Web Application Development

Editor: Wu Cheng Wen

Contents

Figure content

Example Content

Table Content

1 Getting Start

1.1 Creating a Basic Page

Take note of just one more thing before you start writing web pages. You should know what HTML is, what it can do, and most important, what it can't do.

HTML stands for Hypertext Markup Language. HTML was originally based on the Standard Generalized Markup Language (SGML), a much larger, more complicated document-processing system. To write HTML pages, you won't need to know much about SGML.

HTML is a language for describing the **structure** of a document, not its actual presentation (We will know that in CSS). The idea here is that most documents have common elements—for example, titles, paragraphs, and lists. Before you start writing, therefore, you can identify and define the set of elements in that document and name them appropriately (see Figure 1-1 Document elements).

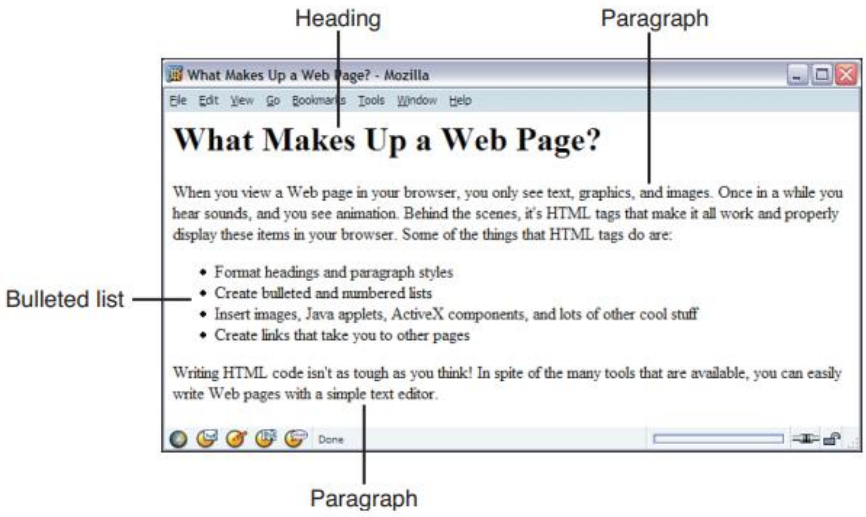


Figure 1-1 Document elements

HTML defines a set of common elements for web pages: headings, paragraphs, lists, and tables. It also defines character formats such as boldface and code examples. These elements and formats are indicated inside HTML documents using **tags**. Each tag has a specific name and is set off from the content of the document using a notation that I discuss a bit later.

When you're working with a word processor or page layout program, styles are not just named elements of a page; they also include formatting information such as the font size and style, indentation, underlining, and so on. So, when you write some text that's supposed to be a heading, you can apply the Heading style to it, and the program automatically formats that paragraph for you in the correct style.

HTML doesn't go this far. For the most part, the HTML specification doesn't say anything about how a page looks when it's viewed. HTML tags just indicate that an element is a heading or a list; they say nothing about how that heading or list is to be formatted. The only thing you have to worry about is marking which section is supposed to be a heading, not how that heading should look.

HTML elements can be modified by **attributes**. Attributes are placed within the opening tag in an element. Many elements support specialized attributes, and there are also a few global elements that can be used with any tag. For example, the ID attribute is used to specify an identifier that uniquely identifies that element on the page. These identifiers are used with **JavaScript** and **Cascading Style Sheets**(CSS) , as you'll learn in later lessons. Here's what a tag with an attribute looks like:

```
<h1 id="theTopHeading">Everything You Need to Know About HTML</h1>
```

As you can see, the attribute is placed within the opening tag, to the right of the tag name. You can also include multiple attributes in a single tag, as follows:

```
<h1 id="theTopHeading" class="first">Everything You Need to Know
About HTML</h1>
```

Now, let's go.

1.1.1 Open a Text Editor

You can use any text editor you want, as long as it lets you save files as plain text. If you're using Windows, Notepad is fine for now. If you're using Mac, you'll really need to download a text editor, Komodo Edit [1] or TextWrangler [2].

Don't use a word processor like Microsoft Word or Mac TextEdit. These are powerful tools, but they don't save things in the right format. A word processing program won't do it correctly. Even the Save as HTML feature doesn't work right. You really need a very simple text editor, and in this handbook, I show you an editor that make your life easier. You should not use Word or TextEdit.

[1] Komodo Edit, www.activestate.com/komodo-edit.

[2] TextWrangler, www.barebones.com/products/textwrangler/.

1.1.2 Type the Following Code

Really. Type it in your text editor so you get some experience writing the actual code. I explain very soon what all this means, but type it now to get a feel for it:

```
<!DOCTYPE HTML>
<head>
```

```
<meta charset="UTF-8">
<!-- myFirst.html -->
<title>My very first web page!</title>
</head>
<body>
<h1>This is my first web page!</h1>
<p>
This is the first web page I've ever made,
and I'm extremely proud of it.
It is so cool!
</p>
</body>
</html>
```

1.1.3 Save the File as myFirst.html

It's important that your filename has no spaces and ends with the *.html* extension. Spaces cause problems on the Internet, and the *.html* extension is how most computers know that this file is an HTML file (which is another name for a web page). It doesn't matter where you save the file, as long as you can find it in the next step.

1.1.4 Open Your Web Browser.

The web browser is the program used to look at pages. You can double click the file. Your simple text file is transformed! If all went well, it looks like Figure 1-2: The first web page.

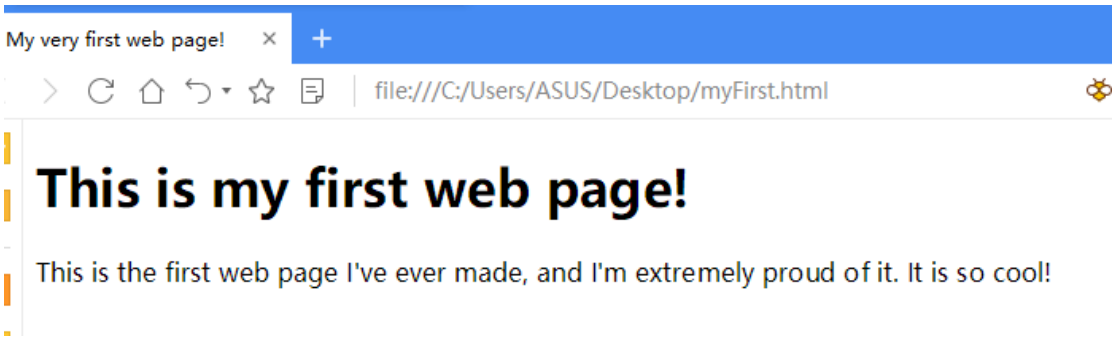


Figure 1-2: The first web page

1.2 Introduction of HBuilder

In this section, we will learn the HTML file editor. This software is very fit to edit HTML file, and many tips are given when we edit file.

Step 1: Skip the login page.

The software is a beta version, so we must skip the login page. When the editor is started as Figure 1-3 , you can click the middle button to skip the login page.

Figure 1-3 HBuilder login page

Step 2, Skip welcome page

Click close button to skip the welcome page. It is shown as Figure 1-4.

Figure 1-4 HBuilder welcome page

Step 3 Create a new project

You can click the menu in turn, as Figure 1-5.the new project dialog is called.

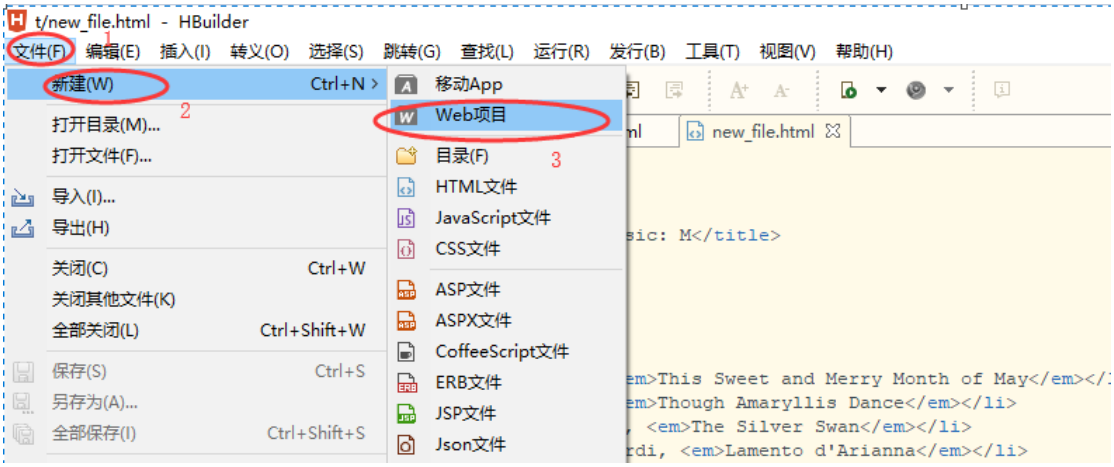


Figure 1-5 Create a new web project

Step 4: Confirm the project information

After you click "web 项目" menu, the new project dialog is shown as Figure 1-6.

Now, you can decide your project name and the directory. And then, click "完成(F)" button， a new project is created in the directory.

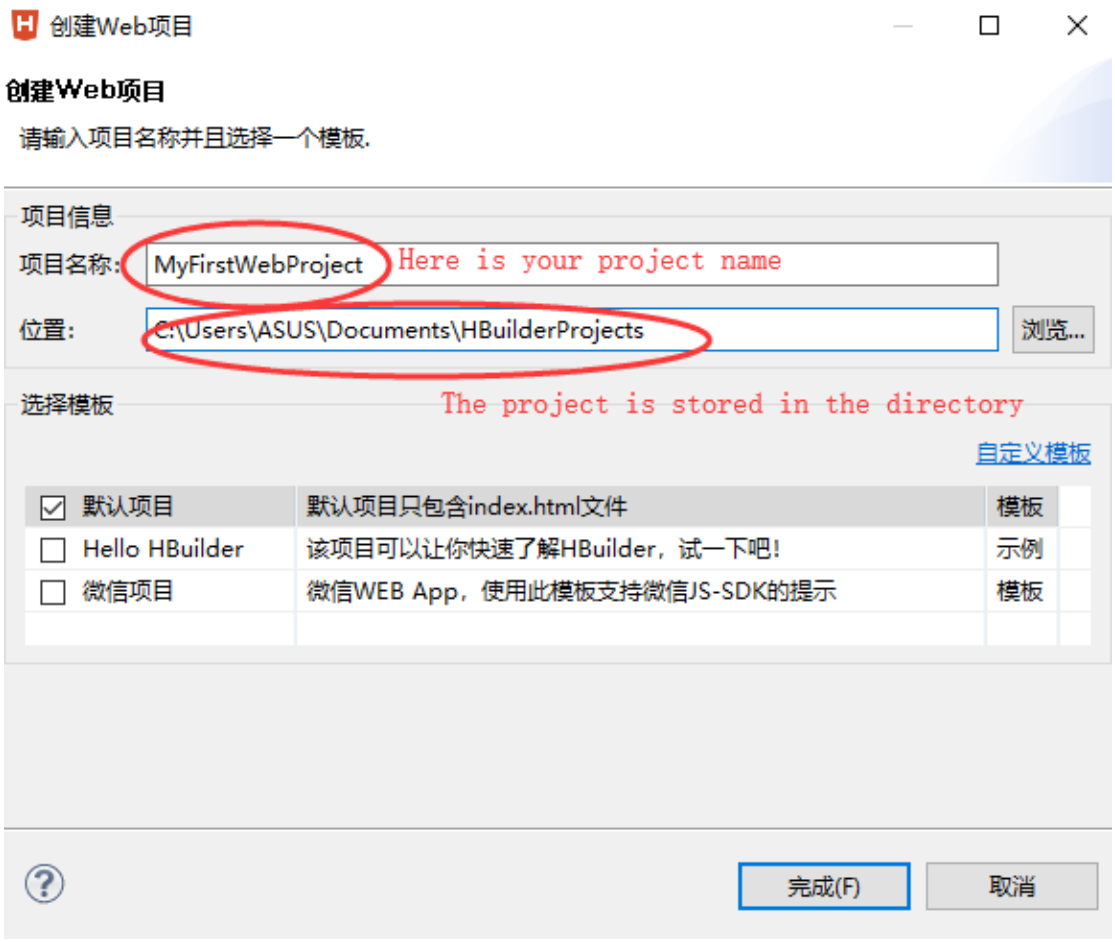


Figure 1-6 the new project information

2 Creating Web Pages

With HTML you can create your own Web site. This chapter teaches you something about HTML. HTML is easy to learn. You will enjoy it.

2.1 HTML basic

HTML defines three tags that are used to define the page's overall structure and provide some simple header information. These three tags—<html>, <head>, and <body>—make up the basic skeleton of every web page.

- All HTML documents must start with a type declaration: <!DOCTYPE html>.
- The HTML document itself begins with <html> and ends with </html>.
- The visible part of the HTML document is between <body> and </body>.
- The HTML <head> element is placed between the <html> tag and the <body> tag.

Example 2-1 There basic tags

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
  </body>
<html>
```

2.1.1 head

The <head> element is a container for all the head elements, rather than information that will be displayed on the page. Generally, only a few tags are used in the <head> portion of the page. This element can include a title for the document, styles, meta information, and more.

The following elements can go inside the <head> element:

- title
- style
- meta

Title

The <title> tag defines a title for your HTML document, the tag is required in all HTML documents and it defines the title of the document.

Usage of the <title> element:

- ✧ defines a title in the browser toolbar
- ✧ provides a title for the page when it is added to favorites
- ✧ displays a title for the page in search-engine results

Example 2-2 Title tag

```
<!DOCTYPE html>
<html>
<head>
<title>HTML Reference</title>
</head>
<body>
The content of the document......
</body>
</html>
```

The result is shown as Figure 2-1 Title tag.

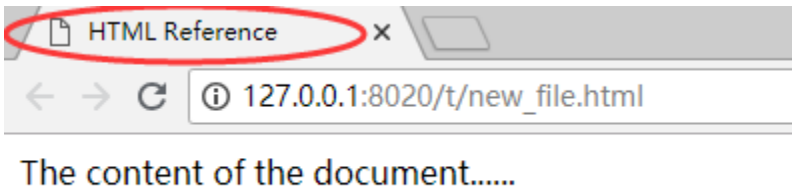


Figure 2-1 Title tag

style

The <style> tag is used to define style information for an HTML document.

Inside the <style> element you specify how HTML elements should render in a browser.

Each HTML document can contain multiple <style> tags.

Example 2-3 Style tag

```
<html>
```

```
<head>
<style>
h1 {color:red;}
p {color:blue;}
</style>
</head>
<body>
<h1>A heading</h1>
<p>A paragraph.</p>
</body>
</html>
```

The result is shown as Figure 2-2 style tag

Figure 2-2 style tag

We will explain that in chapter 3.

Meta

Describe metadata within an HTML document. Metadata is information about data. The <meta> tag provides metadata about the HTML document. **Metadata will not be displayed on the page**, but will be machine parsable. Meta elements are typically used to specify page description, keywords, and author of the document, last modified, and other metadata. The metadata can be used by browsers (how to display content or reload page), search engines (keywords), or other web services.

The three main attribute of the tag are http-equiv, name, and content.

- ✓ http-equiv: Sets an HTTP response header for the value in the content attribute.
- ✓ name: Describe information of HTML document, such as, "**description**"(description information of the document), "**keywords**"(key words of the document, and it can be searched by key words), Do not use the http-equiv and name attributes together.
- ✓ Content: content of the name or http-equiv.

Example 2-4 Meta information

```
<head>
<meta charset="UTF-8">
<meta http-equiv=”refresh” content=”3”>
<meta name="description" content="Free Web tutorials">
<meta name="keywords" content="HTML, CSS, XML, JavaScript">
<meta name="author" content="W3C school">
</head>
```

```
<meta http-equiv=”refresh” content=”3”>
```

The web page will be auto-refreshed every 3 seconds.

```
<meta name="description" content="Free Web tutorials">
```

The description information of the web page is descripted by the content.

```
<meta name="keywords" content="HTML, CSS, XML, JavaScript">
```

The web page will be indexed by these key words and searched by engine.

```
<meta name="author" content="W3C school">
```

This meta data shows the author of the web page.

Metadata will not be displayed on the browser!!!!

2.1.2 body

The content of your HTML page (represented in the past example as ...your page...) resides within the <body> tag. This includes all the text and other content (links, pictures, and so on). In combination with the <html> and <head> tags.

The body tag has many attributes, and we express as follow.

- bgcolor: Specifies the background color for the document or element. Specify either a color name or RGB color.
- background : Specifies the URL for the background image for the document or element.
- text: Specifies the color for text of the document. Specify either a color name or RGB color.
- class: CSS class assigned to the element.
- link: Specifies the color for links in the document. Specify either a color name or RGB color.
- alink: Specifies the color for active links in the document. Specify either a color name or RGB color.

- vlink: Specifies the color for visited links in the document. Specify either a color name or RGB color.

2.1.3 Heading

Headings are used to add titles to sections of a page. HTML defines six levels of headings. Heading tags look like the following:

<h1>Installing Your Safe Lock</h1>

The numbers indicate heading levels (h1 through h6). The headings, when they're displayed, aren't numbered. They're displayed in larger and bolder text so that they stand out from regular text.

Think of the headings as items in an outline. If the text you're writing is structured, use the headings to express that structure, as shown in the following code:

Example 2-5 Headings

```
<h1>Movies</h1>
<h2>Action/Adventure</h2>
<h3>Caper</h3>
<h3>Sports</h3>
<h3>Thriller</h3>
<h3>War</h3>
<h2>Comedy</h2>
<h3>Romantic Comedy</h3>
<h3>Slapstick</h3>
<h2>Drama</h2>
<h3>Buddy Movies</h3>
<h3>Mystery</h3>
<h3>Romance</h3>
<h2>Horror</h2>
```

The result is shown as Figure 2-3 HTML heading elements.

Figure 2-3 HTML heading elements

2.1.4 Comments

You can put comments into HTML pages to describe the page itself or to provide some kind of indication of the status of the page. Some source code control programs store the page status in comments, for example. Text in comments is ignored when the HTML file is parsed; comments never show up onscreen—that's why they're ‘‘. Comments look like the following:

Example 2-6 Comment information

```
<!DOCTYPE html>
<html>
<head>
</head>
<body>
<!-- This is a comment -->
<!-- users can view your comments -->
</body>
</html>
```

As you can see from Figure 2-4 HTML comments displayed within the source for a page., users can view your comments using the View Source functionality in their browsers, so don't put anything in comments that you don't want them to see.

Figure 2-4 HTML comments displayed within the source for a page.

At this point, you know enough to get started creating simple HTML pages. You understand what HTML is, you've been introduced to a handful of tags, and you've even opened an HTML file in your browser.

This exercise shows you how to create an HTML file that uses the tags you've learned about up to this point. It'll give you a feel for what the tags look like when they're displayed onscreen and for the sorts of typical mistakes you're going to make. (Everyone makes them, and that's why using an HTML editor that does the typing for you is often helpful. The editor doesn't forget the closing tags, leave off the slash, or misspell the tag itself.)

So, create a simple example in your text editor. Your example doesn't have to say much of anything; in fact, all it needs to include are the structure tags, a title, a couple of headings, and a paragraph or two. Here's an example:

Example 2-7 Create a real HTML page

```
<!DOCTYPE html>
<html>
<head>
<title>Camembert Incorporated</title>
</head>
<body>
<h1>Camembert Incorporated</h1>
<p>"Many's the long night I dreamed of cheese -- toasted, mostly."
-- Robert Louis Stevenson</p>
<h2>What We Do</h2>
<p>We make cheese. Lots of cheese; more than eight tons of cheese
a year.</p>
<h2>Why We Do It</h2>
<p>We are paid an awful lot of money by people who like cheese.
So we make more.</p>
</body>
</html>
```

2.2 Organizing Information with Lists

In the previous lesson, you learned about the basic elements that make up a web page. In this lesson, I introduce lists, unlike the other tags that have been discussed thus far, are composed of multiple tags that work together. As you'll see, lists come in a variety of types and can be used not only for traditional purposes, like shopping lists or bulleted lists, but also for creating outlines or even navigation for websites.

In this section, you'll learn the following:

- ✓ How to create numbered lists
- ✓ How to create bulleted lists
- ✓ How to create definition lists

Lists are a general-purpose container for collections of things. They come in three varieties. Ordered lists are numbered and are useful for presenting things like your top 10 favorite songs from 2017 or the steps to bake a cake. Unordered lists are not numbered and by default are presented with bullets for each list item. However, these days' unordered lists are often used as a general-purpose container for any list-like collection of items. Yes, they're frequently used for bulleted lists of the kind you might see on a PowerPoint slide, but they're also used for things like collections of navigation links and even pull-down menus. Finally, definition lists are used for glossaries and other items that pair a label with some kind of description.

All the list tags have the following common elements:

- ✓ Each list has an outer element specific to that type of list. For example, <ul> and for unordered lists, <ol> and for ordered lists, or <dl> and </dl> for definition lists.
- ✓ Each list item has its own tag: <dt> and <dd> for the glossary lists, and <li> for the other lists.

2.2.1 Numbered Lists

Numbered lists are surrounded by the <ol>... tags (ol stands for ordered list), and each item within the list is included in the <li>... (list item) tag.

When the browser displays an ordered list, it numbers and indents each of the elements sequentially. You don't have to perform the numbering yourself and, if you add or delete items, the browser renumbers them the next time the page is loaded.

Ordered lists are lists in which each item is numbered or labeled with a counter of some kind (like letters or roman numerals).

Use numbered lists only when the sequence of items on the list is relevant. Ordered lists are good for steps to follow or instructions to the readers, or when you want to rank the items in a list. If you just want to indicate that something has a number of elements that can appear in any order, use an unordered list instead.

For example, the following is an ordered list of steps that explain how to boil an egg. You can see how the list is displayed in a browser in Figure 2-5 An ordered list in HTML

Example 2-8 Ordered list

