaScript Chapter 2</title>
```

```
</head>
<body>
<h1>Here's another basic page</h1>
<script type="text/javascript">
  var temperature = 76;
  var weather = "raining";
  if (temperature > 75) {
    alert("It's too hot to mow");
  } else if (weather == "raining") {
    alert("It's raining, can't mow");
  } else {
    alert("Gotta mow");
  }
</script>
</body>
</html>
```

Save the file as `cond.html` in your document root, and view the page in a browser by going to `http://localhost/cond.html`. You should see an alert like the one in Figure 2-4.

Figure 2-4:
An alert
based on a
conditional
test.



Click OK to dismiss the alert.

To see how the program responds when you change a value, within the editor, change the value for temperature to 70. Here's the code; the line that changed is in bold:

```
<!doctype html>
<html>
<head>
<title>JavaScript Chapter 2</title>
</head>
<body>
<h1>Here's another basic page</h1>
<script type="text/javascript">
  var temperature = 70;
  var weather = "raining";
  if (temperature > 75) {
    alert("It's too hot to mow");
  } else if (weather == "raining") {
    alert("It's raining, can't mow");
  }
```

```

    } else {
        alert("Gotta mow");
    }
</script>
</body>
</html>

```

Save `cond.html`.

Reload the page in your browser by pressing Ctrl+R or Command+R. You should see an alert like the one in Figure 2-5.

Figure 2-5:
Getting into
the else if
condition.



Click OK to dismiss the alert.

Take a look at what happens when you change another variable. Within `cond.html`, change the weather variable to `"sunny"`. The code should look like this; again the change has been bolded.

```

<!doctype html>
<html>
<head>
<title>JavaScript Chapter 2</title>
</head>
<body>
<h1>Here's another basic page</h1>
<script type="text/javascript">
    var temperature = 70;
    var weather = "sunny";
    if (temperature > 75) {
        alert("It's too hot to mow");
    } else if (weather == "raining") {
        alert("It's raining, can't mow");
    } else {
        alert("Gotta mow");
    }
</script>
</body>
</html>

```

Save `cond.html` and reload the page in your browser. You should see an alert like the one in Figure 2-6.

Figure 2-6:
Getting into
the else
condition.



Click OK to dismiss the alert.

If a test fails, a conditional can be set up to run another test. In the case of the example, a second test is set up to look at the weather to see if it's raining. Notice the use of the double equals signs in the `else if` condition.

Finally, if all tests fail then a block of code can be set up so that it runs when all else fails. This is noted by the `else` keyword in the code sample.



It's important to note that once a condition is true, in the example once the temperature is greater than 75, the code in that block will execute but none of the other conditions will be evaluated. This means that none of the other code in any of the other blocks will ever run.

Performing Actions Multiple Times with Loops

Sometimes you want to repeat the same code over and over again. This is called *looping*, and JavaScript includes a couple ways to do it, including `for` and `while`.

For what it's worth

If you want to do something multiple times in JavaScript, a common way to do it is with a `for` loop. A `for` loop has pretty specific syntax, as you see here:

```
for (var i = 0; i < 10; i++) {  
    // Do something here  
}
```

That structure includes three specific things within the parentheses.

- ◆ **Variable:** First, a variable is set up, in this case simply called `i`. That variable is set to the number 0.
- ◆ **Condition:** Next is the condition to be tested. In this case, the loop tests whether the variable `i` is less than 10. If `i` is less than 10, the code inside the braces runs.

- ◆ **Postfix operator:** The final piece of the `for` loop construct increments the `i` variable using something called a *postfix operator* (`i++`), which basically increments the value by 1.

In plain language, this loop creates a variable and sets it to 0, then it tests to see whether the variable is still less than 10. If it is, then the code within the block is executed. For now, that code is merely a comment so nothing happens. If not, then the variable is incremented by 1 and the whole thing starts all over again.



The first two parts of a `for` loop use semicolons; the last part doesn't.

The first time through, the variable `i` is 0, which is (obviously) less than 10 — and the code inside the block executes so that `i` is incremented by 1. The next time through, the value of `i` is 1, which is still less than 10, so the code in the block is executed again. This keeps going until the value of `i` is 10.

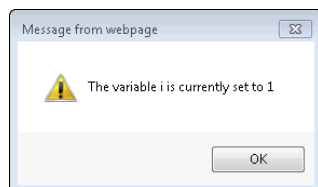
Try it out with an exercise. Begin by opening your text editor with a new or blank document. In the text editor, place the following HTML and JavaScript:

```
<!doctype html>
<html>
<head>
<title>JavaScript Chapter 2</title>
</head>
<body>
<h1>Here's another basic page</h1>
<script type="text/javascript">
    for (i = 0; i < 10; i++) {
        alert("The variable i is currently set to " + i);
    }
</script>
</body>
</html>
```

Save the file as `for.html` in your document root.

View the file in a browser by going to `http://localhost/for.html`. You'll see a series of alerts, one of which is shown in Figure 2-7.

Figure 2-7:
The results
of a `for` loop.



Now take a look at how to determine the length of an array. Earlier in the chapter you saw an array like this one:

```
var myArray = ["Steve", "Jakob", "Rebecca", "Owen"];
```

A common use of a `for` loop is to spin through an array and do something with each value. The conditional in the example `for` loop you saw earlier set the value at 10. But how do you know how many values are in an array? Yes, you easily could count the number of variables in the array shown, but sometimes you have no idea how many elements are in an array.

You can ask the array itself to tell you how long it is by asking it. You ask through the `length` property, like this:

```
myArray.length;
```

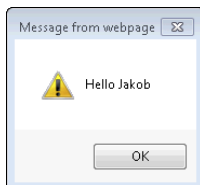
Listing 2-2 shows an example that loops through the array shown and displays each element.

Listing 2-2: Using a for Loop on an Array

```
<!doctype html>
<html>
<head>
<title>JavaScript Chapter 2</title>
</head>
<body>
<h1>Here's another basic page</h1>
<script type="text/javascript">
    var myArray = ["Steve", "Jakob", "Rebecca", "Owen"];
    for (i = 0; i < myArray.length; i++) {
        alert("Hello " + myArray[i]);
    }
</script>
</body>
</html>
```

This code uses a standard `for` loop but instead of setting the length to a certain number, it uses the `length` property to find out how long `myArray` really is. Each time through the loop, an alert is shown, like the one in Figure 2-8.

Figure 2-8:
Displaying
an alert
from an
array.



The `i` variable is used right within the loop, to access each element of the `myArray` variable. You see, an array is an ordered list of things, with the order being provided by numbers that you don't normally see. These hidden numbers are called *indexes*. The index of the first element in an array is 0 (not 1 as you might expect).

In this example, since `i` is 0 the first time through the loop it can access the first element. The second time through the loop, as shown in Figure 2-8, `i` is equal to 1 and so the second element is shown.



The syntax that you see there, `myArray[i]`, is a really common syntax that you see in `for` loops.

While you're here

Another type of loop is called a `while` loop, and it looks like this:

```
while (i < 10) {
    // Do something interesting
    // Don't forget to increment the counter!
}
```

A `while` loop is similar to a `for` loop insofar as the code within the braces executes as long as the condition is true. Unlike a `for` loop, though, you need to explicitly do something inside of the loop in order to break out of the loop. If you forget, you'll be stuck in an endless loop!

Using Functions to Avoid Repeating Yourself

A good programming practice is to reuse code whenever possible. Not only does this cut down on the number of possible errors in your code, but it also makes for less work, which is always good when it comes to coding. This section looks at a primary way to implement code reuse: functions.

JavaScript includes a number of built-in functions. You've been using one throughout the chapter: `alert()`. The `alert()` function creates a dialog in the browser.

Creating functions

Functions are created with the `function` keyword followed by the name of the function, parentheses, and opening and closing braces, like this:

```
function myFunction() {  
    // Function code goes here  
}
```

What you do inside of the function is up to you. Anything that you can do outside of the function you can do inside of it. If you find that your page needs to update a bunch of HTML, you could use a function so that you don't need to keep repeating that same code over and over again.

Adding function arguments

The power of functions comes with their capability to accept input, called *arguments*, and then do something with that input.

For example, here's a simple function to add two numbers:

```
function addNumbers(num1,num2) {  
    alert(num1+num2);  
}
```

This function accepts two arguments called `num1` and `num2`. Those arguments are then used within the `alert()` function. You've seen the `alert()` function throughout the chapter and now you understand a bit more about what's going on! The `alert()` function accepts one argument, the text to display in the alert dialog. In this case, because you're adding two numbers the alert displays the resulting number. You work through an exercise for this in the following section.

Calling a function

Just creating the function isn't enough; you need to call it too. *Calling a function* means that you execute it, just like when you called the `alert()` function earlier in this chapter. Until you call a function it doesn't really do anything, much like the `alert()` function doesn't do anything until you invoke it.

Calling one of your own functions looks just like the call to the `alert()` function. Here's that exercise we promised. Begin by opening your text editor with a new or blank document. In the text editor, place the following HTML and JavaScript:

```
<!doctype html>
<html>
<head>
<title>JavaScript Chapter 2</title>
</head>
<body>
<h1>Here's another basic page</h1>
<script type="text/javascript">

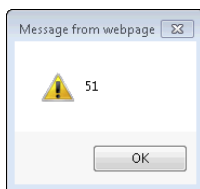
    // Define the function
    function addNumbers(num1,num2) {
        alert(num1+num2);
    }

    // Call the function
    addNumbers(49,2);

</script>
</body>
</html>
```

Save the file as `func.html` in your document root. Open your browser and point to `http://localhost/func.html`. You should see an alert like the one shown in Figure 2-9.

Figure 2-9:
Executing a
function.



Improving the `addNumbers` function

The function that you've created, `addNumbers()`, accepts two arguments and adds them. But what if you send in something that isn't a number? The function has no way to test that and so it happily tries to add them. To experiment with this, change the 2 in the call to `addNumbers` to "two", like this:

```
addNumbers(49, "two");
```

When you reload the page, you'll see an alert like the one in Figure 2-10.

Figure 2-10:
Trying
to add
something
that isn't a
number.



JavaScript includes a function to test whether something is a number. The function is called `isNaN()`, which stands for *is not a number*. This can be added anywhere that you need to test to make sure something is a number before working with it, like in the case of the `addNumbers()` function. You use the `isNaN()` function within an `if` conditional and then react accordingly if it isn't a number. Here's an updated `addNumbers()` function:

```
function addNumbers(num1,num2) {  
    if (isNaN(num1)) {  
        alert(num1 + " is not a number");  
    } else if (isNaN(num2)) {  
        alert(num2 + " is not a number");  
    } else {  
        alert(num1+num2);  
    }  
}
```

If you call it with one of the two arguments as something other than a number, you'll receive an alert stating that. For example, if you changed the number 2 to "two", you'd receive the alert shown in Figure 2-11.

Figure 2-11:
An alert
generated
using
`isNaN()`.



JavaScript is case sensitive, so you need to make sure you use `isNaN()` with the correct case.

You see variations on functions throughout the remainder of the book that help build on this introduction.

Returning results from functions

The function that you created in this section sends an alert. But there are times when you want to have the function send something back to you — to do something and then return the results.

The `return` keyword is used to return results from a function. In the `addNumbers()` function example shown, instead of using an `alert()` right within the function you could return the result. Here's an example:

```
function addNumbers(num1,num2) {  
    var result = num1+num2;  
    return result;  
}
```

You can call the function just like before, but you now need to capture the result, typically into another variable, like this:

```
var myResult = addNumbers(49,2);
```

Listing 2-3 shows the full HTML for this example:

Listing 2-3: Returning a Value from a Function

```
<!doctype html>  
<html>  
<head>  
<title>JavaScript Chapter 2</title>  
</head>  
<body>  
<h1>Here's another basic page</h1>  
<script type="text/javascript">  
  
    // Define the function  
    function addNumbers(num1,num2) {  
        var result = num1+num2;  
        return result;  
    }  
  
    // Call the function  
    var myResult = addNumbers(49,2);  
  
</script>  
</body>  
</html>
```




Only one return is allowed from a function and nothing after the return statement executes. Once you call a return, the function ends.

Objects in Brief

You've seen arrays and how they can be used to hold multiple values. Arrays hold that data using an unseen numbered index. On the other hand, objects can hold multiple values but those values are accessed through a named index, called a *property*. Objects can actually do a lot more than this, such as hold functions (called *methods*) but for this example, consider this narrow focus, using an object to hold multiple values.

Creating objects

Here's an example object for a ball:

```
var ball = {  
  "color": "white",  
  "type": "baseball"  
};
```

Whereas arrays are created with square brackets, objects are created with curly braces, as shown in the example. When you define an object, you can define one or more properties (akin to the elements in an array). In this instance, you created two properties, one called `color` and one called `type`. The values are then set to `white` and `baseball`, respectively.

You can access objects using a single dot, like so:



```
ball.color
```

The single dot is known as *dot notation* when used in this way.

Listing 2-4 shows HTML to create a ball object and display its color property.

Listing 2-4: Creating an Object and Displaying a Property

```
<!doctype html>  
<html>  
<head>  
<title>JavaScript Chapter 2</title>  
</head>  
<body>  
<h1>Here's another basic page</h1>  
<script type="text/javascript">  
  var ball = {  
    "color": "white",  
    "type": "baseball"
```

```

    };
    alert(ball.color);
</script>
</body>
</html>

```

When viewed in a browser, you get an alert like the one shown in Figure 2-12.

Figure 2-12:
Displaying
an object
property.



Adding properties to objects

Sometimes you want to add properties onto objects after they've been created. This too can be done using dot notation, like so:

```
ball.weight = 15;
```

Here's an exercise to create an object, add to it, and then loop through it using a new kind of loop constructor. Begin by opening your text editor with a new or blank document. In the text editor, place the following HTML and JavaScript:

```

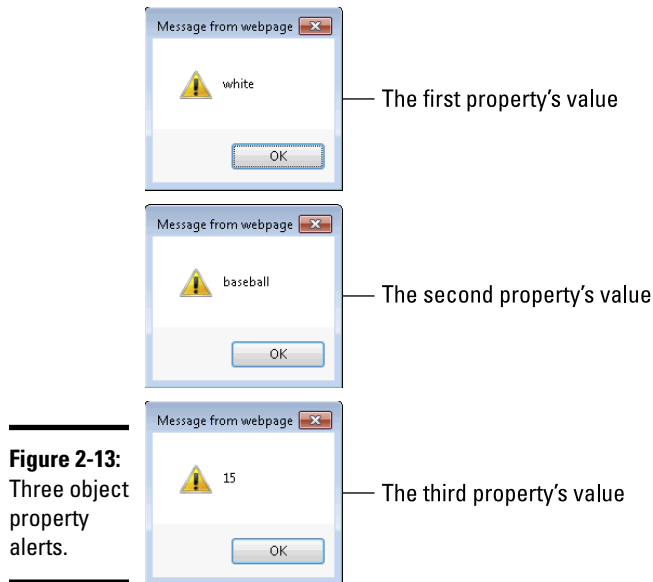
<!doctype html>
<html>
<head>
<title>JavaScript Chapter 2</title>
</head>
<body>
<h1>Here's another basic page</h1>
<script type="text/javascript">
    var ball = {
        "color": "white",
        "type": "baseball"
    };