```
<h1>How to Boil an Egg</h1>
<ol>
<li>Put eggs in a pot filled with cold water</li>
<li>Bring the water to a boil</li>
<li>Take the pot off the heat, cover it, and let it sit for
12 minutes</li>
<li>Remove the eggs from the hot water and cool them by
running water over them or placing them in a bowl of
ice water to cool off</li>
<li>Peel and eat</li>
</ol>
```

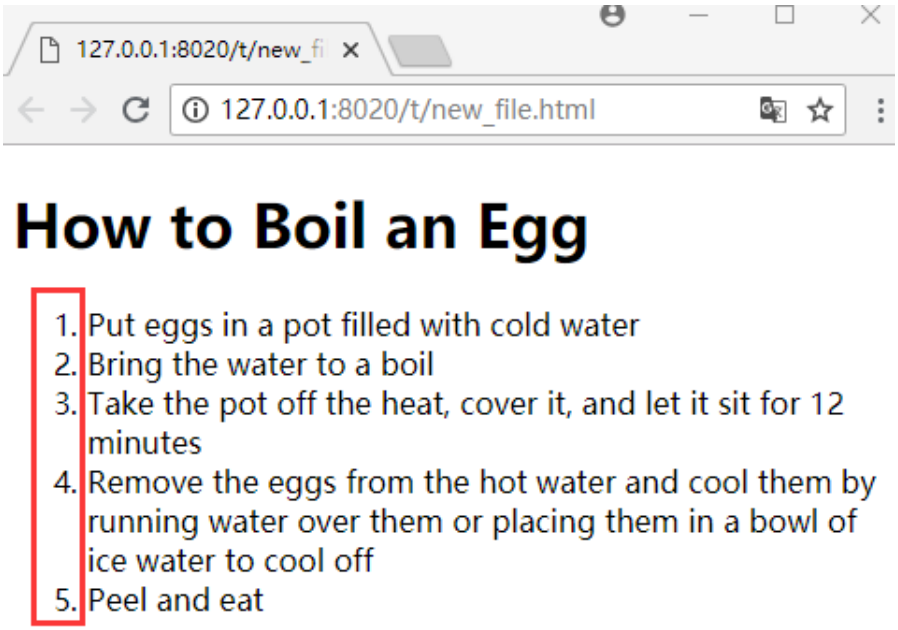


Figure 2-5 An ordered list in HTML

There are two customizations that are specific to ordered lists.

- The first enables you to change the numbering style for the list. There are two ways to change the numbering style: the CSS property list-style-type (see Table 2-1 Ordered List Numbering Styles), and the **type** attribute (see Table 2-2 Ordered List type attribute), which is obsolete in HTML5. If you're creating a new ordered list, you should always use the CSS property, however, you may see existing Web pages in which the **type** attribute is used instead.
- The second enables you to change the numbering itself. You can change it with the start attribute

Table 2-1 Ordered List Numbering Styles

CSS Value	Attribute Value	Description
decimal	1	Standard Arabic numerals (1, 2, 3, 4, and so on)
lower-alpha	a	Lowercase letters (a, b, c, d, and so on)
upper-alpha	A	Uppercase letters (A, B, C, D, and so on)
lower-roman	i	Lowercase Roman numerals (i, ii, iii, iv, and so on)
upper-roman	I	Uppercase Roman numerals (that is, I, II, III, IV, and so on)

Table 2-2 Ordered List type attribute

	Attribute Value	Description
type	1	Standard Arabic numerals (1, 2, 3, 4, and so on)
	a	Lowercase letters (a, b, c, d, and so on)
	A	Uppercase letters (A, B, C, D, and so on)
	i	Lowercase Roman numerals (i, ii, iii, iv, and so on)
	I	Uppercase Roman numerals (that is, I, II, III, IV, and so on)

You can also use the list-style-type property with the <li> tag, changing the numbering type in the middle of the list, but you need to change every list item following it if you want them all to have the same new type. Using the start attribute, you can specify the number or letter with which to start your list. The default starting point is 1, of course. You can change this number by using start. <ol start="4">, for example, would start the list at number 4, whereas <ol style="list-style-type: lower-alpha" start="3"> would start the numbering with c and move through the alphabet from there. The value for the start attribute should always be a decimal number, regardless of the numbering style being used.

For example (Example 2-9 Customization of ordered list), you can list the last six months of the year and start numbering with the Roman numeral VII as follows. The results appear in Figure 2-6 an ordered list with an alternative numbering style and starting number.

Example 2-9 Customization of ordered list

```
<p>The Last Six Months of the Year (and the Beginning of the Next
Year):</p>
       <ol style="list-style-type: upper-roman;" start="7">
          <li>July</li>
          <li>August</li>
          <li>September</li>
          <li>October</li>
          <li>November</li>
          <li>December</li>
          <li style="list-style-type: lower-roman;">January</li>
       </ol>
```

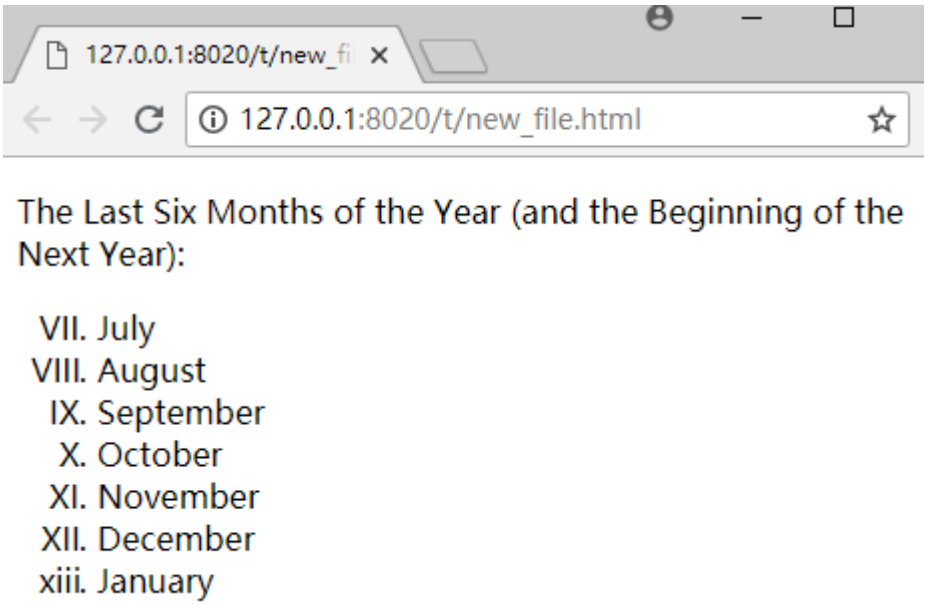


Figure 2-6 an ordered list with an alternative numbering style and starting number

2.2.2 Unordered lists

Unordered lists are often referred to as bulleted lists. Instead of being numbered, each element in the list has the same marker. The markup to create an unordered list looks just like an ordered list except that the list is created by using <ul>... tags rather than ol. The elements of the list are placed within <li> tags, just as with ordered lists.

Browsers have standardized on using a solid bullet to mark each item in an unordered list by default. Text browsers usually use an asterisk for these lists. The following input and output example shows an unordered list. Figure 2-7 an unordered list shows the results in a browser.

Example 2-10 an unordered list

```
<p>Things I like to do in the morning:</p>
<ul>
<li>Drink a cup of coffee</li>
<li>Watch the sunrise</li>
<li>Listen to the birds sing</li>
<li>Hear the wind rustling through the trees</li>
<li>Curse the construction noises for spoiling the peaceful mood</li>
</ul>
```

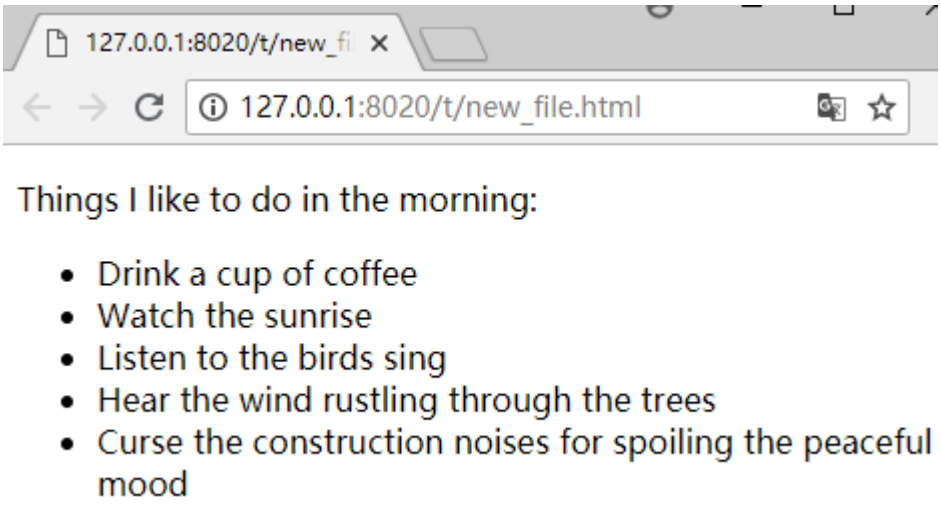


Figure 2-7 an unordered list

As with ordered lists, unordered lists can be customized using the type attribute or the list-style-type property. As mentioned in the section on ordered lists, the type attribute is no longer valid for HTML5. The bullet styles are as follows:

- ✓ "disc"—A disc or bullet; this style is the default.
- ✓ "square"—Obviously, a square rather than a disc.
- ✓ "circle"—As compared with the disc, which most browsers render as a filled circle, this value should generate an unfilled circle.

In this case, the values for list-style-type and for the type attribute are the same. In the following input and output example, you see a comparison of these three types as rendered in a browser as Figure 2-8 Specifies styles of unordered lists.

Example 2-11 Specifies styles of unordered lists

```
<ul style="list-style-type: disc">
<li>DAT - Digital Audio Tapes</li>
<li>CD - Compact Discs</li>
<li>Cassettes</li>
</ul>
<ul style="list-style-type: square">
<li>DAT - Digital Audio Tapes</li>
<li>CD - Compact Discs</li>
<li>Cassettes</li>
</ul>
<ul type="circle">
<li>DAT - Digital Audio Tapes</li>
<li>CD - Compact Discs</li>
```

```
<li>Cassettes</li>
</ul>
```

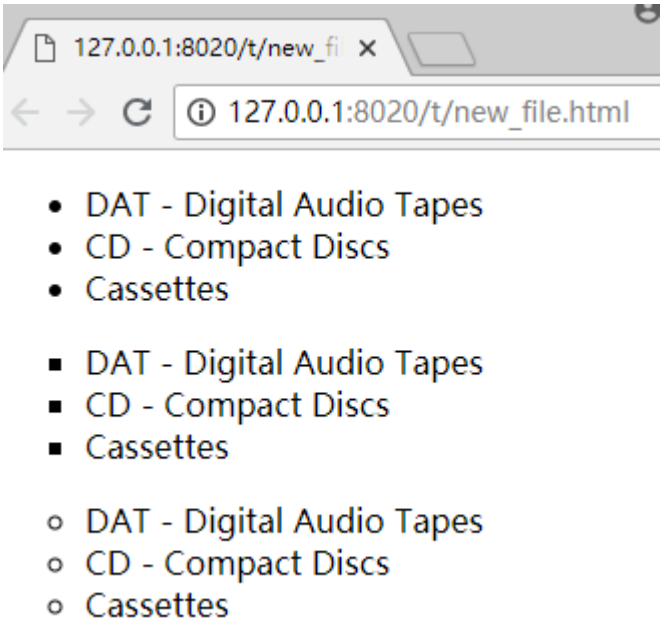


Figure 2-8 Specifies styles of unordered lists

If you don't like any of the bullet styles used in unordered lists, you can substitute an image of your own choosing in place of them. To do so, use the list-style-image property. By setting this property, you can use an image of your choosing for the bullets in your list. Here's an example:

```
<ul style="list-style-image: url(/bullet.gif);">
<li>Example</li>
</ul>
```

2.2.3 Definition lists

Definition lists differ slightly from other lists. Each list item in a definition list has two parts:

- ✓ A term
- ✓ The term's definition

Each part of the glossary list has its own tag: <dt> for the term (definition term), and <dd> for its definition (definition description). <dt> and <dd> usually occur in pairs, although most browsers can handle single terms or definitions. The entire glossary list is indicated by the tags <dl>...</dl> (definition list). The following is a glossary list example with a set of herbs and descriptions of how they grow

Example 2-12 a definition list

```
<dl>
<dt>Basil</dt>
<dd>Annual. Can grow four feet high; the scent of its tiny
whiteflowers is heavenly</dd>
<dt>Oregano</dt>
<dd>Perennial. Sends out underground runners and is difficult to
get rid of once established.</dd>
<dt>Coriander</dt>
<dd>Annual. Also called cilantro, coriander likes cooler weather of
spring and fall.</dd>
</dl>
```

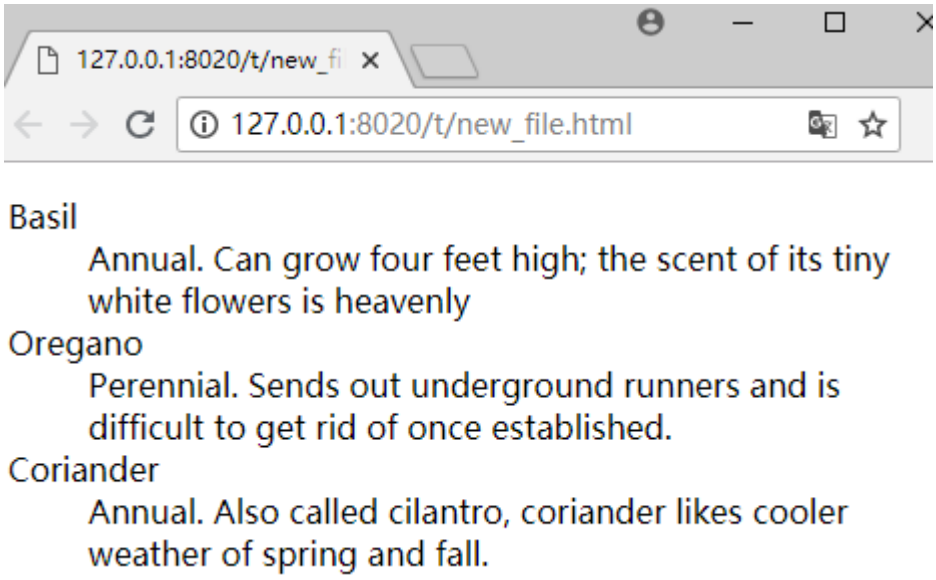


Figure 2-9 a definition list

Definition lists usually are formatted in browsers with the terms and definitions on separate lines, and the left margins of the definitions are indented.

You don't have to use definition lists for terms and definitions, of course. You can use them anywhere that the same sort of list is needed. Here's an example involving a list of frequently asked questions:

```
<dl>
<dt>What is the WHATWG?</dt>
<dd>The Web Hypertext Application Technology Working Group (WHATWG)
is a growing
community of people interested in evolving the Web. It focuses
primarily on the
```

```
development of HTML and APIs needed for Web applications.</dd>
<dt>What is the WHATWG working on?</dt>
<dd>The WHATWG's main focus is HTML5. The WHATWG also works on Web
Workers and
occasionally specifications outside WHATWG space are discussed on
the WHATWG
mailing list and forwarded when appropriate.</dd>
<dt>How can I get involved?</dt>
<dd>There are lots of ways you can get involved, take a look and
see What you can
do!</dd>
<dt>Is participation free?</dt>
<dd>Yes, everyone can contribute. There are no memberships fees
involved, it's an
open process. You may easily subscribe to the WHATWG mailing lists.
You may also
join the the W3C's new HTMLWG by going through the slightly longer
application
process.</dd>
</dl>
```

2.2.4 Nesting lists

What happens if you put a list inside another list? Nesting lists is fine as far as HTML is concerned; just put the entire list structure inside another list as one of its elements. The nested list just becomes another element of the first list, and it's indented from the rest of the list. Lists like this work especially well for menu-like entities in which you want to show hierarchy (for example, in tables of contents) or as outlines.

Indenting nested lists in HTML code itself helps show their relationship to the final layout:

Example 2-13 a nesting lists

```
<ol>
<li>WWW</li>
<li>Organization</li>
<li>Beginning HTML</li>
<li>
<ul>
<li>What HTML is</li>
<li>How to Write HTML</li>
<li>Doc structure</li>
<li>Headings</li>
<li>Paragraphs</li>
```

```
<li>Comments</li>
</ul>
</li>
<li>Links</li>
<li>More HTML</li>
</ol>
```

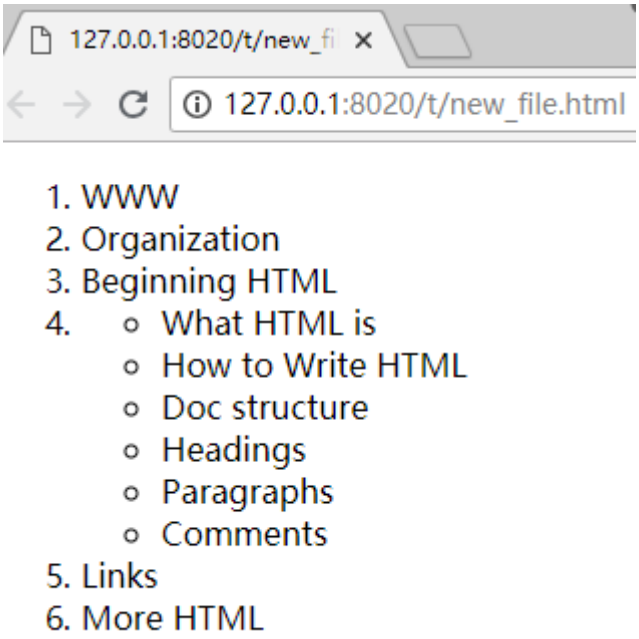


Figure 2-10 a nesting lists

Many browsers format nested ordered lists and nested unordered lists differently from their enclosing lists. They might, for example, use a symbol other than a bullet for a nested list, or they might number the inner list with letters (a, b, c) rather than numbers. Don't assume that this will be the case.

In this section, you got a look at HTML lists. Lists are a core structural element for presenting content on web pages and can be used for everything from the list of steps in a process to a table of contents to a structured navigation system for a website. They come in three varieties: ordered lists, which are numbered; unordered lists, which by default are presented bullets; and definition lists, which are presented as a series of terms and the definitions associated with them.

Not only are there CSS properties specifically associated with lists, but lists can also be styled using properties that apply to any block-level element, like lists and list items. The full list of HTML tags discussed in this section is shown in Table 2-3 summary of lists.

Table 2-3 summary of lists

Tag	Attribute	**Use**

** ... **		... An ordered (numbered) list. Each of the items in the list begins with .
	type	Specifies the numbering scheme to use in the list. Replaced with CSS in HTML5.
	start	Specifies at which number to start the list.
** ... **		An unordered (bulleted or otherwise marked) list. Each of the items in the list begins with .
	type	Specifies the bulleting scheme to use in the list. Replaced with CSS in HTML5.
** ... **		Individual list items in ordered, unordered lists.
	type	Resets the numbering or bulleting scheme from the current list element. Applies only to and lists. Replaced with CSS in HTML5.
	value	Resets the numbering in the middle of an ordered ( ) list.
<dl>...</dl>		A glossary or definition list. Items in the list consist of pairs of elements: a term and its definition.
<dt>...</dt>		The term part of an item in a glossary list.
<dd>...</dd>		The definition part of an item in a glossary list.

2.3 Links

After finishing the preceding lesson, you now have a couple of pages that have some headings, text, and lists in them. These pages are all well and good, but rather boring. The real fun starts when you learn how to create hypertext links and link your pages to the Web. In this lesson, you'll learn just that. Specifically, you'll learn about the following:

- ✓ The HTML link tag (<a>) and its various parts
- ✓ How to link to other pages using relative and absolute paths
- ✓ How to link to other pages on the Web using URLs
- ✓ How to use links and anchors to link to specific locations inside pages

To create a link in HTML, you need two things:

- The name of the file (or the URL) to which you want to link
- The text that will serve as the clickable link

Only the text included within the link tag is actually visible on your page. When your readers click the link, the browser loads the URL associated with the link.

To create a link in an HTML page, you use the HTML link tag <a>.... The <a> tag is also called an anchor tag because it also can be used to create anchors for links. The <a> tag uses attributes to define the link. It looks something like the following:

```
<a href="menu.html" >
```

The additional attributes (in this example, href) describe the link itself. The attribute you'll probably use most often is the href attribute, which is short for `hypertext reference`. You use the href attribute to specify the name or URL to which this link points.

Like most HTML tags, the link tag also has a closing tag, . All the text between the opening and closing tags will become the actual link on the screen and be highlighted, underlined, or otherwise marked as specified in the page's style sheet when the web page is displayed. That's the text you or your readers will click to follow the link to the URL in the href attribute.

Figure 2-11 a link on a page shows the parts of a typical link using the <a> tag, including the href, the text of the link, and the closing tag.

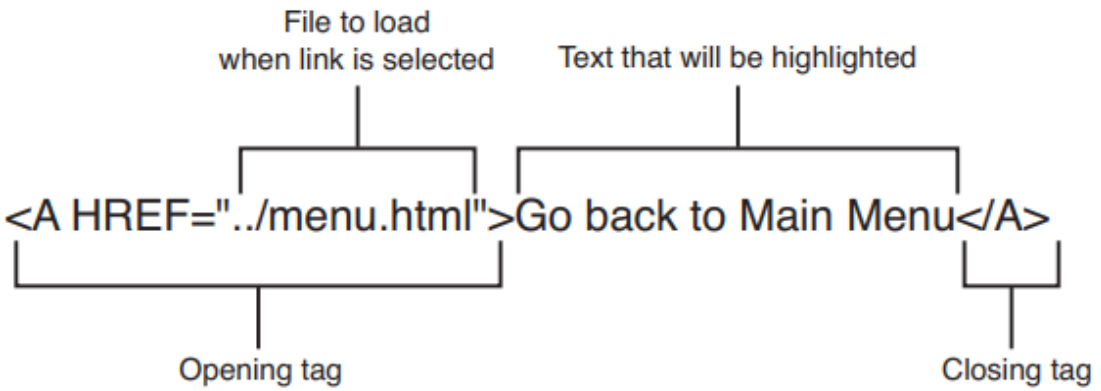


Figure 2-11 a link on a page

The following example shows a simple link and what it looks like (Figure 2-12).

```
Go back to <a href="menu.html">Main Menu</a>
```

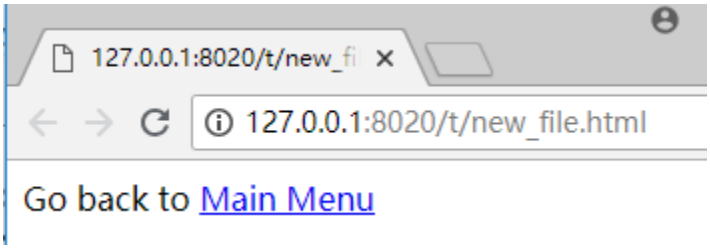


Figure 2-12 a simple link

2.3.1 Linking Two Pages

Now you can try a simple example with two HTML pages on your local disk. You'll need your text editor and your web browser for this exercise. Because both the pages you'll work with are on your local disk, you don't need to be connected to the Internet.

Create two HTML pages and save them in separate files. Here's the code for the two HTML files I created for this section, which I called menu.html and claudius.html. What your two pages look like or what they're called really doesn't matter. However, make sure that you insert your own filenames if you're following along with this example.

The following is the first file, called menu.html:

Example 2-14 menu.html

```
<!DOCTYPE html>
<html>
<head>
<title>The Twelve Caesars</title>
</head>
<body>
<h1>"The Twelve Caesars" by Suetonius</h1>
<p>Seutonius (or Gaius Suetonius Tranquillus) was born circa A.D.
70 and died sometime after A.D. 130. He composed a history of the
twelve Caesars from Julius to Domitian (died A.D. 96). His work was
a significant contribution to the best-selling novel and television
series "I, Claudius." Suetonius' work includes biographies of the
following Roman emperors:</p>
<ul>
<li>Julius Caesar</li>
<li>Augustus</li>
<li>Tiberius</li>
<li>Gaius (Caligula)</li>
<li>Claudius</li>
<li>Nero</li>
```

```
<li>Galba</li>
<li>Otho</li>
<li>Vitellius</li>
<li>Vespasian</li>
<li>Titus</li>
<li>Domitian</li>
</ul>
</body>
</html>
```

The list of menu items (Julius Caesar, Augustus, and so on) will be links to other pages.

For now, just type them as regular text; you'll turn them into links later.

The following is the second file, claudius.html

Example 2-15 claudius.html

```
<!DOCTYPE html>
<html>
<head>
<title>The Twelve Caesars: Claudius</title>
</head>
<body>
<h2>Claudius Becomes Emperor</h2>
<p>Claudius became Emperor at the age of 50. Fearing the attack of
Caligula's assassins, Claudius hid behind some curtains. After a
guardsman discovered him, Claudius dropped to the floor, and then
found himself declared Emperor.</p>
<h2>Claudius is Poisoned</h2>
<p>Most people think that Claudius was poisoned. Some think his
wife Agrippina poisoned a dish of mushrooms (his favorite food).
His death was revealed after arrangements had been made for her son,
Nero, to succeed as Emperor.</p>
<p>Go back to Main Menu</p>
</body>
</html>
```

Make sure that both of your files are in the same directory or folder. If you haven't called them menu.html and claudius.html, make sure that you take note of the filenames because you'll need them later.

Create a link from the menu file to the claudius file. Edit the menu.html file, and put the cursor at the following line:

```
<li>Claudius</li>
```

You'll want to nest the <a> tag inside the existing <li> tag. First, put in the link tags themselves (the <a> and tags) around the text that you want to use as the link:

```
<li><a>Claudius</a></li>
```

Now add the URL that you want to link to as the href part of the opening link tag. In this case the URL is simply a pointer to the other file you've created. Enclose the name of the file in quotation marks (straight quotes ["], not curly or typesetter's quotes [“]), with an equal sign between href and the name. Filenames in links are case sensitive, so make sure that the filename in the link is identical to the name of the file you created. (Claudius.html is not the same file as claudius.html; it has to be exactly the same case.) Here I've used claudius.html; if you used different files, use those filenames.

```
<li><a href="claudius.html">Claudius</a></li>
```

Now start your browser and open the menu.html file. The paragraph you used as your link should now show up as a link that is in a different color, underlined, or otherwise highlighted. Figure 2-13 shows how it looked when I opened it.

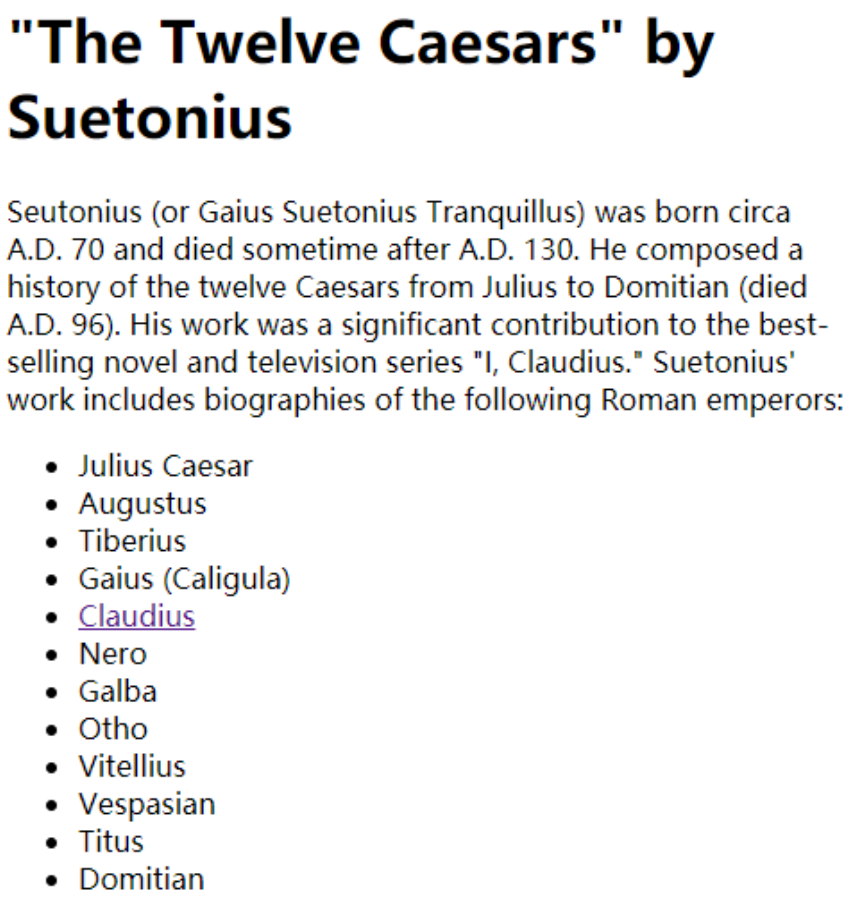


Figure 2-13 menu.html

Now when you click the link, your browser should load and display the claudius.html page, as shown in Figure 2-14.

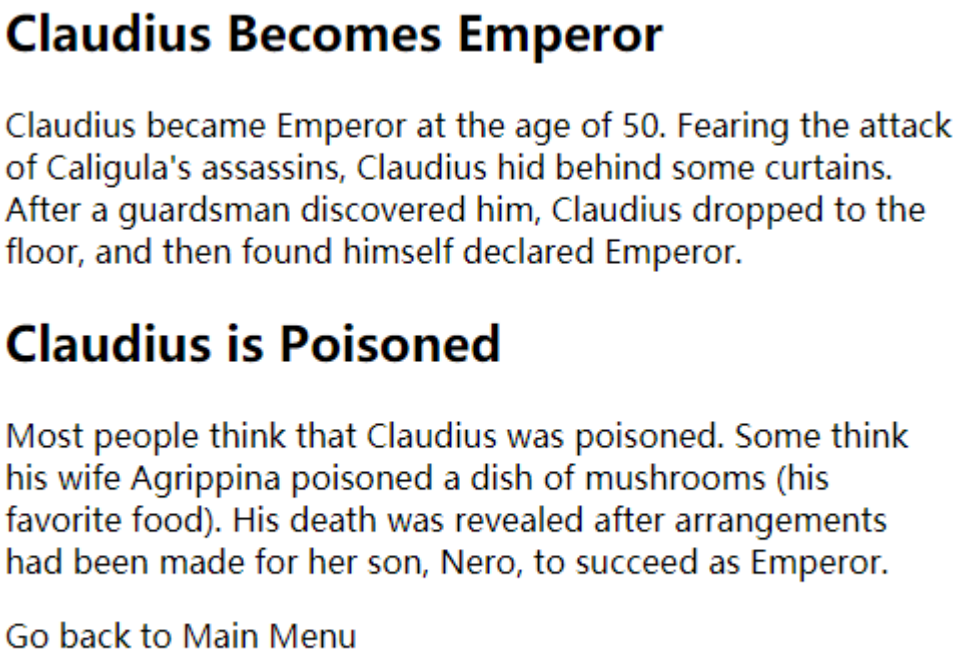


Figure 2-14 claudius.html

If your browser can't find the file when you click on the link, make sure that the name of the file in the href part of the link tag is the same as the name of the file on the disk, uppercase and lowercase match, and both files are in the same directory. Remember to close your link, using the tag, at the end of the text that serves as the link. Also make sure that you have quotation marks at the beginning and end of the filename (sometimes you can easily forget) and that both quotation marks are ordinary straight quotes. All these things can confuse the browser and prevent it from finding the file or displaying the link properly.

Now you can create a link from the caesar page back to the menu page. A paragraph at the end of the claudius.html page is intended for just this purpose:

```
<p>Go back to Main Menu</p>
```

Add the link tag with the appropriate href to that line, such as the following in which menu.html is the original menu file:

```
<p><a href="menu.html">Go back to Main Menu</a></p>
```

Now when you reload the Claudius file, the link will be active, and you can jump between the menu and the detail page by clicking on those links.

2.3.2 Linking Local Pages Using Relative and Absolute Pathnames

The example in the preceding section shows how to link together pages that are contained in the same folder or directory on your local disk. This section continues that thread, linking pages that are still on the local disk but might be contained in different directories or folders on that disk.

When you specify just the filename of a linked file within quotation marks, as you did earlier, the browser looks for that file in the same directory as the current file. This is true even if both the current file and the file being linked to be on a server somewhere else on the Internet; both files are contained in the same directory on that server. It is the simplest form of a relative pathname.

Relative pathnames point to files based on their locations relative to the current file. They can include directory names, or they can point to the path you would take to navigate to that file if you started at the current directory or folder. A pathname might, for example, include directions to go up two directory levels and then go down two other directories to get to the file.

To specify relative pathnames in links, you must use UNIX -style paths regardless of the system you actually have. You therefore separate directory or folder names with forward slashes (/), and you use two dots to refer generically to the directory above the current one (…).

Table 2-4 Pathname Means

Tag	Use
href=”file.html”	file.html is located in the current directory.
href=”files/file.html”	file.html is located in the directory called files (and the files directory is located in the current directory).
href=”files/morefiles/file.html”	file.html is located in the morefiles directory, which is located in the files directory, which is located in the current directory.
href=”../file.html”	file.html is located in the directory one level up fromthe current directory (the parent directory).
href=”../../files/file.html”	file.html is located two directory levels up, in the directory files

You can also specify the link to another page on your local system by using an absolute pathname.

Absolute pathnames point to files based on their absolute locations on the file system. Whereas relative pathnames point to the page to which you want to link by describing its location relative to

the current page, absolute pathnames point to the page by starting at the top level of your directory hierarchy and working downward through all the intervening directories to reach the file. Absolute pathnames always begin with a slash, which is the way they're differentiated from relative pathnames. Following the slash are all directories in the path from the top level to the file you are linking.

The answer to that question is, "It depends." If you have a set of files that link only to other files within that set, using relative pathnames makes sense. On the other hand, if the links in your files point to files that aren't within the same hierarchy, you probably want to use absolute links. Generally, a mix of the two types of links makes the most sense for complex sites.

The rule of thumb I generally use is that if pages are part of the same collection, I use relative links, and if they're part of different collections, I use absolute links.

2.3.3 Links to Other Documents on the Web

So, now you have a whole set of pages on your local disk, all linked to each other. In some places in your pages, however, you want to refer to a page somewhere else on the Internet—for example, to The First Caesars page by Dr. Ellis Knox at Boise State University for more information on the early Roman emperors. You also can use the link tag to link those other pages on the Internet, which I'll call remote pages. Remote pages are contained somewhere on the Web other than the system on which you're currently working.

The HTML code you use to link pages on the Web looks exactly the same as the code you use for links between local pages. You still use the <a> tag with a href attribute, and you include some text to serve as the link on your web page. Rather than a filename or a path in the href, however, you use the URL of that page on the Web, as Figure 2-15 shows.

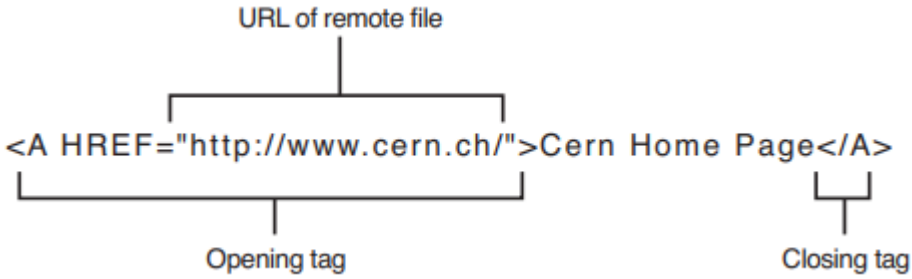


Figure 2-15 link to remote file

Go back to those two pages you linked earlier today—the ones about the Caesars. The menu.html file contains several links to other local pages that provide information about 12 Roman emperors.

Now suppose that you want to add a link to the bottom of the menu file to point to The Twelve Caesars page on Wikipedia, whose URL is http://en.wikipedia.org/wiki/The_Twelve_Caesars.

First, add the appropriate text for the link to your menu page, as follows:

```
<p><i>The Twelve Caesars</i> article in Wikipedia has more
```

```
information on these Emperors. </p>
```

What if you don't know the URL of the home page for The Twelve Caesars page (or the page to which you want to link), but you do know how to get to it by following several links on several different people's home pages? Not a problem. Use your browser to find the home page for the page to which you want to link. Figure 6.6 shows what The Twelve Caesars page looks like in a browser.

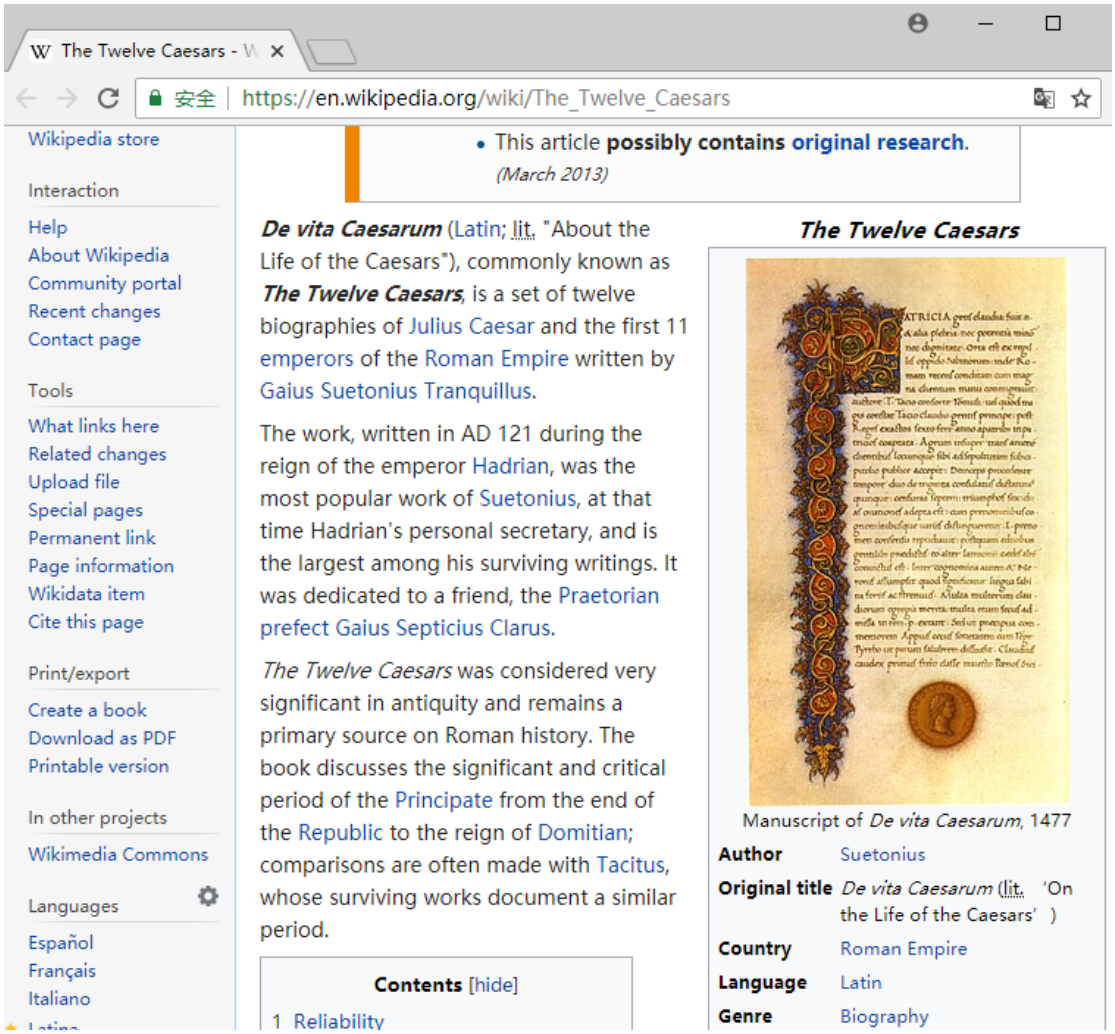


Figure 2-16 the twelve Caesars

You can find the URL of the page you're currently viewing in your browser in the address box at the top of the browser window. To find the URL for a page you want to link to, use your browser to go to the page, copy the URL from the address field, and paste it into the href attribute of the link tag. No typing!

After you have the URL of the page, you can construct a link tag in your menu file and paste the appropriate URL into the link, like this:

```
<p>"<em><a href="http://en.wikipedia.org/wiki/The_Twelve_Caesars">
The Twelve Caesars</a></em>" article in Wikipedia has more
information on these Emperors.</p>
```

In that code I also italicized the title of the page using the tag.

Of course, if you already know the URL of the page to which you want to link, you can just type it into the href part of the link. Keep in mind, however, that if you make a mistake, your browser won't be able to find the file on the other end. Many URLs are too complex for humans to be able to remember them; I prefer to copy and paste whenever I can to cut down on the chances of typing URLs incorrectly.

Figure 2-17shows how the menu.html file, with the new link in it, looks when it is displayed.

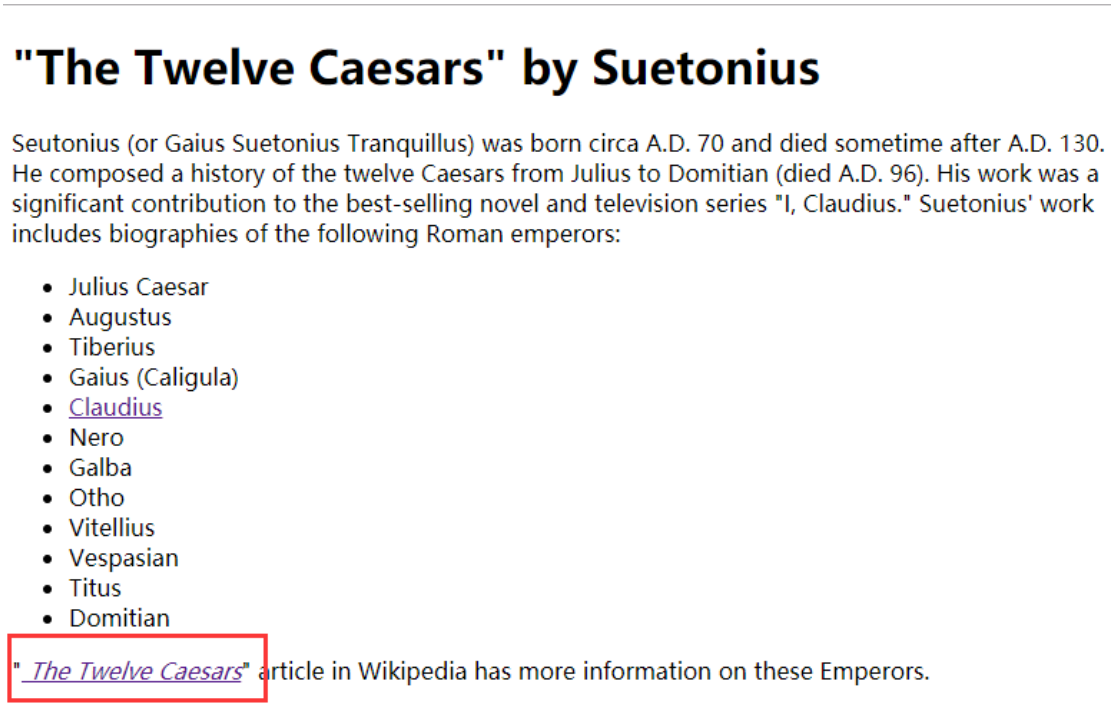


Figure 2-17 twelve Caesars link

2.3.4 Linking to Specific Places within Documents

The links you've created so far in this lesson have been from one point in a page to another page. But what if, rather than linking to that second page in general, you want to link to a specific place within that page—for example, to the fourth major section down? You can do so by referring to the ID of the element you want to link to specifically in the URL in your link. When you follow the link with your browser, the browser will load the second page and then scroll down to the element you specify. Figure 2-18 shows an example.

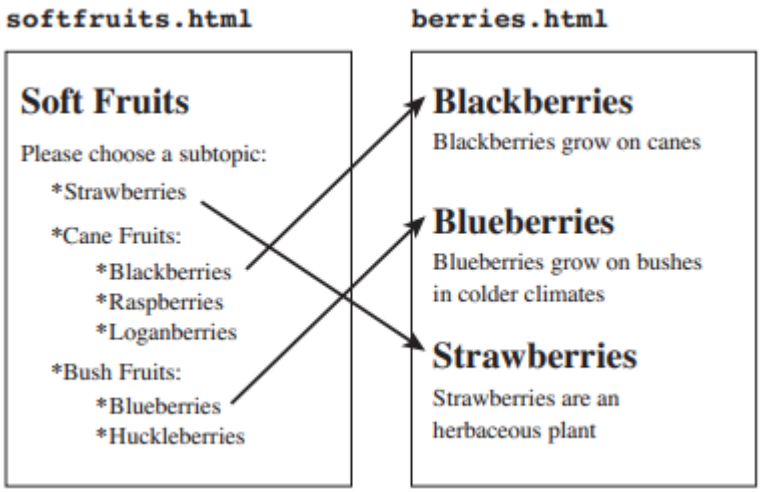


Figure 2-18 links and anchors

You can use links to jump to a specific element within the same page. For example, you can assign `IDs` to the headings at the beginning of each section and include a table of contents at the top of the page that has links to the sections. The id attribute can be used with any element on a page. The only requirement is that each ID is unique within that page. For example, here's a heading with an ID:

```
<h2 id="contents">Table of Contents</a>
```

When you create links using <a>, the link has two parts: the href attribute in the opening <a> tag, and the text between the opening and closing tags that serves as a hot spot for the link.

For example, to create an anchor at the section of a page labeled `Part 4`, you might add an ID `part4` to the heading, similar to the following:

```
<h1 id="part4">Part Four: Grapefruit from Heaven</h1>
```

To point to an anchor in a link, use the same form of link that you would when linking to the whole page, with the filename or URL of the page in the href attribute. After the name of the page, however, include a hash sign (#) and the ID of the element exactly as it appears in the id attribute of that element (including the same uppercase and lowercase characters!), as follows:

```
<a href="mybigdoc.html#part4">Go to Part 4</a>
```

This link tells the browser to load the page mybigdoc.html and then to scroll down to the anchor named part4. The text inside the anchor definition will appear at the top of the screen.

Before browsers supported linking to elements directly using their IDs, you had to use the `name` attribute of the <a> tag to create anchors on the page to which you could link. Rather than including the href attribute in your <a> tag to link to a location, you included the name attribute to indicate that the <a> was an anchor to which someone could link.

For example, you would write the previous example as follows:

```
<h1><a name="part4"></a>Part Four: Grapefruit from Heaven</h1>
```

The tag wouldn't produce a visible change on the page, but it would provide an anchor to which you could link. Best practices recommend that you avoid using the name attribute and use the ID

attribute instead. You can use the ID attribute on any HTML element, not just the <a> tag. However, you may still encounter old markup that uses the <a> tag in this way.

2.3.5 Linking Sections between Two Pages

Now let's create an example with two pages. These two pages are part of an online reference to classical music, in which each web page contains all the references for a particular letter of the alphabet (a.html, b.html, and so on). The reference could have been organized such that each section is its own page. Organizing it that way, however, would have involved several pages to manage, as well as many pages the readers would have to load if they were exploring the reference. Bunching the related sections together under lettered groupings is more efficient in this case.

The first page you'll look at is for M; the first section looks like the following in HTML:

Example 2-16 part M music