    ball.weight = 15;

    for (var prop in ball) {
        alert(ball[prop]);
    }
</script>
</body>
</html>

```

Save this as `obj.html` in your document root. View the page in a browser by going to `http://localhost/obj.html`. You should see three alerts, like the ones in Figure 2-13.



As you can see from the alerts, the properties that were created right along with the object are shown, as is the weight property that was added later using the dot notation.

Later in this chapter, you see much more about objects, so this bit of background will be helpful.

Working with HTML Documents

All this JavaScript programming gets put to practical use when you start adding it to web pages. JavaScript integrates into HTML and has access to everything in a web page. This means that you can add HTML to a page, take it away, or change it, all on the fly, in real time.

In order to work together, JavaScript and HTML need a common language so that JavaScript can know what to do on a page. JavaScript and HTML work together through something called the Document Object Model (DOM). The DOM gives JavaScript access to a web page so that it can manipulate the page.



The connection between JavaScript and HTML is through the `document` object. You see the `document` object used in this section along with other functions to access pieces of a page using JavaScript. The `document` object is actually a child of the `window` object and there are other children that are interesting too, as you see later in this chapter.

Accessing HTML with JavaScript

You access parts of a web page, the `document` object, using JavaScript functions. You can look at a piece of the page to see what text it has in it or change the text in the page. You can also add to the page with JavaScript and much more.

Using *getElementById* to access a specific element

The most specific way that you can access an element on a page is to use its ID. Recall that ID attributes can be placed on any element, and they're (supposed to be) unique throughout the page. In this way, each element that is already part of the DOM can be accessed directly rather than by traversing the document tree.

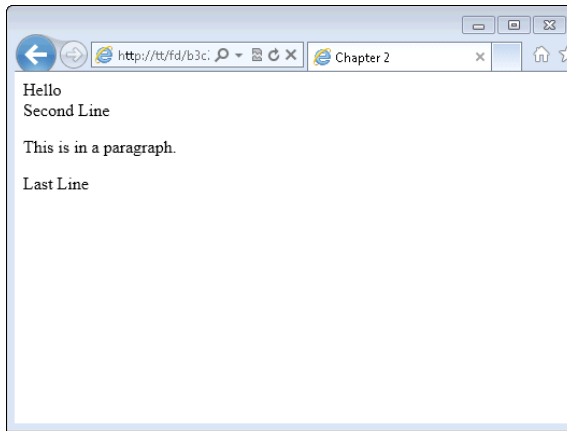
Consider the HTML in Listing 2-5.

Listing 2-5: Basic HTML for Demonstrating the DOM

```
<!doctype html>
<html>
<head>
<title>Chapter 2</title>
</head>
<body>
<div id="myDiv">Hello</div>
<div class="divClass">Second Line</div>
<div class="divClass">
  <p class="pClass">This is in a paragraph.</p>
</div>
<div class="divClass">Last Line</div>
</body>
</html>
```

That HTML, when viewed in a browser, creates a page like the one shown in Figure 2-14.

Figure 2-14:
Creating
a basic
page to
demonstrate
the DOM.



Looking at the HTML again, you can see an ID attribute on the first `<div>` on the page. You can use the `getElementById` function to access that element. You're saying, "Great, I can access the element but what can I do with it?" Glad you asked.

When you access an element, you view its current HTML or make changes such as CSS styling or the actual contents of the element itself. Try it out in an exercise.

1. Open your text editor with a new or blank document.
2. In the text editor, place the following HTML and JavaScript:

```
<!doctype html>
<html>
<head>
<title>Chapter 2</title>
</head>
<body>
<div id="myDiv">Hello</div>
<div class="divClass">Second Line</div>
<div class="divClass">
  <p class="pClass">This is in a paragraph.</p>
</div>
<div class="divClass">Last Line</div>