```
<!DOCTYPE html>
<html>
<head>
<title>Classical Music: M</title>
</head>
<body>
<h1>M</h1>
<h2>Madrigals</h2>
<ul>
<li>William Byrd, <em>This Sweet and Merry Month of May</em></li>
<li>William Byrd, <em>Though Amaryllis Dance</em></li>
<li>Orlando Gibbons, <em>The Silver Swan</em></li>
<li>Claudio Monteverdi, <em>Lamento d'Arianna</em></li>
<li>Thomas Morley, <em>My Bonny Lass She Smileth</em></li>
<li>Thomas Weelkes, <em>Thule, the Period of Cosmography</em></li>
<li>John Wilbye, <em>Sweet Honey-Sucking Bees</em></li>
</ul>
<p>Secular vocal music in four, five and six parts, usually a
capella.15th-16th centuries.</p>
<p><em>See Also</em>
Byrd, Gibbons, Lassus, Monteverdi, Morley, Weelkes, Wilbye </p>
</body>
</html>
```

Figure 2-19 shows how this section looks when it's displayed.

Figure 2-19 part M of online music

2.4 Formatting text

Over the previous sections, you learned the basics of HTML, including tags used to create page structure and add links. With that background, you're now ready to learn more about what HTML and CSS can do in terms of text formatting and layout. In this section, you'll learn about many of the remaining tags in HTML that you'll need to know to construct pages, including how to use HTML and CSS to do the following:

- ✓ Specify the appearance of individual characters (bold, italic, and underlined)
- ✓ Include special characters (characters with accents, copyright marks, and so on)
- ✓ Create preformatted text (text with spaces and tabs retained)
- ✓ Align text left, right, and centered
- ✓ Change the font and font size
- ✓ Create other miscellaneous HTML text elements, including line breaks, horizontal lines, addresses, and quotations

2.4.1 Character-Level Elements

When you use HTML tags to create paragraphs, headings, or lists, those tags affect that block of text as a whole—changing the font, changing the spacing above and below the line, or adding

characters (in the case of bulleted lists). They're referred to as *block-level elements*.

Character-level elements are tags that affect words or characters within other HTML tags and change the appearance of that text so that it's somehow different from the surrounding text—making it bold or underlined, for example. Tags like , <ul>, and <h1> are block-level elements. The only character-level element you've seen so far is the <a> tag.

To change the appearance of a set of characters within text, you can use one of two methods: *semantic HTML tags* or *Cascading Style Sheets* (CSS).

Semantic tags describe the meaning of the text within the tag, not how it should look in the browser. For example, semantic HTML tags might indicate a definition, a snippet of code, or an emphasized word. This can be a bit confusing because there are de facto standards that correlate each of these tags with a certain visual style. In other words, even though a tag like would mean different things to different people, most browsers display it in boldface, but it has the semantic meaning of strong emphasis. Each character style tag has both opening and closing sides and affects the text within those two tags. Table 2-1 shows the semantic HTML tags:

Table 2-5 sematic HTML tags

Tag	Use
****	This tag indicates that the characters are emphasized in some way.Most browsers display in italics. For example: The anteater is the strangest looking animal,isn't it?
****	With this tag, the characters are more strongly emphasized than with —usually in boldface. Consider the following: Take a left turn at Dee's Hop Stop
<code>	This tag indicates that the text inside is a code sample and displays it in a fixed-width font such as Courier. For example: <code>#include "trans.h"</code>
<samp>	This tag indicates sample text and is generally presented in a fixed width font, like <code>. An example of its usage follows: The URL for that page is <samp>http://www.cern.ch/</samp>
<kbd>	This tag indicates text that's intended to be typed by a user. It's also presented in a fixed-width font. Consider the following: Type the following command: <kbd>find . -name "prune"-

	print</kbd>
<var>	This tag indicates the name of a variable, or some entity to be replaced with an actual value. Often it's displayed as italic or underline and is used as follows: <code>chown</code> <var>your_name for the_file </var>
<dfn>	This tag indicates a definition. <dfn> is used to highlight a word (usually in italics) that will be defined or has just been defined, as in the following example: Styles that are named after how they are actually used are called <dfn>logical styles</dfn>
<cite>	This tag indicates the cited title of a work—usually displayed in italics. It is written as in the following: "use the Force, Luke" <cite>"Star Wars"</cite> (1976)
<abbr>	This tag indicates the abbreviation of a word, as in the following: Use the standard two-letter state abbreviation (such as <abbr>CA</abbr> for California)

The following code snippets demonstrate each of the semantic HTML tags mentioned, Figure 2-20 illustrates how all the tags are displayed.

Example 2-17 semantic HTML tag

```
<p>The anteater is the <em>strangest</em> looking animal, isn't
it?</p>
<p>Take a <strong>left turn</strong> at <strong>Dee's Hop Stop
</strong></p>
<p><code>#include "trans.h"</code></p>
<p>The URL for that page is <samp>http://www.cern.ch/</samp></p>
<p>Type the following command: <kbd>find . -name "prune" -
print</kbd></p>
<p><code>chown </code><var>your_name the_file</var></p>
<p>Styles that are named after how they are used are called
<dfn>logical styles</dfn></p>
<p>Eggplant has been known to cause nausea in some people<cite>
(Lemay, 1994)</cite></p>
<p>Use the standard two-letter state abbreviation (such as
<abbr>CA</abbr> for California)</p>
```

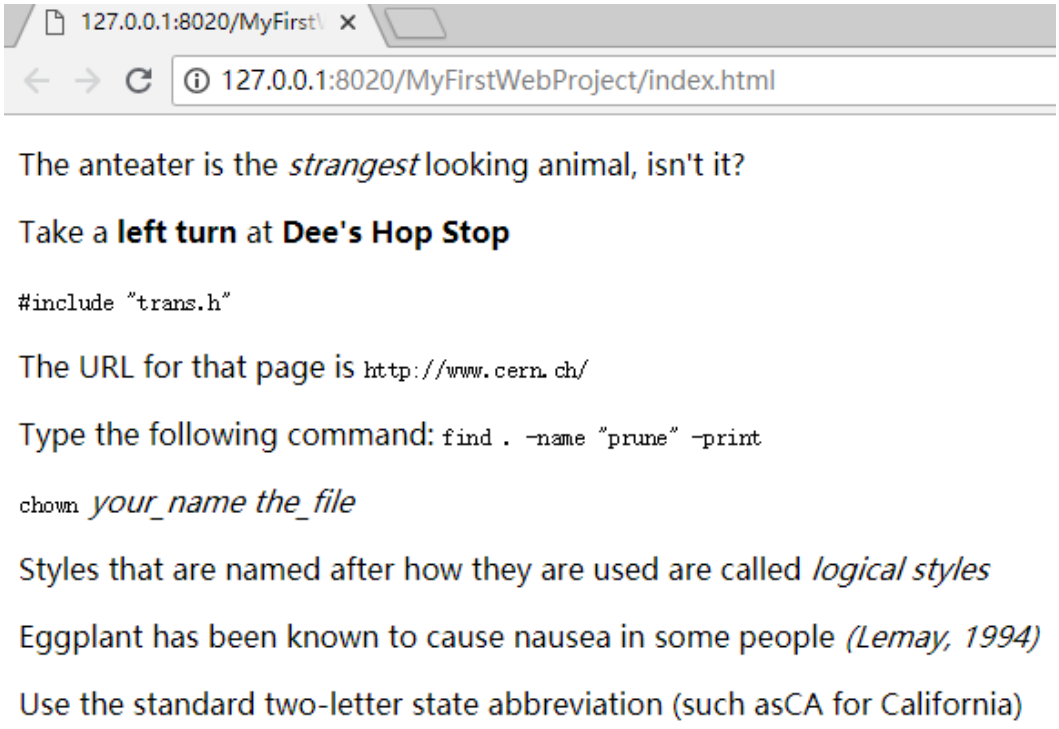


Figure 2-20 sematic HTML tags result

Over time, a number of physical style tags were added to HTML as well. You should avoid using them and use CSS or the semantic equivalents instead, but if you decide to use them, HTML5 has given them semantic meanings:

Table 2-6 character formatting tag

Tag	Use
<u></u>	Specifies that the enclosed text should be rendered as underlined in the browser.
****	Text that is usually displayed as italic
****	Text that is usually bold
<small ></small>	Text that displays as small print
****	Text that displays as subscript
****	Text that displays as superscript

2.4.2 Preformatted Text

Most of the time, text in an HTML file is formatted based on the HTML tags used to mark up that text. I mentioned that any extra whitespace (spaces, tabs, returns) that you include in your HTML source is stripped out by the browser.

The one exception to this rule is the preformatted text tag <pre>. Any whitespace that you put into text surrounded by the <pre> and </pre> tags is retained in the final output. With these tags, the spacing in the text in the HTML source is preserved when it's displayed on the page. The catch is that preformatted text usually is displayed (in graphical displays, at least) in a monospaced font such as Courier. Preformatted text is excellent for displaying code examples in which you want the text formatted with exactly the indentation the author used. Because you can use the <pre> tag to align text by padding it with spaces, you can use it for simple tables. However, the fact that the tables are presented in a monospaced font might make them less than ideal. The following is an example of a table created with <pre>:

Example 2-18 create table with pre tag

```
<pre>
          Diamete    Distance      Time to      Time to
       (miles)      from Sun      Orbit        Rotate
                   (millions
                   of miles)
-------------------------------------------------------------------
---------
Mercury 3100       36           88 days        59 days
Venus   7700       367          225 days      244 days
Earth   7920        93          365 days       24 hrs
Mars    4200       3141         687 days      24 hrs 24 mins
Jupiter  88640      3483        11.9 years      9 hrs 50 mins
Saturn   74500      886         29.5 years      10 hrs 39 mins
Uranus   32000     1782         84 years       23 hrs
Neptune  31000      2793        165 days       15 hrs 48 mins
Pluto     1500      3670         248 years      6 days 7 hrs</pre>
```

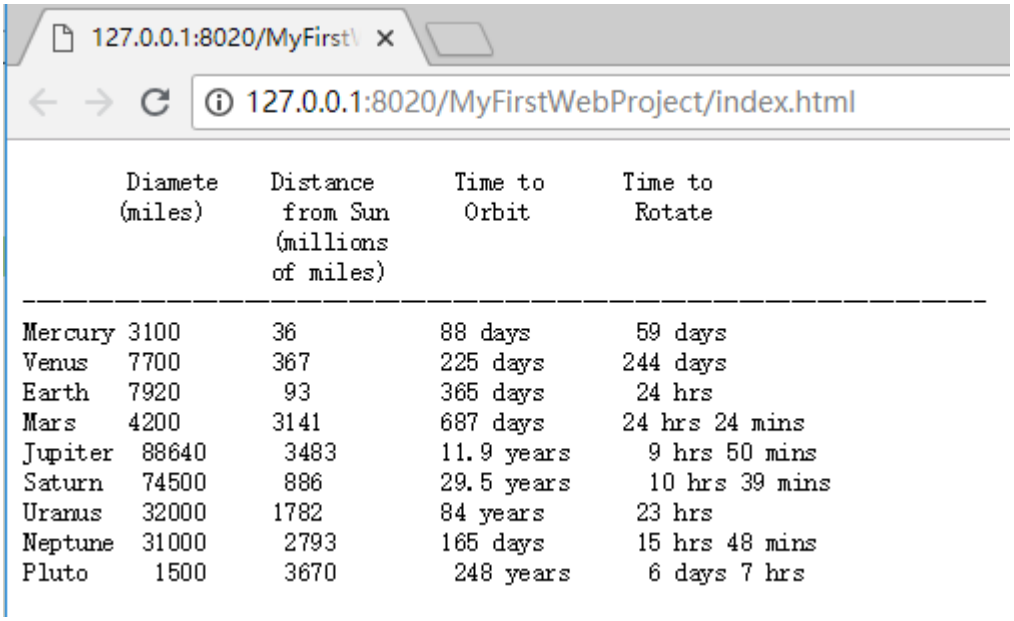


Figure 2-21 create table with pre tag

When you're creating text for the <pre> tag, you can use link tags and character styles but not element tags such as headings or paragraphs. You should break your lines with hard returns and try to keep your lines to 60 characters or fewer. Some browsers might have limited horizontal space in which to display text. Because browsers usually won't reformat preformatted text to fit that space, you should make sure that you keep your text within the boundaries to prevent your readers from having to scroll from side to side.

Be careful with tabs in preformatted text. The actual number of characters for each tab stop varies from browser to browser. One browser might have tab stops at every fourth character, whereas another may have them at every eighth character. You should convert any tabs in your preformatted text to spaces so that your formatting isn't messed up if it's viewed with different tab settings than in the program you used to enter the text.

The <pre> tag is also excellent for converting files that were originally in some sort of text-only form, such as email messages, into HTML quickly and easily. Just surround the entire content of the message within <pre> tags and you have instant HTML, as in the following example:

```
<pre>
To: lemay@lne.com
From: jokes@lne.com
Subject: Tales of the Move From Hell, pt. 1
I spent the day on the phone today with the entire household
services division of northern California, turning off services,
turning on services, transferring services and other such fun things
you have to do when you move. It used to be you just called these
people and got put on hold for an interminable amount of time, maybe
with some nice music, and then you got a customer representative
```

```
who was surly and hard of hearing, but with some work you could
actually get your phone turned off.
</pre>
```

One creative use of the <pre> tag is to create ASCII art for your web pages. The following HTML input and output example shows a simple ASCII-art cow:

Example 2-19 a simple ASCII art

```
<pre>
     (  )
Moo  (oo)
      \/------\
      ||      | \
      || --- W|| *
      ||      ||
</pre>
```

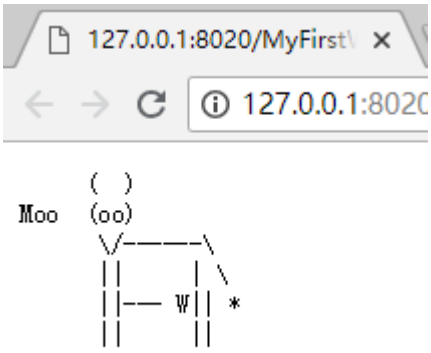


Figure 2-22 result of a simple ASCII art

2.4.3 Horizontal Rules

The <hr> tag, which has no closing tag in HTML and no text associated with it, creates a horizontal line on the page. As of HTML5, the tag has also been given a semantic meaning—thematic break. It's represented by a horizontal line as it always has been, but it has now been ascribed a semantic meaning as well. It represents a change of topic within a section or, for example, a change in scene in a story.

The <hr> tag has no closing tag in HTML. To convert this tag to XHTML, add a space and a forward slash to the end of the tag:

<hr />

If the horizontal line has attributes associated with it, the forward slash still appears at the end of the tag.

The following input shows a horizontal rule used to separate two sections in Emily Bronte's novel Wuthering Heights:

Example 2-20 an example of horizontal rule tag

```
<p>At first, on hearing this account from Zillah, I determined to leave my situation, take a cottage, and get Catherine to come and live with me: but Mr. Heathcliff would as soon permit that as he would set up Hareton in an independent house; and I can see no remedy, at present, unless she could marry again;and that scheme it does not come within my province to arrange.</p>
<hr>
<p>Thus ended Mrs. Dean's story. Notwithstanding the doctor's prophecy, I am rapidly recovering strength; and though it be only the second week in January, I propose getting out on horseback in a day or two, and riding over to Wuthering Heights, to inform my landlord that I shall spend the next six months in London; and, if he likes, he may look out for another tenant to take the place after October. I would not pass another winter here for much.</p>
```

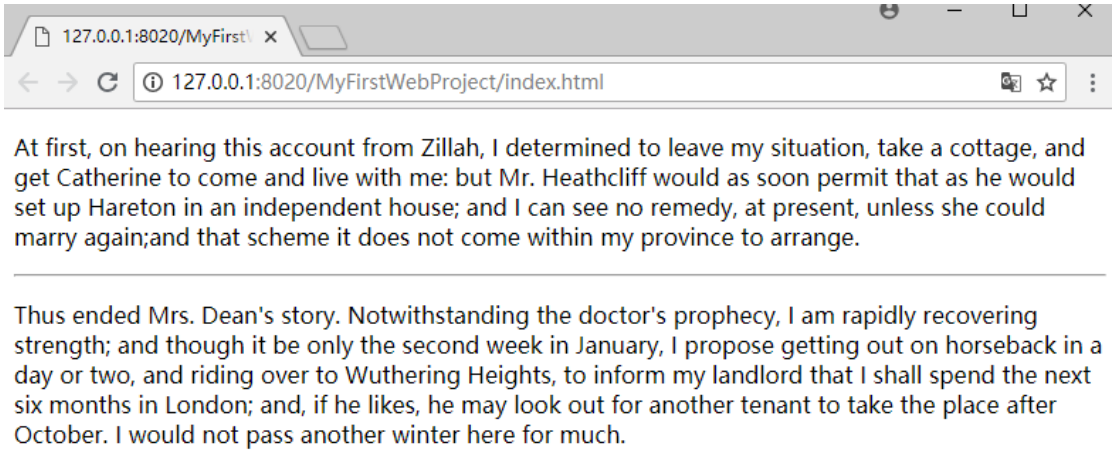


Figure 2-23 an example of horizontal rules

If you're working in HTML5, this one is easy. HTML5 does not support any attributes of the <hr> element other than those supported by all elements. However, past versions of HTML supported a number of attributes that could be used to modify the appearance of a horizontal rule. If you are creating new web pages, you should use CSS to style your horizontal rules. However, you may encounter these attributes in existing HTML.

The size attribute indicates the thickness, in pixels, of the rule line. The default is 2, and this also is the smallest that you can make the rule line. To change the thickness of an <hr> with CSS, use the height property

The width attribute specifies the horizontal width of the rule line. You can specify the exact width of the rule in pixels. You can also specify the value as a percentage of the browser width (for example, 30% or 50%). If you set the width of a horizontal rule to a percentage, the width of the rule will change to conform to the window size if the user resizes the browser window. You should use the width CSS property instead. Figure 2-24 shows the result of the following code, which displays some sample rule line widths.

Example 2-21 Examples of line width and height

```
<h2>100%, Default Size</h2>
<hr>
<h2>75%, Size 2</h2>
<hr width="75%" size="2">
<h2>50%, Size 4</h2>
<hr width="50%" size="4">
<h2>25%, Size 6</h2>
<hr width="25%" size="6">
<h2>10%, Size 8</h2>
<hr width="10%" size="8">
```

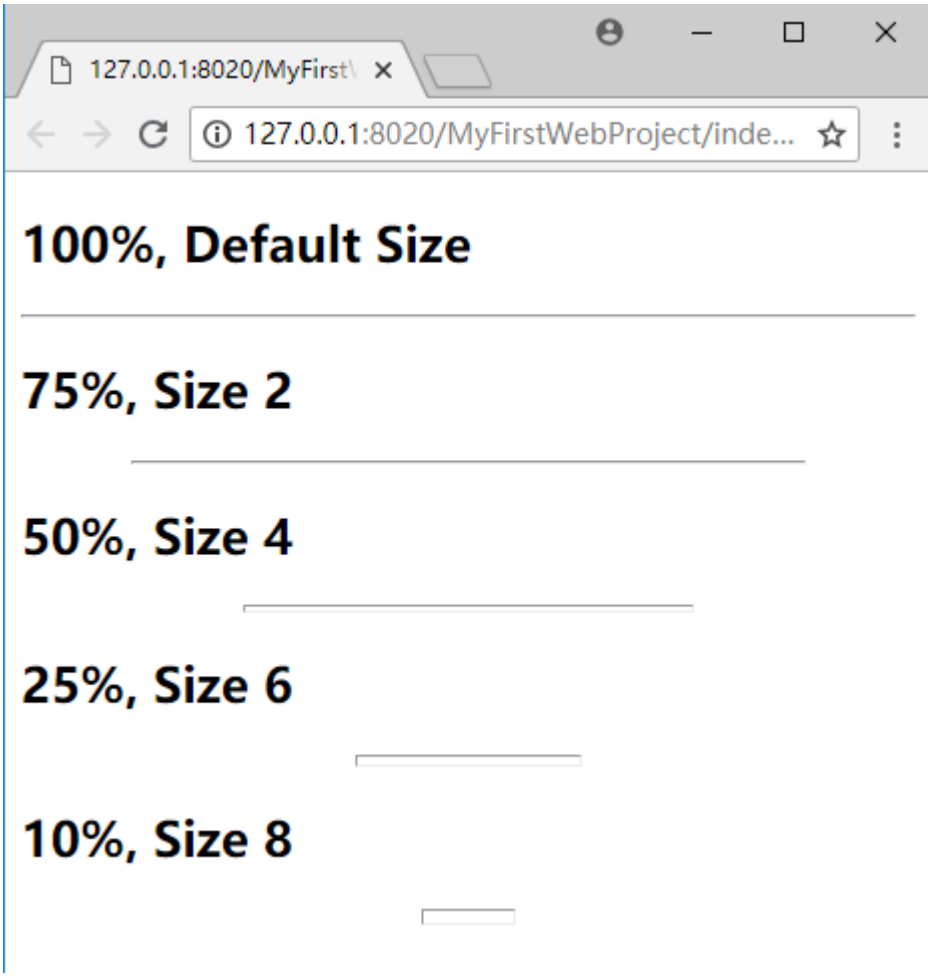


Figure 2-24 line width and height of hr

If you specify a width smaller than the actual width of the browser window, you can also specify the alignment of that rule with the align attribute, making it flush left (align="left"), flush right (align="right"), or centered (align="center"). By default, rule lines are centered. Like all of the other <hr> attributes, the align attribute has been replaced with CSS in HTML5 for all elements that once used it. Alignment will be covered in the following lesson.

Finally, the obsolete noshade attribute causes the browser to draw the rule line as a plain line without the three-dimensional shading.

2.4.4 Line Break

The <br> tag breaks a line of text at the point where it appears. When a web browser encounters a <br> tag, it restarts the text after the tag at the left margin (whatever the current left margin happens to be for the current element). You can use <br> within other elements, such as paragraphs or list items;
 won't add extra space above or below the new line or change the font or style of the

current entity. All it does is restart the text at the next line.

The following example shows a simple paragraph in which each line (except for the last, which ends with a closing <p> tag) ends with a <br>:

Example 2-22 example of line break

```
<p>Tomorrow, and tomorrow, and tomorrow,<br>
Creeps in this petty pace from day to day,<br>
To the last syllable of recorded time;<br>
And all our yesterdays have lighted fools<br>
The way to dusty death. Out, out, brief candle!<br>
Life's but a walking shadow; a poor player,<br>
That struts and frets his hour upon the stage,<br>
And then is heard no more: it is a tale <br>
Told by an idiot, full of sound and fury, <br>
Signifying nothing.</p>
```

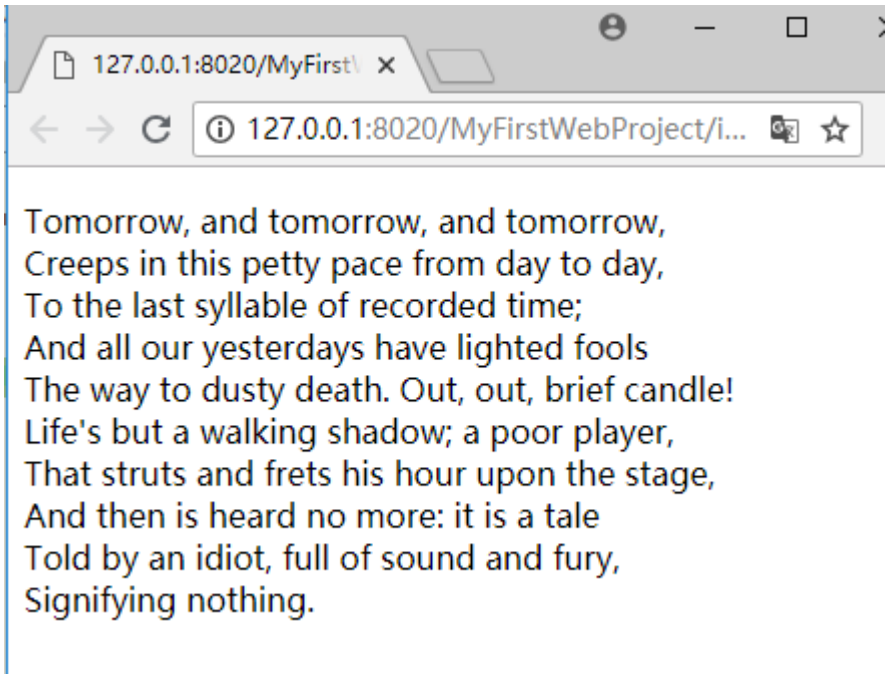


Figure 2-25 line break

2.4.5 Address

The address tag <address> is used to supply contact information on web pages. Address tags usually go at the bottom of the web page and are used to indicate who wrote the web page, whom to contact

for more information, the date, any copyright notices or other warnings, and anything else that seems appropriate.

Signing each of your web pages using the <address> tag is an excellent way to make sure that people can get in touch with you and that visitors who arrive on your web page by way of an external link can see who created it. <address> is a block-level tag, and some browsers italicize the text inside it.

The following input shows an address:

Example 2-23 an address block

```
<address>
Laura Lemay <a href="mailto:lemay@lne.com">lemay@lne.com</a><br />
A service of Laura Lemay, Incorporated <br />
last revised July 10, 2012 <br />
Copyright Laura Lemay 2012 all rights reserved <br />
Void where prohibited. Keep hands and feet inside the vehicle at
all times.
</address>
```

Figure 2-26 an address block

As you can see, by default many browsers italicize the contents of address blocks. To render them in normal text, you can use styles to set the font-style property to normal

2.4.6 Quotation

The <blockquote> tag is used to indicate that a block of text represents an extended quotation. The <blockquote> tag is a block-level element. By default, <blockquote> elements are indented, although that can be changed with CSS. For example, the Macbeth soliloquy I used in the example for line breaks would have worked better as a <blockquote> than as a simple paragraph. Here's an

example:

```
<p>From Shakespeare's <cite>MacBeth</cite>:
<blockquote>Tomorrow, and tomorrow, and tomorrow,<br>
Creeps in this petty pace from day to day,<br>
To the last syllable of recorded time;<br>
And all our yesterdays have lighted fools<br>
The way to dusty death. Out, out, brief candle!<br>
Life's but a walking shadow; a poor player,<br>
That struts and frets his hour upon the stage,<br>
And then is heard no more: it is a tale <br>
Told by an idiot, full of sound and fury, <br>
Signifying nothing.</blockquote>
```

As with paragraphs, you can split lines in a <blockquote> using the line break tag, <br>. The following input example shows an example of this use:

```
<blockquote>
Guns aren't lawful, <br />
nooses give.<br />
gas smells awful.<br />
You might as well live.
</blockquote>
<p>-- Dorothy Parker</p>
```

The <blockquote> tag supports one attribute, cite. The value of the cite attribute is the URL that is the source of the quotation inside the <blockquote> tag. For example, the <blockquote> tag for the preceding Dorothy Parker quotation could point back to the original source using the cite attribute:

```
<blockquote cite="http://www.poetryfoundation.org/poem/174101">
Guns aren't lawful, <br />
nooses give.<br />
gas smells awful.<br />
You might as well live.
</blockquote>
<p>-- Dorothy Parker</p>
```

The cite attribute does not produce visible changes on the page, and for that reason best practices recommend that you not use it. Instead, you should use the <cite> tag directly on the page to indicate the author, title, or URL of the work referenced. For inline quotations, you should use the <q> tag, and, optionally, the cite attribute to indicate them and provide the source URL. The <q> tag does not affect the visual display of the page. Here's an example of how it's used:

```
<p>As Albert Einstein said,
"<q cite="https://en.wikiquote.org/wiki/Albert_Einstein"> I never
think of the future. It comes soon enough.</q>"</p>
```

Finally, the <cite> element is used to cite the author, title, or URL of the work quoted. Like the <q>

tag, it does not affect the visual display of the page in any way, although both can be styled using CSS. And, as previously mentioned, the <cite> tag contents are visible on the page and are the recommended method to cite quotations. Here's how the <cite> tag is used:

```
<p>In Roger Ebert's book <cite>The Great Movies</cite>, he lists
<cite>The Wizard of Oz</cite> as one of the great films.</p>
```

2.4.7 Special Characters

As you've already learned, HTML files are ASCII text and should contain no formatting or fancy characters. In fact, the only characters you should put in your HTML files are the characters that are actually printed on your keyboard. If you have to hold down any key other than Shift or type an arcane combination of keys to produce a single character, you can't use that character in your HTML file. This includes characters you might use every day, such as em dashes and curly quotes. (If you are using a word processor that does automatic curly quotes, you should find another HTML editor that writes text files instead.)

"But wait a minute," you say. "If I can type a character like a bullet or an accented a on my keyboard using a special key sequence, and I can include it in an HTML file, and my browser can display it just fine when I look at that file, what's the problem?" The problem is that the internal encoding your computer does to produce that character (which enables it to show up properly in your HTML file and in your browser's display) probably won't translate to other computers. Someone on the Internet who's reading your HTML file with that funny character in it might end up with some other character or just plain garbage.

So, what can you do? HTML provides a reasonable solution. It defines a special set of codes, called character entities that you can include in your HTML files to represent the characters you want to use. When interpreted by a browser, these character entities display as the appropriate special characters for the given platform and font. Some special characters don't come from the set of extended ASCII characters. For example, quotation marks and ampersands can be presented on a page using character entities even though they're found within the standard ASCII character set. These characters have a special meaning in HTML documents within certain contexts, so they can be represented with character entities to avoid confusing web browsers. Modern browsers generally don't have a problem with these characters, but it's not a bad idea to use the entities anyway.

Before I can talk about how to add special characters to your web page, I first have to talk a little bit about character encoding. When we think of text, we think of characters like "a" or "6" or "&" or a space. Computers, however, think of them as numbered entries in a list. Each of these lists of characters is referred to as a character set. One character set you may have heard of is ASCII, which contains 128 characters, including the upper and lowercase letters, numbers, punctuation, and a number of other special characters like space, carriage return, and tab. The space character is in the 32nd position of the list of ASCII characters. When you convert that to hexadecimal (base 16) notation, it's in position 20. That may ring a bell—back in Lesson 6, you learned that when URL encoding is used, spaces are encoded as %20. That's because encoded characters in URL encoded

are numbered by their position in the list of ASCII characters.

When a web page is displayed, the browser looks up all of the characters on the page in the character set that is being used to display the page. There are a number of ways to specify which character set is used for a page. If none of them are used, the browser displays the page using its default encoding.

You can find a full list of the named entities at

http://www.w3.org/TR/2011/WD-html5-20110113/named-character-references.html

Table 2-7 shows some named entities for special character

Table 2-7 special character

Entities	Result	Desciption
"	“	Quotation mark. (Double.)
&	&	Ampersand
<	<	Less-than sign.
>	>	Greater-than sign.
·	&middot;	Middle dot.
×	&times;	Mulitplication sign.
§	&sect;	Section sign.
¢	&cent;	Cent sign.
¥	&yen;	Yen sign. (Currency.)
£	&pound;	Pound sign. (Currency.)
©	&copy;	Copyright sign.
®	&reg;	Registered trade mark sign.
™	&trade;	Trademark sign.

2.4.8 Fonts and Font Sizes

In fact, you can use CSS to control all font usage on the page. I also described how the font-family property can be used to specify that text should be rendered in a font belonging to a particular general category, such as monospace or serif. You can also use the font-family property to specify a specific font. More detail of CSS will be expressed in next chapter.

You can provide a single font or a list of fonts, and the browser will search for each of the fonts until it finds one on your system that appears in the list. You can also include a generic font family in the list of fonts if you like. Here are some examples:

You can provide a single font or a list of fonts, and the browser will search for each of the fonts until it finds one on your system that appears in the list. You can also include a generic font family in the list of fonts if you like. Here are some examples:

```
<p style="font-family: Verdana, Trebuchet, Arial, sans-serif;">
This is sans-serif text.</p>
<p style="font-family: 'Courier New', monospace;">This is monospace
text.</p>
<p style="font-family: Georgia;">This text will appear in the
Georgia font, or, if that font is not installed, the browser's
default font.</p>
```

Tags, tags, and more tags! In this section, you learned about most of the remaining tags in the HTML language for presenting text, and quite a few of the tags for additional text formatting and presentation. You also put together a real-life HTML home page.

Table 2-8 presents a quick summary of all the tags and attributes you've learned about in this section.

Table 2-8 HTML Tags for Formatting text

Tag	Attribute	Use
<address>...</address>		A signature for each web page; typically occurs near the bottom of each document and contains contact or copyright information.
...		Bold text
<blockquote>... **</blockquote>**		A quotation longer than a few words
	cite	The URL that was the source for the quotation
<cite>...</cite>		A citation
<code>...</code>		A code sample
<dfn>...</dfn>		A definition, or a term about to be defined
...		Emphasized text
...		Italic text.
<pre>...</pre>		Preformatted text; all spaces, tabs, and returns are retained. Text is printed in a monospaced font
...		Text in a smaller font than the text around it.

...		Strongly emphasized text
_{...}		Subscript text
^{...}		Superscript text
<u>...</u>		Underlined text
<var>...</var>		A variable name
<hr>		A horizontal rule line at the given position in the text. There's no closing tag in HTML for <hr>; for XHTML, add a space and forward slash (/) at the end of the tag and its attributes (for example, <hr size="2" width="75%" />).
	size	The thickness of the rule, in pixels. (Obsolete in HTML5.)
	width	The width of the rule, either in exact pixels or as a percentage of page width (for example, 50%). (Obsolete in HTML5.)
	align	The alignment of the rule on the page. Possible values are left, right, and center. (Obsolete in HTML5.)
	noshade	Displays the rule without three-dimensional shading. (Obsolete in HTML5.)
** **		A line break; starts the next character on the next line but doesn't create a new paragraph or list item. There's no closing tag in HTML for ; for XHTML, add a space and forward slash (/) at the end of the tag and its attributes (for example, <br clear="left" />).

2.5 Create a Real Web Page

Here's your chance to apply what you've learned and create a real web page. No more disjointed or overly silly examples. The web page you'll create in this section is a real one, suitable for use in the real world (or the real world of the web, at least).

The task for this example is to design and create a home page for a bookstore called The Bookworm,

which specializes in old and rare books.

2.5.1 Planning the Page

First, consider the content you want to include on this page. The following are some ideas for topics for this page:

- ✓ The address and phone number of the bookstore
- ✓ A short description of the bookstore and why it's unique
- ✓ Recent titles and authors
- ✓ Upcoming events

Now come up with some ideas for the content you're going to link to from this page. Each title in a list of recently acquired books seems like a logical candidate. You also can create links to more information about each book, its author and publisher, its price, and maybe even its availability.

The Upcoming Events section might suggest a potential series of links, depending on how much you want to say about each event. If you have only a sentence or two about each one, describing them on this page might make more sense than linking them to another page. Why make your readers wait for each new page to load for just a couple of lines of text?

Other interesting links might arise in the text itself, but for now, starting with the basic link plan is enough.

2.5.2 Beginning with a Framework

Next, create the framework that all HTML files must include: the document structure, a title, and some initial headings. Note that the title is descriptive but short; you can save the longer title for the <h1> element in the body of the text. The four <h2> subheadings help you define the four main sections you'll have on your web page:

```
<!DOCTYPE htm l>
<html>
<head>
<meta charset="UTF-8">
<title>The Bookworm Bookshop</title>
</head>
<body>
<h1>The Bookworm: A Better Book Store</h1>
<h2>Contents</h2>
<h2>About the Bookworm Bookshop</h2>
<h2>Recent Titles (as of July 11, 2018)</h2>
<h2>Upcoming Events</h2>
</body>
```

```
</html>
```

Each heading you've placed on your page marks the beginning of a particular section. You'll add IDs to each of the topic headings so that you can jump from section to section with ease. The IDs are simple: top for the main heading; contents for the table of contents; and about, recent, and upcoming for the three subsections on the page. With the IDs in place, the revised code looks like the following:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>The Bookworm Bookshop</title>
</head>
<body>
<h1 id="top">The Bookworm: A Better Book Store</h1>
<h2 id="contents">Contents</h2>
<h2 id="about">About the Bookworm Bookshop</h2>
<h2 id="recent">Recent Titles (as of July 11, 2018)</h2>
<h2 id="upcoming">Upcoming Events</h2>
</body>
</html>
```

2.5.3 Adding Content

Now begin adding the content. You're undertaking a literary endeavor, so starting the page with a nice quote about old books would be a nice touch. Because you're adding a quote, you can use the <blockquote> tag to make it stand out as such. Also, the name of the poem is a citation, so use <cite> there, too.

Insert the following code on the line after the level 1 heading:

```
<blockquote>
"Old books are best---how tale and rhyme<br>
Float with us down the stream of time!"<br>
-- Clarence Urmy, <cite>Old Songs are Best</cite>
</blockquote>
```

Immediately following the quote, add the address for the bookstore. Since it contains contact information, it's appropriate to use the <address> tag, as follows:

```
<address style="font-style: normal;">The Bookworm Bookshop<br />
```

```
1345 Applewood Dr<br />
Springfield, CA 94325<br />
(415) 555-0034
</address >
```

2.5.4 Adding the Table of Contents

The page you're creating will require a lot of scrolling to get from the top to the bottom. One nice enhancement is to add a small table of contents at the beginning of the page, listing the sections in a bulleted list. If a reader clicks one of the links in the table of contents, he'll automatically jump to the section that's of most interest to him. Because you've added the IDs already, it's easy to see where the links will take you.

You already have the heading for the table of contents. You just need to add the bulleted list and then create the links to the other sections on the page. The code looks like the following:

```
<h2 id="contents">Contents</h2>
<ul>
<li><a href="#about">About the Bookworm Bookshop</a></li>
<li><a href="#recent">Recent Titles</a></li>
<li><a href="#upcoming">Upcoming Events</a></li>
</ul>
```

Figure 2-27shows an example of the introductory portion of the Bookworm Bookshop page as it appears in a browser.

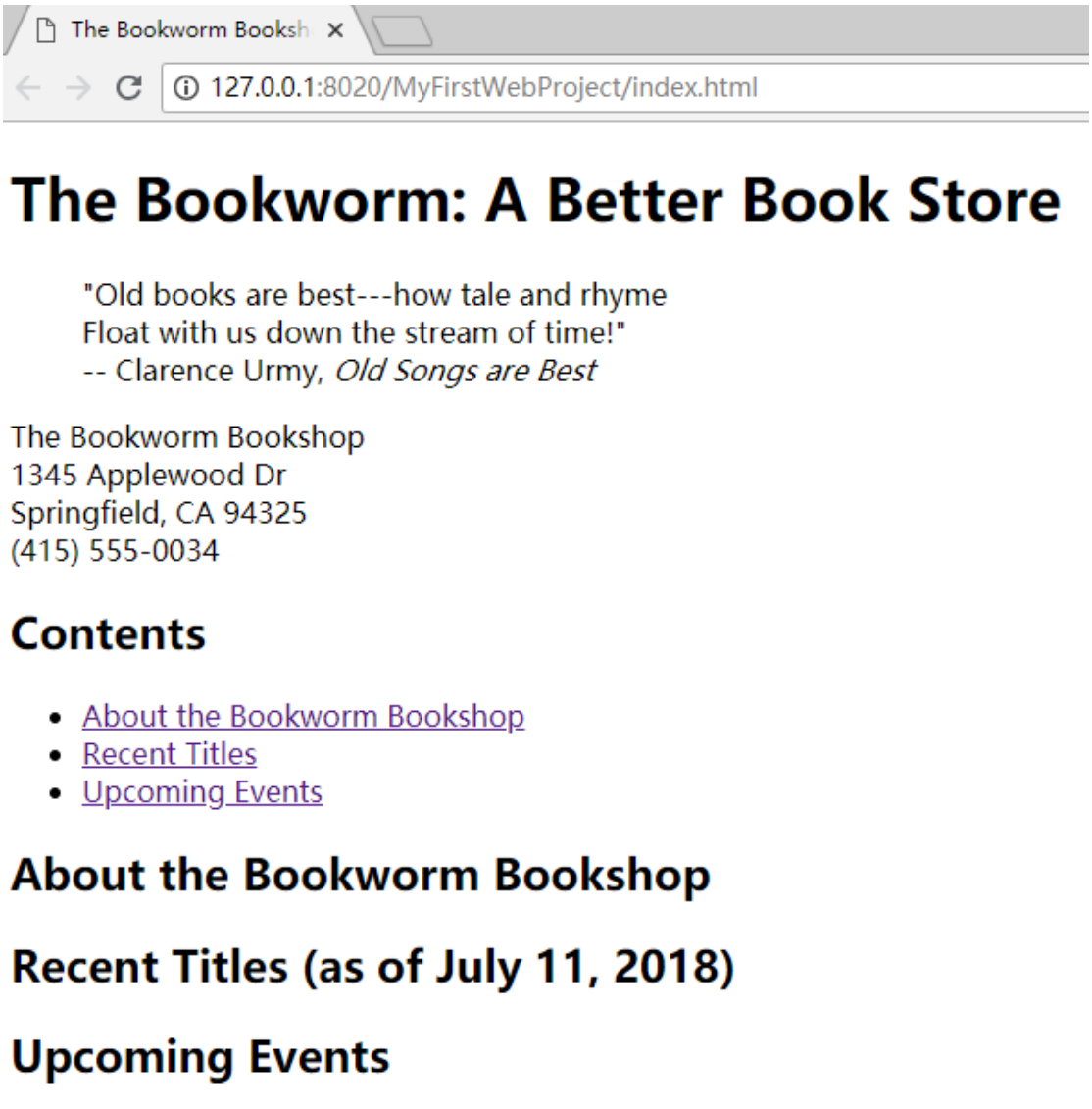


Figure 2-27 first section of real page

2.5.5 Creating the Description of the Bookstore

Now you come to the first descriptive subheading on the page, which you've added already. This section gives a description of the bookstore. After the heading (shown in the first line of the following example), I've arranged the description to include a list of features to make them stand out from the text better:

```
<h2 id= "about">About the Bookworm Bookshop</h2>
<p>Since 1933, The Bookworm Bookshop has offered
rare and hard-to-find titles for the discerning reader.
The Bookworm offers:</p>
<ul>
<li>Friendly, knowledgeable, and courteous help</li>
```

```
<li>Free coffee and juice for our customers</li>
<li>A well-lit reading room so you can "try before you buy"</li>
<li>Four friendly cats: Esmerelda, Catherine, Dulcinea and
Beatrice</li>
</ul>
```

Add a note about the hours the store is open and emphasi ze the actual numbers:

```
<p>Our hours are <strong>10am to 9pm</strong> weekdays,
<strong>noon to 7</strong> on weekends.</p>
```

Then, end the section with links to the table of contents and the top of the page, using the implicit top anchor:

```
<p><a href="#contents">Back to Contents</a> | <a href="#top">Back
to Top</a></p>
```

Figure 2-28 shows you what the About the Bookworm Bookshop section looks like in a browser .

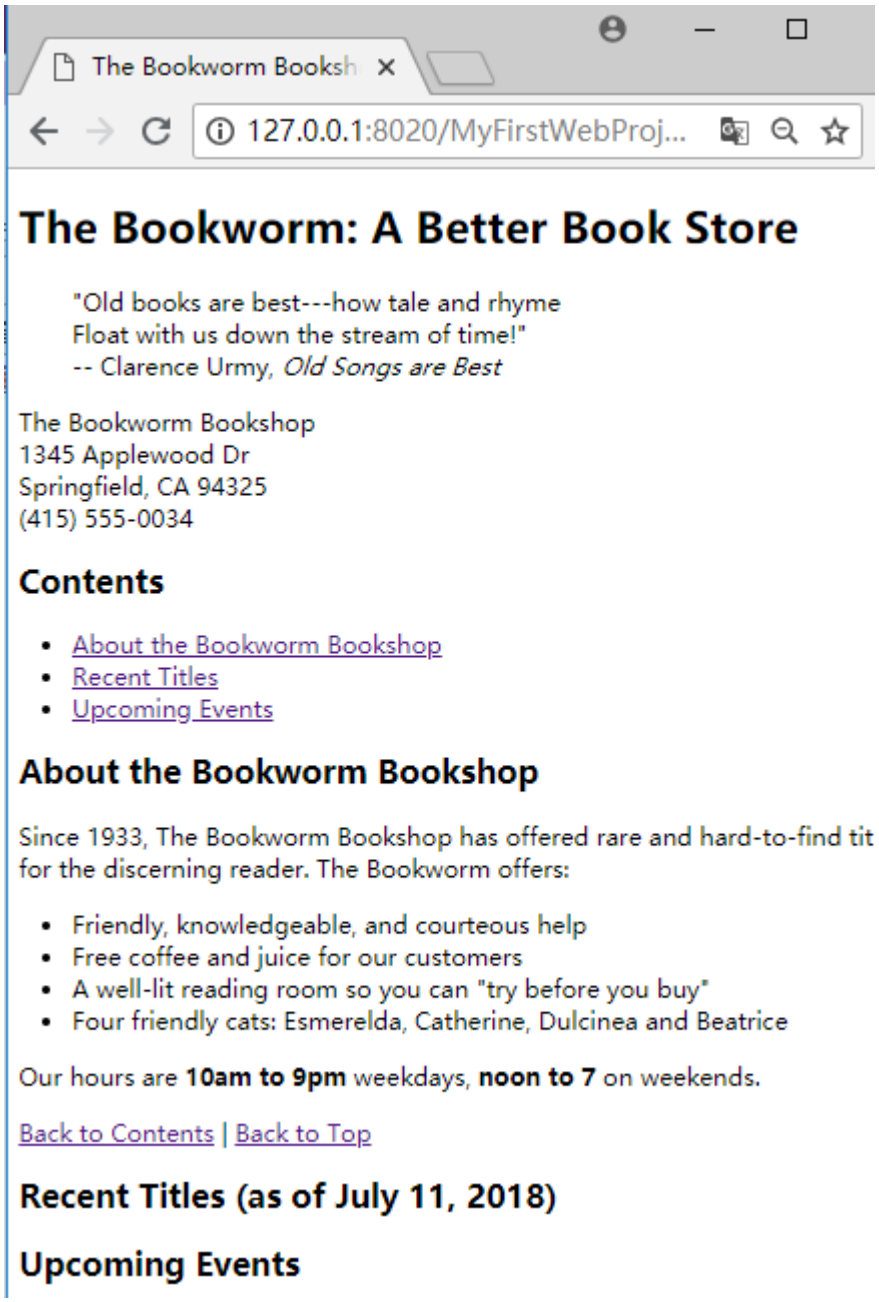


Figure 2-28 about section

2.5.6 Creating the Recent Titles Section

The Recent Titles section itself is a classic link menu, as I described earlier in this section. Here you can put the list of titles in an unordered list, with the titles themselves as citations, by using the <cite> tag. End the section with another horizontal rule.

After the Recent Titles heading (shown in the first line in t he following example), enter the following code:

```
<h2 id="recent">Recent Titles (as of July 11, 2012)</h2>
<ul>
<li>Sandra Bellweather, <cite>Belladonna</cite></li>
<li>Jonathan Tin, <cite>20-Minute Meals for One</cite></li>
<li>Maxwell Burgess, <cite>Legion of Thunder</cite></li>
<li>Alison Caine, <cite>Banquo's Ghost</cite></li>
</ul>
```

Now add the anchor tags to create the links. How far should the link extend? Should it include the whole line (author and title) or just the title of the book? This decision is a matter of preference, but remember that people viewing your page on mobile devices need longer links to be able to tap them with their fingers. Here, I linked only the titles of the books. At the same time, I also added links to the table of con tents and the top of the page

```
<h2 id="recent">Recent Titles (as of July 11, 2018)</h2>
<ul>
<li>Sandra Bellweather, <a
href="belladonna.html"><cite>Belladonna</cite></a></li>
<li>Johnathan Tin, <a href="20minmeals.html"><cite>20-Minute Meals
for One</cite></a></li>
<li>Maxwell Burgess, <a href="legion.html"><cite>Legion of
Thunder</cite></a></li>
<li>Alison Caine, <a href="banquo.html"><cite>Banquo's
Ghost</cite></a></li>
</ul>
<p><a href="#contents">Back to Contents</a> | <a href="#top">Back
to Top</a></p>
```

Note that I put the <cite> tag inside the link tag <a>. I could have just as easily put it outside the anchor tag; character style tags can go just about anywhere. But as I mentioned once before, be careful not to overlap tags. Your browser might not be able to understand what's going on, and it's invalid. In other words, don't do the following:

<cite>Banquo's Ghost</cite>

Take a look at how the Recent Titles section appears in Figure 2-29.