<script type="text/javascript">
  var theDiv = document.getElementById("myDiv");
  alert("The content is " + theDiv.innerHTML);
</script>

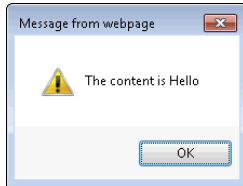
</body>
</html>
```

3. Save the file as `getbyid.html` in your document root.
4. Open your web browser and view the page at `http://localhost/getbyid.html`.

You should see an alert like the one shown in Figure 2-15.

Figure 2-15:

An alert produced by the `getElementById` function.



5. Click OK to dismiss the alert and close your browser.
6. Back within `getbyid.html` in your editor, remove the JavaScript line that begins with `alert (`. Replace that line with these two:

```
theDiv.style.border = "3px solid black";
theDiv.innerHTML = "This is now changed text.";
```

The entire script block should now look like this:

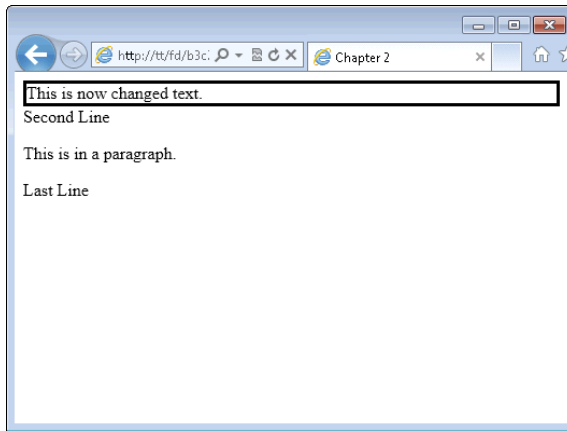
```
<script type="text/javascript">
  var theDiv = document.getElementById("myDiv");
  theDiv.style.border = "3px solid black";
  theDiv.innerHTML = "This is now changed text.";
</script>
```

When viewed in a browser, the page now looks like that in Figure 2-16. Notice specifically that the text of the top line has been changed and now has a border.

In this exercise, you created HTML and JavaScript. In the JavaScript, you accessed an HTML element using the `getElementById` function. From there you displayed it using an `alert()`.

The second part of the exercise saw you change the element's contents using `innerHTML` and also change the CSS style of the element using the `style.border` property.

Figure 2-16:
The page
after
changing
it with
`getElement
ById`.



You’ve now seen how to use `getElementById` as part of the DOM in JavaScript, so check that one off of your bucket list. It’s good to have the understanding that `getElementById` is there in case you need to work with someone else’s JavaScript. However, there’s a better way to work with web pages through JavaScript and it’s called jQuery. You learn about jQuery in the next chapter of this minibook. For now, savor your victory over the DOM.

Working with Web Browsers

This JavaScript primer wraps up with a quick look at JavaScript’s view of the web browser. As you just saw, when a page is loaded, the `document` object gives a view of the page to JavaScript. Likewise, a couple other objects give JavaScript a view of the web browser itself.

Using these objects, which are children of the window object, you can do things like detect what type of browser the visitor is using and also redirect the user to a different web page entirely.

Detecting the browser

The `navigator` object is used to detect things about the visitor’s browser, like what version it is. This information can be used to present a specific page or layout to the user.

Limitations of browser detection

When you use a method like the one shown here, you need to be aware that it isn't always accurate. Detecting the browser in this way relies solely on what the browser claims that it is and this information can be trivially faked

by the user. Therefore, when you use the `navigator` object (or any other "User Agent Sniffer") method, you should be aware that there are limitations to its accuracy and it is definitely not 100% foolproof.

Listing 2-6 shows HTML and JavaScript to display the `userAgent` property of the `navigator` object.

Listing 2-6: Displaying the User Agent