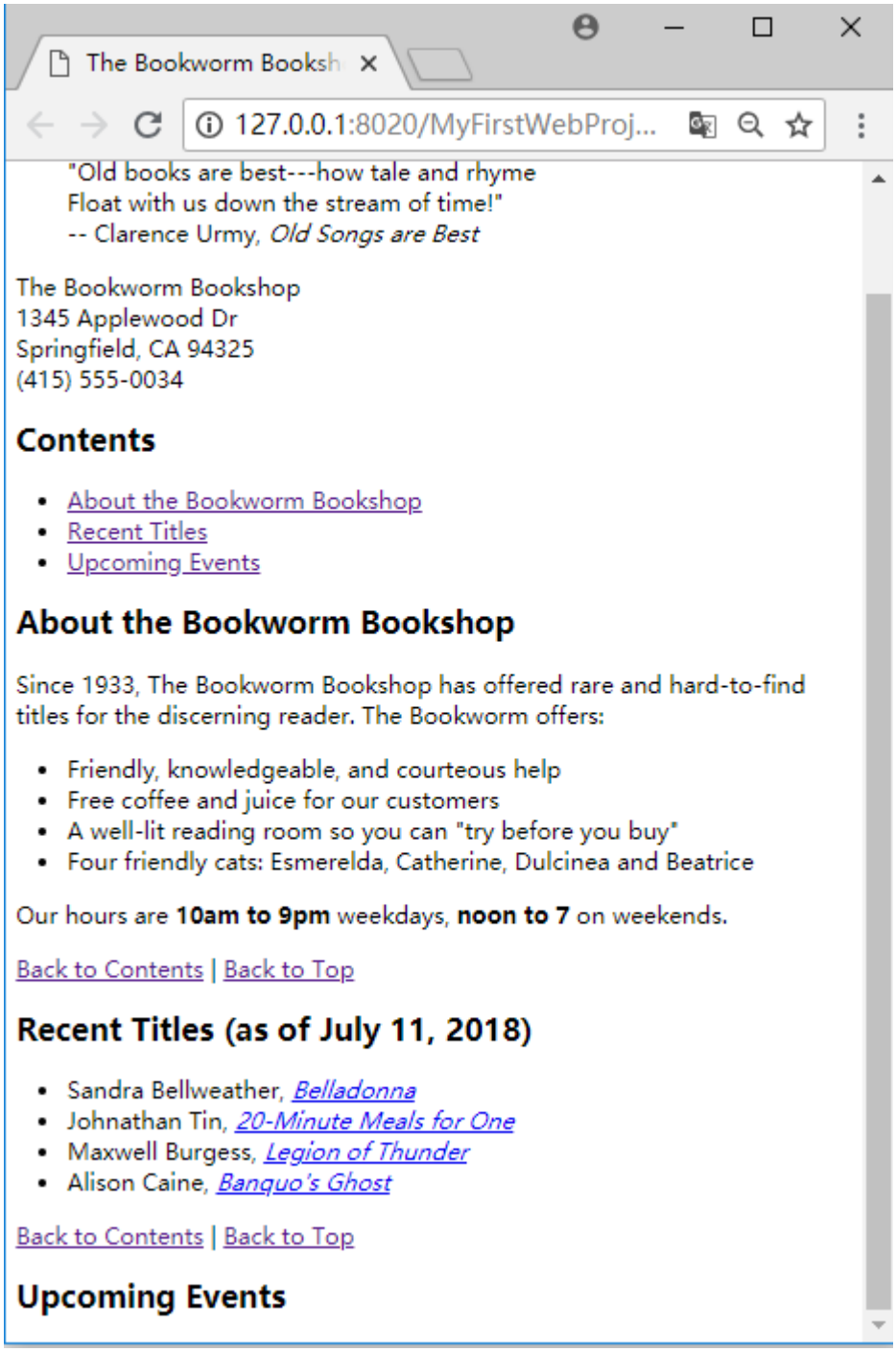


Figure 2-29 recent title section

2.5.7 Completing the Upcoming Events Section

Next , move on to the Upcoming Events section. In the planning stages, you weren't sure whether this would be another link menu or whether the content would work better solely on this page. Again, this decision is a matter of preference. Here, because the amount of extra information is minimal,

creating links for just a couple of sentences doesn't make much sense. So, for this section, create an unordered list using the <ul> tag. I've boldfaced a few phrases near the beginning of each paragraph. These phrases emphasize a summary of the event itself so that the text can be scanned quickly and ignored if the readers aren't interested.

As in the previous sections, you end the section with links to the top and to the table of contents:

```
<h2 id="upcoming">Upcoming Events</h2>
<ul>
<li><strong>The Wednesday Evening Book Review</strong> meets,
appropriately, on Wednesday evenings at 7 pm for coffee and a round-
table discussion. Call the Bookworm for information on joining the
group.</li>
<li><strong>The Children's Hour</strong> happens every Saturday at
1 pm andincludes reading, games, and other activities. Cookies and
milk are served.</li>
<li><strong>Carole Fenney</strong> will be at the Bookworm on
Sunday, January 19, to read from her book of poems <cite>Spiders
in the Web.</cite></li>
<li><strong>The Bookworm will be closed</strong> March 1st to
remove a family of bats that has nested in the tower. We like the
company, but not the mess they leave behind! </li>
</ul>
<p><a href="#contents">Back to Contents</a> | <a href="#top">Back
toTop</a></p>
```

2.5.8 Signing the Page

To finish, sign what you have so that your readers know who did the work. Here, I've separated the signature from the text with a rule line. I've also included the most recent revision date, my name as the webmaster, and a basic copyright.

```
<address>
Last Updated: July 11, 2012<br>
Webmaster: Laura Lemay
<a href="mailto:lemay@bookworm.com">lemay@bookworm.com</a><br>
&copy; copyright 2012 the Bookworm<br>
</address>
```

Figure 2-30shows the signature at the bottom portion of the page as well as the Upcoming Events section.

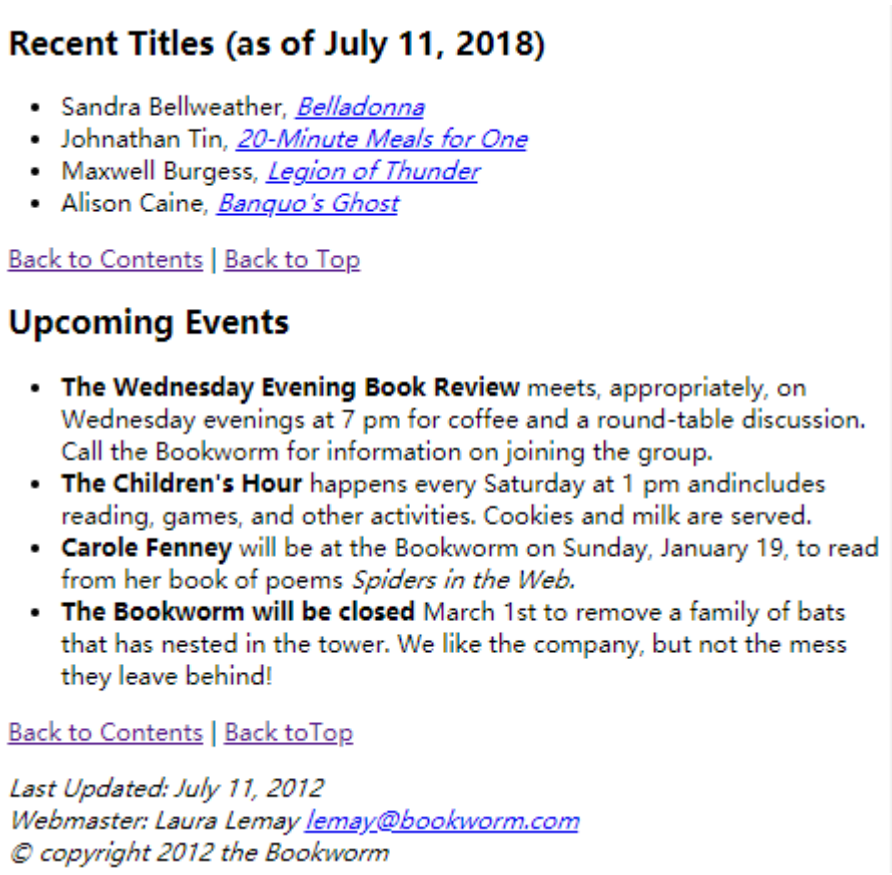


Figure 2-30 Upcoming section

Here's the HTML code for the page

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>The Bookworm Bookshop</title>
</head>
<body>
<h1>The Bookworm: A Better Book Store</h1>
<blockquote>
"Old books are best---how tale and rhyme<br>
Float with us down the stream of time!"<br>
-- Clarence Urmy, <cite>Old Songs are Best</cite>
</blockquote>
<address style="font-style: normal">The Bookworm Bookshop<br>
1345 Applewood Dr<br>
Springfield, CA 94325<br>
(415) 555-0034
</address>
```

```
<h2 id="contents">Contents</h2>
<ul>
<li><a href="#about">About the Bookworm Bookshop</a></li>
<li><a href ="#recent">Recent Titles</a></li>
<li><a href ="#upcoming">Upcoming Events</a></li>
</ul>
<h2 id="about">About the Bookworm Bookshop</h2>
<p>Since 1933, The Bookworm Bookshop has offered rare and hard-to-
find titles for the discerning reader. The Bookworm offers:</p>
<ul>
<li>Friendly, knowledgeable, and courteous help</li>
<li>Free coffee and juice for our customers</li>
<li>A well-lit reading room so you can "try before you buy"</li>
<li>Four friendly cats: Esmerelda, Catherine, Dulcinea and
Beatrice</li>
</ul>
<p>Our hours are <strong>10am to 9pm</strong> weekdays,
<strong>noon to 7</strong> on weekends.</p>
<p><a href=" #contents">Back to Contents</a> | <a href="#top">Back
to Top
</a></p>
<h2 id="recent">Recent Titles (as of July 11, 2018)</h2>
<ul>
<li>Sandra Bellweather, <a href="belladonna.html">
<cite>Belladonna</cite></a></li>
<li>Johnathan Tin, <a href="20minmeals.html"> <cite>20-Minute Meals
for One</cite></a></li>
<li>Maxwell Burgess, <a href="legion.html"> <cite>Legion of
Thunder</cite></a></li>
<li>Alison Caine, <a href="banquo.html"> <cite>Banquo's
Ghost</cite></a></li>
</ul>
<p><a href=" #contents">Back to Contents</a> | <a href="#top">Back
to Top</
a></p>
<h2 id="upcoming">Upcoming Events</h2>
<ul>
<li><strong>The Wednesday Evening Book Review</strong> meets,
appropriately, on Wednesday evenings at 7 pm for coffee and a round-
table. discussion Call the Bookworm for information on joining the
group.</li>
<li><strong>The Children's Hour</strong> happens every Saturday at
1 pm and includes reading, games, and other activities. Cookies and
milk are served.</li>
```

```
<li><strong>Carole Fenney</strong> will be at the Bookworm on
Sunday, January 19, to read from her book of poems <cite>Spiders
in the Web. </cite></li>
<li><strong>The Bookworm will be closed</strong> March 1st to
remove a family of bats that has nested in the tower. We like the
company, but not the mess they leave behind!</li>
</ul>
<p><a href="#contents">Back to Contents</a> | <a href="#top">Back
to
Top</a></p>
<address>
Last Updated: July 11, 2012<br>
Webmaster: Laura Lemay
<a href="mailto:lemay@bookworm.com">lemay@bookworm.com</a><br>
&copy; copyright 2012 the Bookworm<br>
</address>
</body>
</html>
```

Now you have some headings, some text, some topics, and some links, which form the basis for an excellent web page. With most of the content in place, now you need to consider what other links you might want to create or what other features you might want to add to this page.

For example, the introductory section has a note about the four cats owned by the bookstore. Although you didn't plan for them in the original organization, you could easily create web pages describing each cat (and showing pictures) and then link them back to this page, one link (and one page) per cat.

Is describing the cats important? As the designer of the page, that's up to you to decide. You could link all kinds of things from this page if you have interesting reasons to link them (and something to link to). Link the bookstore's address to an online mapping service so that people can get driving directions. Link the quote to an online encyclopedia of quotes. Link the note about free coffee to the Coffee home page.

My reason for bringing up this point here is that after you have some content in place on your web pages, there might be opportunities for extending the pages and linking to other places that you didn't think of when you created your original plan. So, when you're just about finished with a page, stop and review what you have, both in the plan and on your web page.

For the purposes of this example, stop here and stick with the links you have. You're close enough to being done, and I don't want to make this lesson any longer than it already is!

Now that all the code is in place, you can preview the results in a browser. Figure 2-27 through Figure 2-30 show how it looks in a browser. Actually, these figures show what the page looks like after you fix the spelling errors, the forgotten closing tags, and all the other strange bugs that always seem to creep into an HTML file the first time you create it. These problems always seem to happen

no matter how good you are at creating web pages. If you use an HTML editor or some other help tool, your job will be easier, but you'll always seem to find mistakes. That's what previewing is for—so you can catch the problems before you actually make the document available to other people. Plus, the more browsers that you view your pages in, the fewer problems your customers will see.

2.6 Quiz

1. What three HTML tags are used to describe the overall structure of a web page, and what do each of them define?

2. Where does the <title> tag go, and what is it used for?

3. How many different levels of headings does HTML support? What are their tags?

4. Why is it a good idea to use two-sided paragraph tags, even though the closing tag </p> is optional in HTML?

5. Ordered and unordered lists use the <li> tag for list items. What tags are used by definition lists?

6. Is it possible to nest an ordered list within an unordered list?

7. Which attribute is used to set the starting number for an ordered list? What about to change the value of an element within a list?

8. What are the three types of bullets that can be specified for unordered lists using the type attribute?

9. What two things do you need to create a link in HTML?

10. What's a relative pathname? Why is it advantageous to use one?

11. What's an absolute pathname?

12. What's an anchor, and what is it used for?

13. Besides HTTP (web page) URLs, what other kinds are there?

14. List eight semantic tags and what they're used for

15. What's the most common use of the <address> tag?

16. What are some things that the <pre> (preformatted text) tag can be used for?

2.7 Practice

1. Create a page that describes the following head information

- ✓ Title information: The first HTML page
- ✓ Key words information: CSS, HTML, JavaScript
- ✓ Description information: What HTML is?

2. Create a page, and then set the body's attribute.

- ✓ Specifies the color for text of the document to white;
- ✓ Specifies the color for background of the document to #008080
- ✓ Specifies the color for text of the **link** to #0000ff
- ✓ Specifies the color for text of the **alink** to #808000
- ✓ Specifies the color for text of the **vlink** to #ff0000

3. Use nested lists to create an outline of the topics covered in Figure 2-31 a nest lists for exercise

Figure 2-31 a nest lists for exercise

4. Use nested lists and the list-style-type CSS property to create a traditional outline of the topics you plan to cover on your own website.

5. Creating a link menu

Link menus are links on your web page that are arranged in list form or in some other short, easy-to-read, and easy-to-understand format. Start with a simple page framework: a first-level heading and some basic text:

```
<!DOCTYPE html>
<html>
```

```
<head>
<title>Really Honest Book Reviews</title>
</head>
<body>
<h1>Really Honest Book Reviews</h1>
<ul>
<li><em>The Rainbow Returns</em> by E. Smith</li>
<li><em>Seven Steps to Immeasurable Wealth</em> by R. U. Needy</li>
<li><em>The Food-Lovers Guide to Weight Loss</em> by L. Goode</li>
<li><em>The Silly Person's Guide to Seriousness</em> by M. Nott</li>
</ul>
</body>
</html>
```

Now, modify each of the list items so that they include links. You need to keep the <li> tag in there because it indicates where the list items begin. Just add the <a> tags around the text itself. Here you'll link to filenames on the local disk in the same directory as this file, with each file containing the review for the particular book:

```
<ul>
<li><a href="rainbow.html"><em>The Rainbow Returns</em> by E.
Smith</a></li>
<li><a href="wealth.html"><em>Seven Steps to Immeasurable
Wealth</em> by R. U.
Needy</a></li>
<li><a href="food.html"><em>The Food-Lovers Guide to Weight
Loss</em> by L.
Goode</a></li>
<li><a href="silly.html"><em>The Silly Person's Guide to
Seriousness</em> by M.
Nott</a></li>
</ul>
```

Now, create the four files and add the link to the file, and the final result is shown as Figure 2-32.

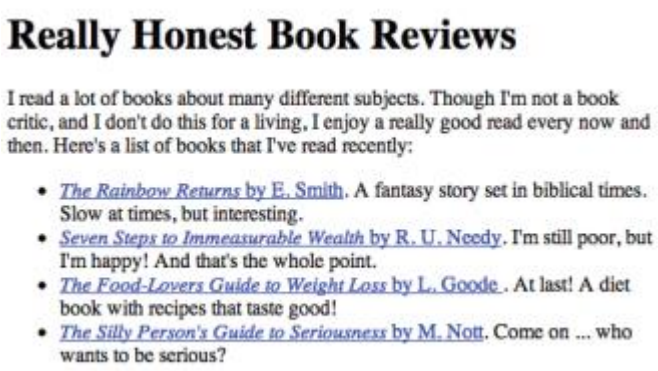


Figure 2-32 the final list menu

6. Use anchor to create an outline of the topics covered in this section.

7. Create the real page in section 2.5

8. Create your homepage likes section 2.5.

3 Doing More with HTML and CSS

HTML does a good job of setting up the basic design of a page, but face it: The pages it makes are pretty ugly. In the old days, developers added a lot of other tags to HTML to make it prettier, but changing the design with HTML code was difficult. Now, HTML disallows all the tags that made pages more attractive. That sounds bad, but it isn't really a loss. Today, HTML is almost always written in concert with CSS (Cascading Style Sheets). It's amazing how much you can do with CSS to beautify your HTML pages.

CSS allows you to change the color of any font on the page, add backgrounds and borders, change the visual appearance of elements (like lists and links), as well as customize the entire layout of your page. Additionally, CSS allows you to keep your HTML simple because all the formatting is stored in the CSS. CSS is efficient, too, because it allows you to reuse a style across multiple elements and pages. If HTML gives your pages structure, CSS gives them beauty.

3.1 How to add CSS to HTML

How to add CSS to HTML?

- ✓ Inline style
- ✓ Embedded style tag
- ✓ Link stylesheet file

3.1.1 What is CSS

CSS stands for Cascading Style Sheets, CSS describes how HTML elements are to be displayed on screen, paper, or in other media.

Adding CSS to HTML can be confusing because there are many ways to do it. CSS can be added to HTML by linking to a separate stylesheet file, importing files from existing stylesheets, embedded CSS in a style tag, or adding inline styles directly to HTML elements.

3.1.2 Inline Styles

Style rules can be added directly to any HTML element. To do this, simply add a style attribute to the element then enter your rules as a single line of text (string of characters) for the value.

Thus far, when I've discussed style sheets, I've applied them using the style attribute. For example,

I've shown how you can modify the font for some text using tags such as or how you can modify the appearance of a list item by applying a style within an <li> tag. If you rely on the style attribute of tags to apply CSS, if you want to embolden every paragraph on a page, you need to put style="font-weight: bold" in every tag. This is no improvement over just using and instead. The good news is that the style attribute is the least efficient method of applying styles to a page or a site.

Here's an example of a heading with inline styles:

Example 3-1 inline style

```
<h2 style="color:red; background:black;">This is a red heading with a black background</h2>
```

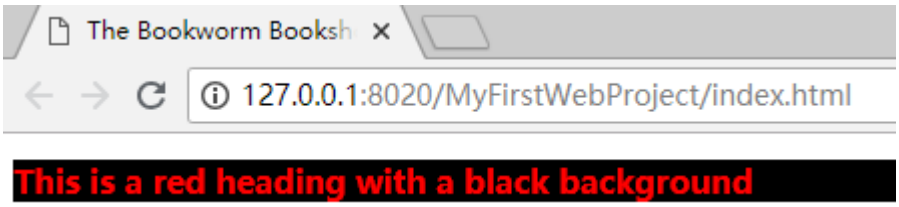


Figure 3-1 result of inline style

Advantages of inline styles

- ✓ Inline styles override external styles. This is because inline styles have a higher specificity than external CSS.
- ✓ Faster pages: Just like embedded CSS, no extra HTTP requests are required.
- ✓ You can change styles without editing the main CSS file: Sometimes you might need to change a style rule but you don't have access to the main website stylesheet, with this method you can add rules directly to each element instead.
- ✓ Great for dynamic styles: For example: you can add a background-image URL as an inline style if it's different for each element.
- ✓ Useful for HTML emails: EDMs can be tricky to get right, often the best way is to use inline styles everywhere.

Disadvantages

- ✓ Excess styles can bloat your page: If you're specifying the same styles over and over your page weight will grow.
- ✓ Maintenance can be tricky: Site-wide style changes will need to be made on every page, this can be tedious.

3.1.3 Embedded Style Tag

You can embedded CSS rules directly into HTML documents by using a style tag. It is a page-level style.

First, let's look at how we can apply styles to our page at the page level. Thus far, you've seen how styles are applied, but you haven't seen any style sheets. Here's what one looks like:

Example 3-2 Style tag in head

```
<style type="text/css">
h1 { font-size: x-large; font-weight: bold; }
h2 { font-size: large; font-weight: bold; }
</style>
```

The <style> tag should be included within the <head> tag on your page. The type attribute indicates the type of the style sheet. text/css is the only value you'll use. It's not required in HTML5, and most designers leave it out. The body of the style sheet consists of a series of rules. All rules have the same structure:

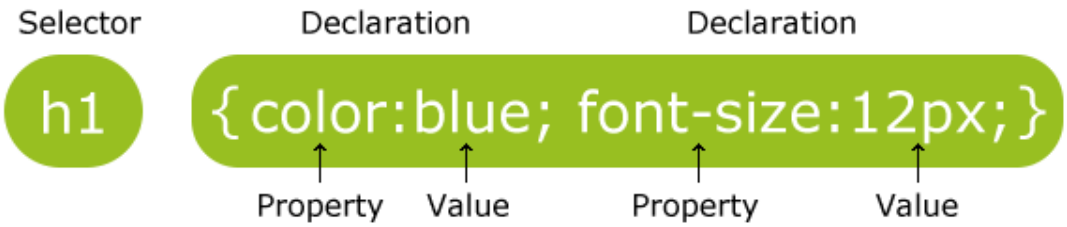


Figure 3-2 CSS rule sets

Each rule consists of a selector followed by a list of properties and values associated with those properties. All the properties being set for a selector are enclosed in curly braces, as shown in the example. You can include any number of properties for each selector, and they must be separated from one another using semicolons. You can also include a semicolon following the last property/value pair in the rule, or not, but best practices recommend that you do.

You should already be familiar with CSS properties and values because that's what you use in the style attribute of tags. Selectors are something new. I discuss them in detail in a bit. The ones I've used thus far have the same names as tags. If you use h1 as a selector, the rule will apply to any <h1> tags on the page. By the same token, if you use p as your selector, it will apply to <p> tags.

The secret to CSS is the style sheet, a set of rules for describing how various objects will display. For example, look at Example 3-3.

Example 3-3 embedded style tag

```
<!DOCTYPE html>
<html lang = "en-US">
<head>
```

```
<meta charset = "UTF-8">
<title>basicColors.html</title>
<style type = "text/css">
body {
color: yellow;
background-color: red;
}
h1 {
color: red;
background-color: yellow;
}
</style>
</head>
<body>
<h1>Red text on a yellow background</h1>
<p>
Yellow text on a red background
</p>
</body>
</html>
```

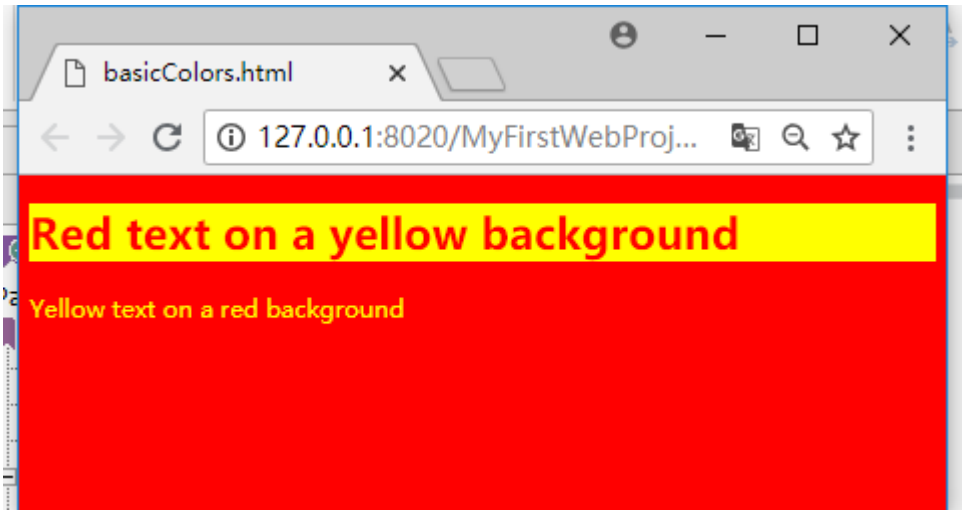


Figure 3-3 an embedded style tag

Each selector can have a number of style rules. Each rule describes some attribute of the selector. To set up a style, keep the following in mind:

- ✓ Begin with the style tags. The type of style you'll be working with for now is embedded into the page. You should describe your style in the header area.
- ✓ Include the style type in the header area. The style type is always "text/css". The beginning <style> tag always looks like this: <style type = "text/css">. The type attribute is not necessary

in HTML5, and you can leave it out.

- ✓ Define an element. Use the element name (the tag name alone) to begin the definition of a particular element's style. You can define styles for all the HTML elements (and other things, too, but not today). The selector for the body is designated like this:

```
body {
```

- ✓ Use braces ({}) to enclose the style rules. Each style's rules are enclosed in a set of braces. Similar to many programming languages, braces mark off special sections of code. It's traditional to indent inside the braces.

- ✓ Give a rule name. In this section, I'm working with two very simple rules: color and background-color. Throughout this chapter, you can read about many more CSS rules (sometimes called attributes) that you can modify. A colon (:) character always follows the rule name.

- ✓ Enter the rule's value. Different rules take different values. The attribute value is followed by a semicolon. Traditionally, each name-value pair is on one line, like this:

```
body {
color: yellow;
background-color: red;
}
```

Advantages of embedded style tags

- ✓ Faster loading times: Because the CSS is part of the HTML document, the whole page exists as just one file. No extra HTTP requests are required. I use this method on my liquid-layout demos so when people view the source of the page they can see the HTML and the CSS code together.

- ✓ Works great with dynamic styles: If you're using a database to generate page content you can also generate dynamic styles at the same time. Blogger does this by dynamically inserting the colors for headings and other elements into the CSS rules embedded in the page. This allows users to change these colors from an admin page without actually editing the CSS in their blog templates.

Disadvantages

- ✓ Embedded styles must be downloaded with every page request: These styles cannot be cached by the browser and re-used for another page. Because of this, it's recommended to embed a minimal amount of CSS as possible.

3.1.4 Link Stylesheet File

This is the most common method of attaching CSS rules to HTML documents. With this method, all your style rules are contained in a single text file that's saved with the .css extension. This file is saved on your server and you link to it directly from each HTML file.

You can store all of your style information in a file and include it in your Web pages using an HTML tag. A CSS file contains the body of a <style> tag. To turn the style sheet from the previous section into a separate file, you could just save the following to a file called styles.css:

```
h1 { font-size: x-large; font-weight: bold; }
h2 { font-size: large; font-weight: bold; }
```

In truth, the extension of the file is irrelevant, but the extension .css is the de facto standard for style sheets, so you should probably use it. After you've created the style sheet file, you can include it in your page using the <link> tag, like this:

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

- ✓ The rel attribute is set to stylesheet to tell the browser that the linked file is a Cascading Style Sheet (CSS).
- ✓ The type attribute is the same as that of the <style> tag and is not required in HTML5.
- ✓ The href attribute is the same as that of the <a> tag. It can be a relative URL, an absolute URL, or even a fully qualified URL that points to a different server. As long as the browser can fetch the file, any URL will work. This means that you can just as easily use other people's style sheets as your own.
    - ➢ If it's saved in a folder, then you can specify the folder path relative to the HTML file like this:

```
<link href="foldername/mystyles.css" ... >
```

    - ➢ You can also specify a path relative to the root of your domain by prefixing with a forward slash like this:

```
<link href="/foldername/mystyles.css" ... >
```

    - ➢ Absolute URLs also work:

```
<link
href="https://www.yourdomain.com/foldername/mystyles.css" ... >
```

- ✓ There's another attribute of the link tag, too: media. This enables you to specify different style sheets for different display mediums. For example, you can specify one for print, another for screen display, and others for things like aural browsers for use with screen readers. Not all browsers support the different media types, but if your style sheet is specific to a particular medium, you should include it. The options are screen, print, projection, aural, braille, tty, tv, embossed, handheld and all. **You can leave off the media attribute completely if you want your styles to be applied for all media types.**
    - ➢ screen indicates for use on a computer screen.

- projection for projected presentations.
- handheld for handheld devices (typically with small screens).
- print to style printed web pages.
- all (default value) This is what most people choose..

Advantages of linking to a separate CSS file

✓ Changes to your CSS are reflected across all pages: You only need to make a CSS change once in your single CSS file and all website pages will update.

✓ Changing your website theme is easy: You can replace your CSS file to completely change the look of your website.

✓ Site speed increases for multiple page requests: When a person first visits your website their browser downloads the HTML of the current page plus the linked CSS file. When they navigate to another page, their browser only needs to download the HTML of the new page, the CSS file is cached so doesn't need to be downloaded again. This can make a big difference particularly if you have a large CSS file.

Disadvantages

✓ An additional HTTP request is required for each linked CSS file: Excess HTTP requests can delay the rendering of your page. We'll cover the solution to this problem shortly.

An example is shown as following.

Example 3-4 An example of link style

CSS filename is new_file.css:

```
body
{
    font-family: "times new roman";
    font-size: 10px;
    font-style: italic;
    text-underline: initial;
}
```

HTML file

```
<!DOCTYPE html>
<html lang = "en-US">
<head>
<meta charset = "UTF-8">
<title>basicColors.html</title>
<link rel="stylesheet" type="text/css" href="css/new_file.css" />
```

```
<style type = "text/css">
</style>
</head>
<body>
<p>Font style of text is italic<br>
Font family of text is times new roman</p>
</body>
</html>
```

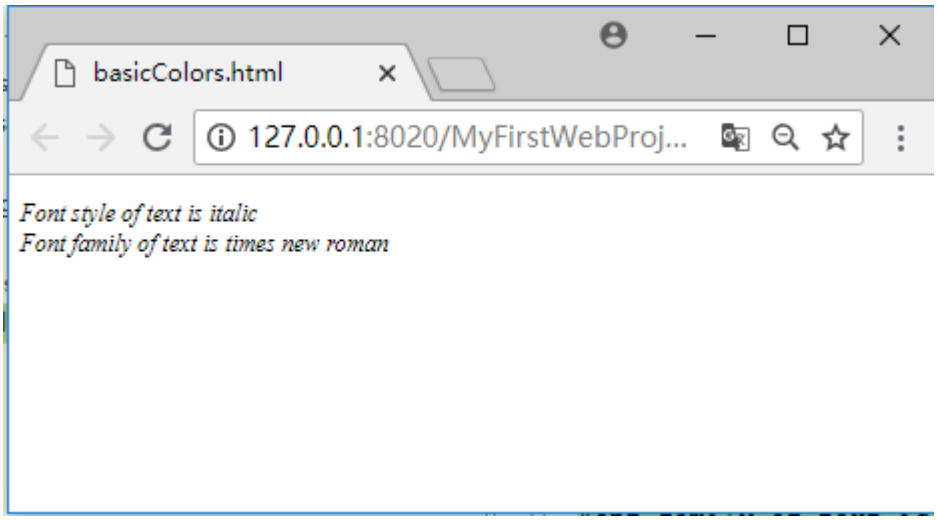


Figure 3-4 result of link style

As it is, you can include links to multiple style sheets in your pages, and all the rules will be applied. This means that you can create one general style sheet for your entire site, and then another specific to a page or to a section of the site, too.

As you can see, the capability to link to external style sheets provides you with a powerful means for managing the look and feel of your site. After you've set up a sitewide style sheet that defines the styles for your pages, changing things such as the headline font and background color for your pages all at once is trivial. Before CSS, making these kinds of changes required a lot of manual labor or a facility with tools that had search and replace functionality for multiple files. Now it requires quick edits to a single linked style sheet.

3.2 Coloring Your World

This section gets you started by describing how to add color to your pages. At first we look at a simple style example

```
body {
```

```
color: yellow;
background-color: red;
}
```

In this very simple example, I just changed some colors around. Here are the two primary color attributes in CSS:

- ✓ color: This refers to the foreground color of any text in the element.
- ✓ background-color: The background color of the element. (The hyphen is a formal part of the name. If you leave it out, the browser won't know what you're talking about.)

With these two elements, you can specify the color of any element. For example, if you want all your paragraphs to have white text on a blue background, add the following text to your style:

```
p {
color: white;
background-color: blue;
}
```

CSS is case-sensitive. CSS styles should be written entirely in lowercase. You'll figure out many more style elements in your travels, but they all follow the same principles illustrated by the color attributes.

Here are the two main ways to define colors in CSS. You can use color names, such as pink and fuchsia, or you can use hex values. Each approach has its advantages.

3.2.1 Using Color Names

Color names seem like the easiest solution, and, for basic colors like red and yellow, they work fine. However, here are some problems with color names that make them troublesome for web developers:

- ✓ **Only 16 color names will validate**. Although most browsers accept hundreds of color names, only 16 are guaranteed to validate in CSS and HTML validators. See Table 3-1 for a list of those 16 colors.

Table 3-1 Legal Color Names and Hex Equivalents

Color	Hex Value
Black	#000000
Silver	#C0C0C0
Gray	#808080
White	#FFFFFF

Maroon	#800000
Red	#FF0000
Purple	#800080
Fuchsia	#FF00FF
Green	#008800
Lime	#00FF00
Olive	#808000
Yellow	#FFFF00
Navy	#000080
Blue	#0000FF
Teal	#008080
Aqua	#00FFFF

- ✓ **Color names are somewhat subjective.** You'll find different opinions on what exactly constitutes any particular color, especially when you get to the more obscure colors. (I personally wasn't aware that PeachPuff and PapayaWhip are colors. They sound more like dessert recipes to me.)
- ✓ **It can be difficult to modify a color**. For example, what color is a tad bluer than Gainsboro? (Yeah, that's a color name, too. I had no idea how extensive my color disability really was.)
- ✓ **They're hard to match**. Say you're building an online shrine to your cat and you want the text to match your cat's eye color. It'll be hard to figure out exactly what color name corresponds to your cat's eyes. I guess you could ask the cat.

Hex color values can be indicated in uppercase or lowercase. The mysterious hex codes are included in this table for completeness. It's okay if you don't understand what they're about. All is revealed in the next section.

I added a simple page to display all the named colors. Check namedColors.html to see the actual colors.

Example 3-5 name color

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>namedColors.html</title>
```

```
</head>
<body>
  <h1>Named Colors</h1>
  <table border="1">
    <tr>
      <th>color name</th>
      <th>color value</th>
    </tr>
    <tr>
      <td>aqua</td>
      <td style="background-color: aqua;"><br /></td>
    </tr>
    <tr>
      <td>black</td>
      <td style="background-color: black;"><br /></td>
    </tr>
    <tr>
      <td>blue</td>
      <td style="background-color: blue;"><br /></td>
    </tr>
    <tr>
      <td>fuchsia</td>
      <td style="background-color: fuchsia;"><br /></td>
    </tr>
    <tr>
      <td>gray</td>
      <td style="background-color: gray;"><br /></td>
    </tr>
    <tr>
      <td>green</td>
      <td style="background-color: green;"><br /></td>
    </tr>
    <tr>
      <td>lime</td>
      <td style="background-color: lime;"><br /></td>
    </tr>
    <tr>
      <td>maroon</td>
      <td style="background-color: maroon;"><br /></td>
    </tr>
    <tr>
      <td>navy</td>
      <td style="background-color: navy;"><br /></td>
    </tr>
```

```
    <tr>
      <td>olive</td>
      <td style="background-color: olive;"><br /></td>
    </tr>
    <tr>
      <td>purple</td>
      <td style="background-color: purple;"><br /></td>
    </tr>
    <tr>
      <td>red</td>
      <td style="background-color: red;"><br /></td>
    </tr>
    <tr>
      <td>silver</td>
      <td style="background-color: silver;"><br /></td>
    </tr>
    <tr>
      <td>teal</td>
      <td style="background-color: teal;"><br /></td>
    </tr>
    <tr>
      <td>white</td>
      <td style="background-color: white;"><br /></td>
    </tr>
    <tr>
      <td>yellow</td>
      <td style="background-color: yellow;"><br /></td>
    </tr>
  </table>
 </body>
</html>
```

The result of name color is shown as Figure 3-5

Figure 3-5 result of name color

3.2.2 Putting a Hex on Your Colors

Colors in HTML are a strange thing. The “easy” way (with color names) turns out to have many problems. The method most web developers really use sounds a lot harder, but it isn’t as bad as it may seem at first. The hex color scheme uses a seemingly bizarre combination of numbers and letters to determine color values. #00FFFF is green. #FFFF00 is yellow. It’s a scheme only a computer scientist could love. Yet, after you get used to it.

Hex colors work by describing exactly what the computer is doing, so you have to know a little more about how computers work with color. Each dot (or pixel) on the screen is actually composed of three tiny beams of light (or LCD diodes or something similar). Each pixel has tiny red, green,

and blue beams.

The light beams work kind of like stage lights. Imagine a black stage with three spotlights (red, green, and blue) trained on the same spot. If all the lights are off, the stage is completely dark. If you turn on only the red light, you see red. You can turn on combinations to get new colors. For example, turning on red and green creates a spot of yellow light. Turning on all three lights makes white.

3.2.3 Coloring by Number

In a computer system, each of the little lights can be adjusted to various levels of brightness. These values measure from 0 (all the way off) to 255 (all the way on). Therefore, you could describe red as rgb(255, 0, 0) and yellow as rgb(255, 255, 0).

The 0 to 255 range of values seems strange because you're probably used to base 10 mathematics. The computer actually stores values in binary notation. The way a computer sees it, yellow is actually 11111111111111100000000. There has to be an easier way to handle all those binary values. That's why we use hexadecimal notation.

Figure 3-6 shows a page which allows you to pick colors with red, green, and blue sliders. Each slider shows its value in base 10 as well as in hexadecimal.

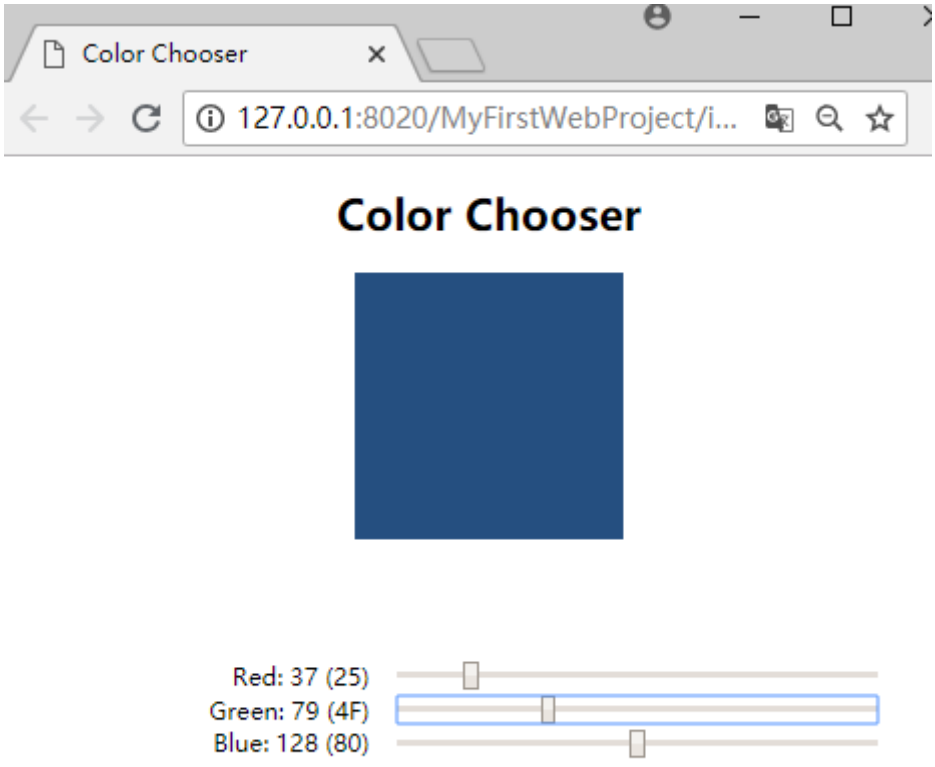


Figure 3-6 Pick color

The colorChooser program shown in Figure 3-6 uses technology that will be described in chapter 4. Any page that interacts with the user will tend to use a programming language (in this case, JavaScript). **Don't worry if you're not yet ready to understand the code, you'll get there soon in chapter 4.**

Example 3-6 pick your color

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Color Chooser</title>
  <style type = "text/css">
    h1 {
      text-align: center;
    }

    #swatch {
      width: 200px;
      height: 200px;
      margin-left: auto;
      margin-right: auto;
      background-color: rgb(128, 128, 128);
      margin-bottom: 5em;
    }

    output {
      float: left;
      width: 10em;
      clear: left;
      margin-left: 5em;
      text-align: right;
      padding-right: 1em;
    }

    input {
      float: left;
      width: 20em;
    }
  </style>
  <script type = "text/javascript">
  function changeColor(){
    //set up form elements
    var swatch = document.getElementById("swatch");
    var sldRed = document.getElementById("sldRed");
```

```
    var outRed = document.getElementById("outRed");
    var sldGreen = document.getElementById("sldGreen");
    var outGreen = document.getElementById("outGreen");
    var sldBlue = document.getElementById("sldBlue");
    var outBlue = document.getElementById("outBlue");

    //get values
    var red = sldRed.value;
    var green = sldGreen.value;
    var blue = sldBlue.value;

    //change color
    swatch.style.backgroundColor = "rgb(" + red + "," +
                                    green + "," +
                                    blue + ")";

    //get hexVals
    var hexRed = Number(red).toString(16).toUpperCase();
    var hexGreen = Number(green).toString(16).toUpperCase();
    var hexBlue = Number(blue).toString(16).toUpperCase();

    //update output
    outRed.value = "Red: " + red + " (" + hexRed + ")";
    outGreen.value = "Green: " + green + " (" + hexGreen + ")";
    outBlue.value = "Blue: " + blue + " (" + hexBlue + ")";

  } // end changeColor
  </script>

</head>
<body>
  <h1>Color Chooser</h1>
  <div id = "swatch"></div>

  <form action="">
    <output id = "outRed">Red: 128 (80)</output>
    <input id = "sldRed"
          type = "range"
          min = "0"
          max = "255"
          value = "128"
          onchange = "changeColor()" />

    <output id = "outGreen">Green: 128 (80)</output>
```

```
    <input id = "sldGreen"
           type = "range"
           min = "0"
           max = "255"
           value = "128"
           onchange = "changeColor()" />