```
<!doctype html>
<html>
<head>
<title>Chapter 2</title>
</head>
<body>
<div id="output"></div>
<script type="text/javascript">
    var outputDiv = document.getElementById("output");
    outputDiv.style.border = "3px solid black";
    outputDiv.style.padding = "3px";

    var userAgent = navigator.userAgent;
    outputDiv.innerHTML = "You are using " + userAgent;
</script>
</body>
</html>
```

When viewed in a browser, the output looks like that in Figure 2-17. Note that if you run this code, your browser version will likely be different than this.

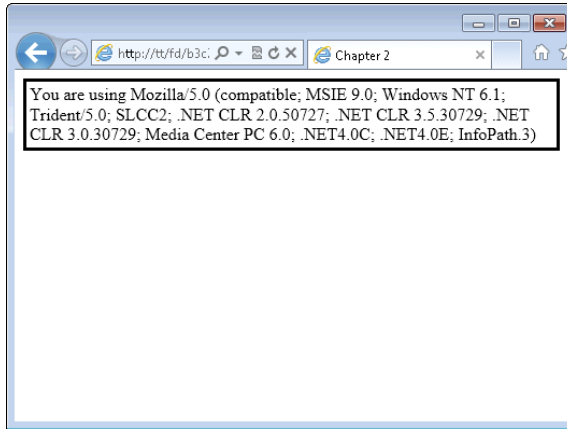


Figure 2-17:
Viewing the
userAgent
property.

Redirecting to another page

You’ve probably encountered one somewhere along the way, a page that says “Click here if you’re not automatically redirected” and then automatically redirects you anyway. Did you ever wonder why they bother with the “Click here” part? That’s done in case your browser doesn’t have JavaScript enabled. This section shows the code for such a page.

The `location` object provides the capability to redirect to another page, and it’s one of the simplest JavaScript pages you’ll ever write. Listing 2-7 shows the HTML and JavaScript.

Listing 2-7: A Redirect Page

```
<!doctype html>
<html>
<head>
<title>Chapter 2</title>
</head>
<body>
<div>
  <a href="http://www.braingia.org">          Click here if
    you're not automatically redirected...
  </a>
</div>
<script type="text/javascript">
  window.location.replace("http://www.braingia.org");
</script>
</body>
</html>
```

A page viewed in a browser looks like the one in Figure 2-18, but only for a short time. In fact, you may not even see it before being redirected!

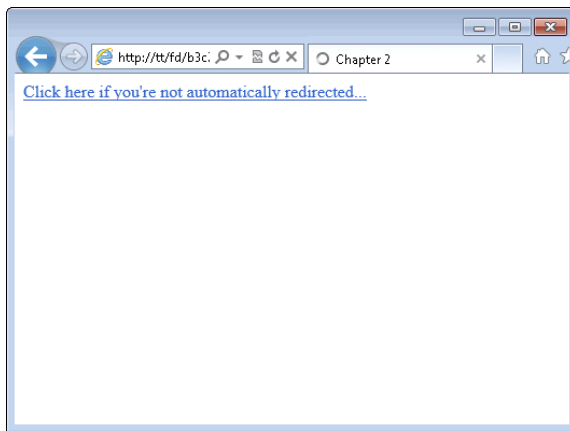


Figure 2-18:
The redirect
page just
prior to
being
redirected.

The HTML for this page simply sets up a basic `<a>` element with a link, no JavaScript necessary. Then the JavaScript uses the `location.replace` object to send the user to a different page. Almost nothing to it!

There's more to both the `navigator` and `location` objects in JavaScript. For more information on the `navigator` object, go to this page on Mozilla Developer Network:

```
https://developer.mozilla.org/en-US/docs/DOM/window.  
navigator
```

For more information on the `location` object, see this page:

```
https://developer.mozilla.org/en-US/docs/DOM/window.  
location
```