  <output id = "outBlue">Blue: 128 (80)</output>
  <input id = "sldBlue"
           type = "range"
           min = "0"
           max = "255"
           value = "128"
           onchange = "changeColor()" />
  </form>
</body>
</html>
```

3.2.4 Hex Education

Programmers like to convert to another format that's easier to work with. Believe it or not, it's easier to convert binary numbers to base 16 than base 10, so that's what programmers do. You can survive just fine without understanding base 16 (also called hexadecimal or hex) conversion, but you should understand a few key features, such as:

- ✓ Each hex digit is shorthand for four digits of binary. The whole reason programmers use hex is to simplify working with binary.
- ✓ Each digit represents a value between 0 and 15. Four digits of binary represent a value between 0 and 15.
- ✓ We have to invent some digits. The whole reason hex looks so weird is the inclusion of characters. This is for a simple reason: There aren't enough numeric digits to go around! The ordinary digits 0–9 are the same in hex as they are in base 10, but the values from 10–15 (base ten) are represented by alphabetic characters in hexadecimal.

You're very used to seeing the value 10 as equal to the number of fingers on both hands, but that's not always the case when you start messing around with numbering systems like we're doing here. The number 10 simply means one of the current base. Until now, you may have never used any base but base ten, but all that changes today. The numeral 10 is ten in base ten, but in base two, 10 means two. In base eight, 10 means eight, and in base sixteen, 10 means sixteen. This is important because when you want to talk about the number of digits on your hands in hex, you can't use the familiar notation 10 because in hex 10 means sixteen. We need a single-digit value to represent ten, so computer scientists legislated themselves out of this mess by borrowing letters. 10 is A, 11 is B, and

15 is F.

If all this math theory is making you dizzy, don't worry, but you need to understand to use hex colors:

- ✓ A color requires six digits of hex. A pixel requires three colors, and each color uses eight digits of binary. Two digits of hex cover each color. Two digits represent red, two for green, and finally two for blue.
- ✓ Hex color values usually begin with a sign. To warn the browser that a value will be in hexadecimal, the value is usually preceded with a sign (#). So, yellow is #FFFF00.

Working with colors in hex may seem really crazy and difficult, but it has some important advantages:

- ✓ **Precision**: Using this system gives you a huge number of colors to work with (over 16 million, if you really want to know). There's no way you could come up with that many color names on your own. Well, you could, but you'd be very, very old by the time you were done.
- ✓ **Objectivity**: Hex values aren't a matter of opinion. There could be some argument about which color is burnt sienna, but hex value #666600 is unambiguous.
- ✓ **Portability**: Most graphic editing software uses the hex system, so you can pick any color of an image and get its hex value immediately. This would make it easy to find your cat's eye color for that online shrine.
- ✓ **Predictability**: After you understand how it works, you can take any hex color and convert it to a value that's a little darker, a little brighter, or that has a little more blue in it. This is difficult to do with named colors.
- ✓ **Ease of use**: This one may seem like a stretch, but after you understand the next section, it's very easy to get a rough idea of a color and then tweak it to make exactly the form you're looking for.

3.2.5 Choosing Your Colors

Colors can seem overwhelming, but with a little bit of practice, you'll be managing colors with style. The point is to make things look good on your pages. To add color to your pages, do the following:

- ✓ Step 1

  Define the HTML as normal.

- ✓ Step 2

  The HTML shouldn't have any relationship to the colors. Add the color strictly in CSS.

- ✓ Step 3

Add a <style> tag to the page in the header area. Don't forget to set the type = "text/css" attribute. Add a selector for each tag you want to modify. You can modify any HTML tag, so if you want to change all the paragraphs, add a p { } selector. Use the tag name without the angle braces, so <h1> becomes h1{ }.

✓ Step 4

Add color and background-color attributes. You'll discover many more CSS elements you can modify, but for now, stick to color and background-color.

✓ Step 5

Specify the color values with color names or hex color values

3.2.6 Changing CSS on the Chrome Web Browser

The Chrome web browser has an especially cool trick when it comes to CSS coding. You can look at the CSS of any element on a web page and change it, seeing the results in real time!

Here's how it works:

✓ Step 1

Build the page in the normal way. Use your text editor to build the basic page.

✓ Step 2

Add CSS selectors. Specify the CSS for the elements you intend to change. You can put any values you want in the CSS, or you can simply leave the CSS blank for now. If you want to experiment, you can add selectors for elements described on the page.

✓ Step 3

Load your page in Chrome. The other browsers are starting to develop tools like Chrome, but it's clearly the leader, so start with Chrome.

✓ Step 4

Inspect an element. Right-click any element and choose Inspect element from the resulting pop-up menu.

✓ Step 5

Gasp in wonderment at the awesome developer tools. Figure 3-7 shows the developer tools that pop up when you inspect an element. Keep it in the Elements tab for now.

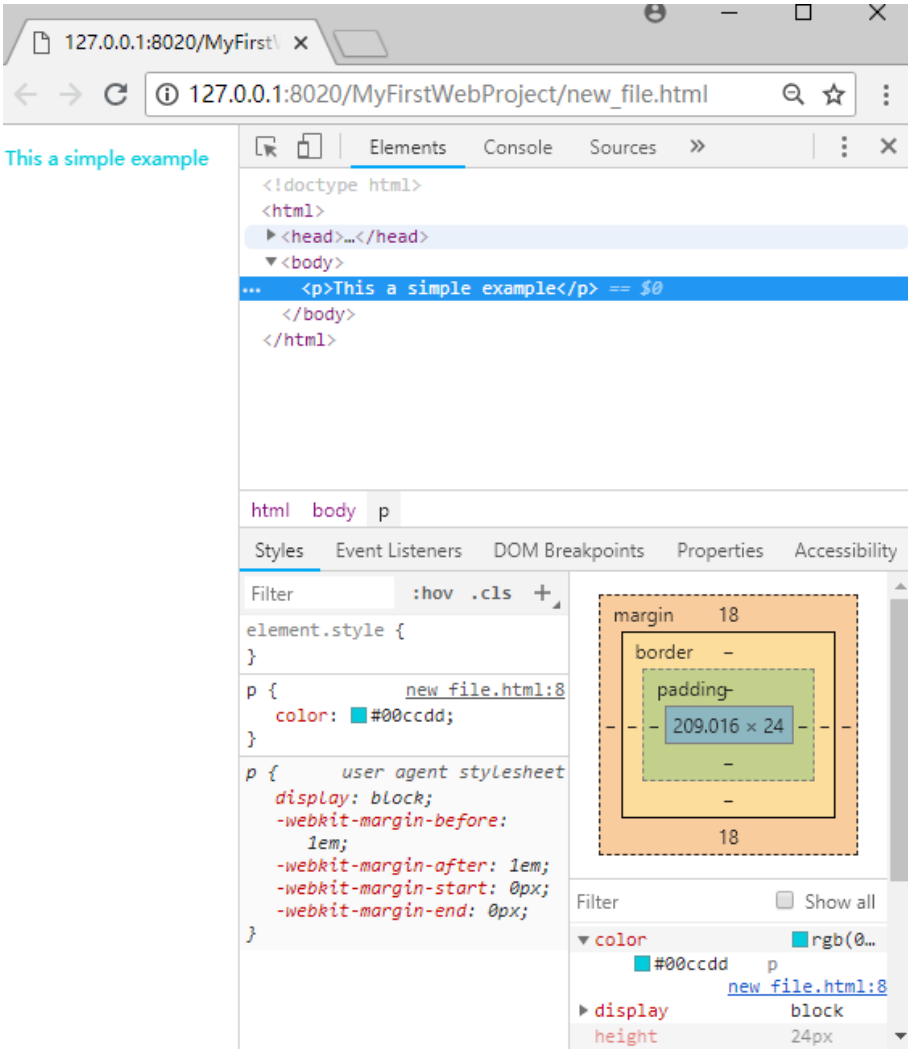


Figure 3-7 the chrome developer tools

✓ Step 6

Change the HTML code! You can double-click the code in the code viewer and modify the contents. This is fun, but not permanent or especially helpful.

✓ Step 7

You can also modify the CSS. If a style selector has been defined, it appears under the Styles tab in the Matched CSS Rules section. You can add new style rules or change the existing ones, and you'll be able to see the results on the fly.

✓ Step 8

You can even use a fancy color selector. When a color rule has been defined, you'll see a little color swatch. Click on that color to get a nice color selector you can use.

✓ Step 9

Select different parts of the page to modify other rules. You can modify the CSS of any element

as long as some sort of rule has been saved.

- ✓ Step 10

  Copy and paste any style rules you want to keep. Modifications made in the web developer toolbar are not permanent. If you find colors or other style rules you like, you can copy them from the developer window and paste them into your code.

The technical side of setting colors isn't too difficult, but deciding what colors to use can be a challenge. A little bit of subjectivity is in the process, but a few tools and rules can get you started.

3.3 Styling Text

Web pages are still primarily a text-based media, so you'll want to add some formatting capabilities. HTML doesn't do any meaningful text formatting on its own, but CSS adds a wide range of tools for choosing the typeface, font size, decorations, alignment, and much more. In this section, you discover how to manage text the CSS way.

A bit of semantics is in order. The thing most people think a **font** is more properly a **typeface**. Technically, a font is a particular typeface at a particular size with a specific set of decorations (underlining, italic, and so on). The distinction is honestly not that important in a digital setting. You don't explicitly set the font in CSS. You determine the font family (which is essentially a typeface), and then you modify its characteristics (creating a font as purists would think of it). Still, when I'm referring to the thing that most people call a font (a file in the operating system that describes the appearance of an alphabet set), I use the familiar term font.

3.3.1 Setting the Font Family

To assign a font family to part of your page, use some new CSS. Figure 3-8 illustrates a page with the heading set to Comic Sans MS. If this page is viewed on a Windows machine, it generally displays the font correctly because Comic Sans MS is installed with most versions of Windows. If you're on another type of machine, you may get something else. More on that in a moment, but for now, look at the simple case.

Here's the code:

Example 3-7 set the font family of heading

```
<!DOCTYPE html>
<html>
<head>
<meta charset = "UTF-8">
<title>comicHead.html</title>
<style type = "text/css">
```

```
h1 {
font-family: "Comic Sans MS";
}
</style>
</head>
<body>
<h1>This is a heading</h1>
<p>
This is ordinary text.
</p>
</body>
</html>
```

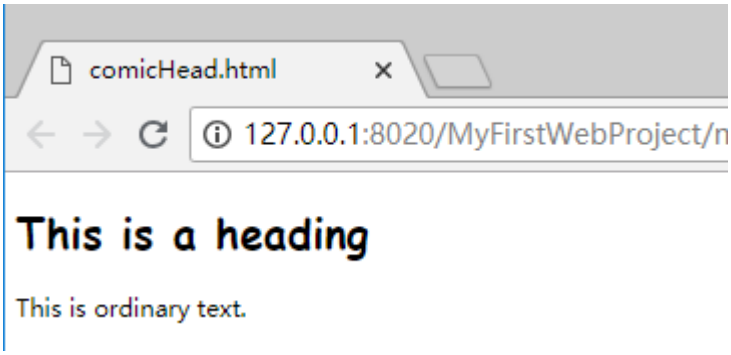


Figure 3-8 result of font family setting

The secret to this page is the CSS font-family attribute. Like most CSS elements, this can be applied to any HTML tag on your page. In this particular case, I applied it to my level one heading.

```
h1 {
font-family: "Comic Sans MS";
}
```

You can then attach any font name you wish, and the browser attempts to use that font to display the element. Even though a font may work perfectly fine on your computer, it may not work if that font isn't installed on the user's machine.

If you run exactly the same page on an iPad, you might see the result shown in Figure 3-9

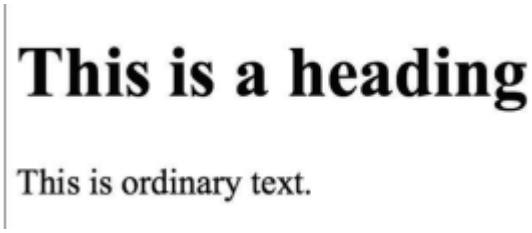


Figure 3-9 on an iPad, the heading might not be the same font

The specific font Comic Sans MS is installed on Windows machines, but the MS stands for Microsoft. This font isn't always installed on Linux or Mac. (Sometimes it's there, and sometimes it isn't.) You can't count on users having any particular fonts installed.

The Comic Sans font is fine for an example, but it has been heavily over-used in web development. Serious web developers avoid using it in real applications because it tends to make your page look amateurish.

3.3.2 Using Generic Fonts

It's a little depressing. Even though it's easy to use fonts, you can't use them freely because you don't know if the user has them. Fortunately, you can do a few things that at least increase the odds in your favor. The first trick is to use generic font names. These are virtual font names that every compliant browser agrees to support. Figure 3-10 shows a page with all the generic fonts.

Figure 3-10 generic fonts

Here is the code:

```
<!DOCTYPE html>
<html >
  <head>
    <meta charset = "UTF-8">
    <title>fontFamilyDemo </title>
  </head>
  <body>
    <h1>Generic Font Names</h1>
    <ul>
```

```
    <li style="font-family: serif;">serif</li>
    <li style="font-family: sans-serif;">sans-serif</li>
    <li style="font-family: cursive;">cursive</li>
    <li style="font-family: fantasy;">fantasy</li>
    <li style="font-family: monospace;">monospace</li>
  </ul>
 </body>
</html>
```

The generic fonts really are families of fonts:

- **Serif**: These fonts have those little serifs (the tiny cross strokes that enhance readability). Print text (like the paragraph you're reading now) tends to use serif fonts, and they're the default font for most browsers. The most common serif typeface is Times New Roman or Times.
- **Sans Serif:** Sans serif fonts don't have the little feet. Generally, they're used for headlines or other emphasis. Sometimes, they're seen as more modern and cleaner than serif fonts, so sometimes they're used for body text. Arial is the most common sans serif font.
- **Cursive**: These fonts look a little like handwriting. In Windows, the script font is usually Comic Sans MS. Script fonts are used when you want a less formal look.
- **Fantasy**: Fantasy fonts are decorative. Just about any theme you can think of is represented by a fantasy font, from Klingon to Tolkien. You can also find fonts that evoke a certain culture, making English text appear to be Persian or Chinese. Fantasy fonts are best used sparingly, for emphasis, as they often trade readability for visual appeal.
- **Monospace**: Monospace fonts produce a fixed-width font like typewritten text. Monospace fonts are frequently used to display code. Courier is a common monospace font. All code listings in this textbook use a monospaced font.

Because the generic fonts are available on all standards-compliant browsers, you'd think you could use them confidently. Well, you can be sure they'll appear, but you still might be surprised at the result. Figure 3-10 shows the result in windows, Figure 3-11 shows the same page on an iPad.

Figure 3-11 generic font on an iPad

Macs display yet another variation because the fonts listed here aren't actual fonts. Instead, they're virtual fonts. A standards-compliant browser promises to select an appropriate stand in. For example, if you choose sans serif, one browser may choose to use Arial. Another may choose Chicago. You can always use these generic font names and know the browser can find something close, but there's no guarantee exactly what font the browser will choose. Still, it's better than nothing. When you use these fonts, you're assured something in the right neighborhood, if not exactly what you intended.

3.3.3 Making a List of Fonts

This uncertainty is frustrating, but you can take some control. You can specify an entire list of font names if you want. The browser tries each font in turn. If it can't find the specified font, it goes to the next font and on down the line.

You might choose a font that you know is installed on all Windows machines, a font found on Macs, and finally one found on all Linux machines. The last font on your list should be one of the generic fonts, so you'll have some control over the worst-case scenario.

Table 3-2 shows a list of fonts commonly installed on Windows, Mac, and Linux machines.

Table 3-2 Font Equivalents in three OS

Windows	Mac	Linux
Arial	Arial	Nimbus Sans L
Arial Black	Arial Black	
Comic Sans MS	Comic Sans MS	TSCu_Comic
Courier New	Courier New	Nimbus Mono L
Georgia	Georgia	Nimbus Roman No9L
Lucida Console	Monaco	
Palatino	Palatino	FreeSerif
Tahoma	Geneva	Kalimati
Times New Roman	Times	FreeSerif
Trebuchet MS	Helvetica	FreeSans
Verdana	Verdana	Kalimati

You can use this chart to derive a list of fonts to try. For example, look at the following style:

```
p {
font-family: "Trebuchet MS", Helvetica, FreeSans, sans-serif;
```

```
}
```

This style has a whole smorgasbord of options. First, the browser tries to load Trebuchet MS. If it's a Windows machine, this font is available, so that one displays. If that doesn't work, the browser tries Helvetica (a default Mac font). If that doesn't work, it tries FreeSans, a font frequently installed on Linux machines. If this doesn't work, it defaults to the old faithful sans serif, which simply picks a sans serif font. Note that font names of more than one word must be encased in quotes, and commas separate the list of font names.

Don't get too stressed about Linux fonts. It's true that the equivalencies are harder to find, but Linux users tend to fall into two camps: They either don't care if the fonts are exact, or they do care and they've installed equivalent fonts that recognize the common names. In either case, you can focus on Mac and Windows people for the most part, and, as long as you've used a generic font name, things work okay on a Linux box.

3.3.4 The Problem of Web-based Fonts

Fonts seem pretty easy at first, but some big problems arise with actually using them.

Understanding the problem

There used to be a tag in old-school HTML called the <font> tag. You could use this tag to change the size, color, and font family. There were also specific tags for italic (<i>), boldface (<b>), and centering ( ). These tags were very easy to use, but they caused some major problems. To use them well, you ended up littering your page with all kinds of tags trying to describe the markup, rather than the meaning. There was no easy way to reuse font information, so you often had to repeat things many times throughout the page, making it difficult to change. Web developers are now discouraged from using <font>, < i>, <b>, or tags. The CSS elements I show in this chapter more than compensate for this loss. You now have a more flexible, more powerful alternative.

The problem with fonts is this: Font resources are installed in each operating system. They aren't downloaded with the rest of the page. Your web page can call for a specific font, but that font isn't displayed unless it's already installed on the user's computer.

Say I have a cool font called Happygeek. It's installed on my computer, and when I choose a font in my word processor, it shows up in the list. I can create a word-processing document with it, and everything will work great.

If I send a printout of a document using Happygeek to someone, everything's great because the paper doesn't need the actual font. It's just ink. If I send her the digital file and tell her to open it on her computer, we'll have a problem. See, she's not that hip and doesn't have Happygeek installed. Her computer will pick some other font.

This isn't a big problem in word processing because people don't generally send around digital copies of documents with elaborate fonts in them. However, web pages are passed around **only** in digital form. To know which fonts you can use, you have to know what fonts are installed on the user's machine, and that's impossible.

Part of the concern is technical (figuring out how to transfer the font information to the browser), but the real issue is digital rights management. If you've purchased a font for your own use, does that give you the right to transfer it to others, so now they can use it without paying?

Using embedded fonts

Although a web developer can suggest any font for a web page, the font files are traditionally a client-level asset. If the client doesn't have the font installed, she won't see it. Fortunately, CSS3 supports a sensible solution for providing downloadable fonts, called @font-face. Figure 3-12 shows a page with a couple of embedded fonts.

Figure 3-12 a page includes embedded font

The @font-face style does not work like most CSS elements. It doesn't apply markup to some part of the page. Instead, it defines a new CSS value that can be used in other markup. Specifically, it allows you to place a font file on your server and define a font family using that font.

```
@font-face {
font-family: "waltograph42";
src: url("waltograph42.otf");
}
```

The font-family **attribute** indicates the name you will be giving this font in the rest of your CSS code. Typically it is similar to the font file name, but this is not required. The **src** attribute is the URL of the actual font file as it is found on the server. After a font-face has been defined, it can be used in an ordinary font-family attribute in the rest of your CSS code:

```
h1 {
font-family: waltograph42;
}
```

Here's the code for the custom font example:

Example 3-8 an example of embedded font

```
<!DOCTYPE html>
<head>
<title>Embedded Font</title>
<style type = "text/css">
```

```
@font-face {
font-family: "waltograph42";
src: url("waltograph42.otf");
}
@font-face {
font-family: "waltographUI";
src: url("waltographUI.ttf");
}
h1 {
font-family: waltograph42;
font-size: 300%;
}
h2 {
font-family: waltographUI;
}
</style>
</head>
<body>
<h1>Embedded Font Demo</h1>
<h2>Here's another custom font</h2>
</body>
</html>
```

Although all modern browsers support the @font-face feature, the actual file types supported vary from browser to browser. Here are the primary font types:

- ✓ **TTF**: The standard TrueType format is well-supported, but not by all browsers. Many open-source fonts are available in this format.
- ✓ **OTF**: This is similar to TTF, but is a truly open standard, so it is preferred by those who are interested in open standards. It is supported by most browsers except IE.
- ✓ **WOFF**: WOFF is a proposed standard format currently supported by Firefox. Microsoft has hinted at supporting this format in IE.
- ✓ **EOT**: This is Microsoft's proprietary embedded font format. It only works in IE, but to be fair, Microsoft has had embedded font support for many years.

When you use this technique, you need to have a copy of the font file locally. For now, it should be in the same directory as your web page (just as you do with images.) When you begin hosting on a web server, you'll want to move your font file to the server along with all the other resources your web page needs. Just because you can include a font doesn't mean you should. Think carefully about readability. Also, be respectful of intellectual property. Fortunately there are many excellent free open-source fonts available. Begin by looking at Open Font Library (http://openfontlibrary.org/). Google Fonts (www.google.com/fonts/) is another great resource for free fonts. With the Google Font tool, you can select a font embedded on Google's servers, and you can copy code that makes the font available without downloading.

3.3.5 Specifying the Font Size

Like font names, font sizes are easy to change in CSS, but there are some hidden traps.

Size is only a suggestion

In print media, after you determine the size of the text, it pretty much stays there. The user can't change the font size in print easily. By comparison, web browsers frequently change the size of text. A cellphone-based browser displays text differently than one on a high-resolution LCD panel. Further, most browsers allow the user to change the size of all the text on the screen. Use Ctrl++ (plus sign) and Ctrl+– (minus sign) to make the text larger or smaller. In older versions of IE (prior to IE7), choose the Text Size option from the Page menu to change the text size.

The user should really have the ability to adjust the font size in the browser. When I display a web page on a projector, I often adjust the font size so students in the back can read. Some pages have the font size set way too small for me to read.

Determining font sizes precisely is counter to the spirit of the web. If you declare that your text will be exactly 12 points, for example, one of two things could happen:

- ✓ The browser might enforce the 12-point rule literally. This takes control from the user, so users who need larger fonts are out of luck. Older versions of IE do this.
- ✓ The user might still change the size. If this is how the browser behaves (and it usually is), 12 points doesn't always mean 12 points. If the user can change font sizes, the literal size selection is meaningless.

The web developer should set up font sizes, but only in relative terms. Don't bother using absolute measurements (in most cases) because they don't really mean what you think. Let the user determine the base font size and specify relative changes to that size.

Using the font-size style attribute

The basic idea of font size is pretty easy to grasp in CSS. Take a look at font Example 3-9 in Figure 3-13.

Example 3-9 an example of font size

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>fontSize </title>
    </head>
    <body>
        <h1 style="font-size: 100%;">Font Sizes</h1>
        <p>This paragraph is normal</p>
        <p style="font-size:small;">
            This paragraph is small
```

```
        </p>
        <p style="font-size: 0.5em;">
            This paragraph is half size
        </p>
        <p style="font-size: 2em;">
            This paragraph is double size
        </p>
    </body>
</html>
```

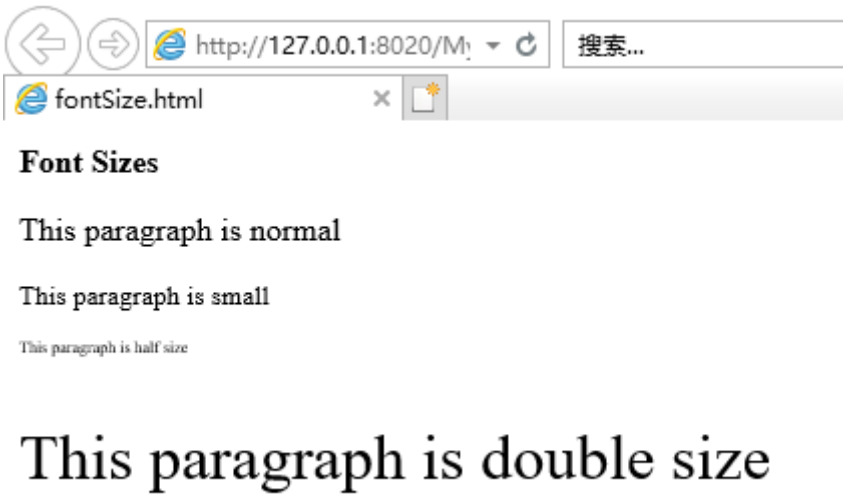


Figure 3-13 an example of font size

This page obviously shows a number of different font sizes. The line “Font Sizes” is an ordinary h1 element. All the other lines are paragraph tags. They appear in different sizes because they have different styles applied to them.

Font sizes are changed with the (cleverly named) font-size attribute:

```
p {
font-size: small;
}
```

Simply indicate the font-size rule, and, well, the size of the font. In this example, I used the special value small, but there are many other ways to specify sizes in CSS.

3.3.6 Absolute measurement units

Many times, you need to specify the size of something in CSS. Of course, font size is one of these

cases. The different types of measurement have different implications. It's important to know there are two distinct kinds of units in CSS. **Absolute measurements** attempt to describe a particular size, as in the real world. **Relative measurements** are about changes to some default value. Generally, web developers are moving toward relative measurement for font sizes.

Points (pt)

In word processing, you're probably familiar with points as a measurement of font size. You can use the abbreviation **pt** to indicate you're measuring in points, for example:

```
p {
font-size: 12pt;
}
```

There is no space between 12 and pt.

Unfortunately, points aren't an effective unit of measure for web pages. Points are an absolute scale, useful for print, but they aren't reliable on the web because you don't know what resolution the user's screen has. A 12-point font might look larger or smaller on different monitors. In some versions of IE, after you specify a font size in points, the user can no longer change the size of the characters. This is unacceptable from a usability standpoint. Relative size schemes (which I describe later in this section) prevent this problem.

Pixels (px)

Pixels refer to the small dots on the screen. You can specify a font size in pixels, although that's not the way it's usually done. For one thing, different monitors make pixels in different sizes. You can't really be sure how big a pixel will be in relationship to the overall screen size. Different letters are different sizes, so the pixel size is a rough measurement of the width and height of the average character. Use the **px** abbreviation to measure fonts in pixels:

```
p {
font-size: 20px;
}
```

Traditional measurements (in, cm)

You can also use inches (in) and centimeters (cm) to measure fonts, but this is completely impractical. Imagine you have a web page displayed on both your screen and a projection system. One inch on your own monitor may look like ten inches on the projector. Real-life measurement units aren't meaningful for the web. The only time you might use them is if you'll be printing something and you have complete knowledge of how the printer is configured. If that's the case, you're better off using a print-oriented layout tool (like a word processor) rather than HTML.

3.3.7 Relative measurement units

Relative measurement is a wiser choice in web development. Use these schemes to change sizes in relationship to the standard size.

Named sizes

CSS has a number of font size names built in:

```
xx-small
x-small
small
medium
large
x-large
xx-large
```

It may bother you that there's nothing more specific about these sizes: How big is large? Well, it's bigger than medium. That sounds like a flip answer, but it's the truth. The user sets the default font size in the browser (or leaves it alone), and all other font sizes should be in relation to this preset size. The medium size is the default size of paragraph text on your page. For comparison purposes, <h1> tags are usually xx-large.

Percentage (%)

The percentage unit is a relative measurement used to specify the font in relationship to its normal size. Use 50% to make a font half the size it would normally appear and 200% to make it twice the normal size. Use the % symbol to indicate percentage, as shown here:

```
p {
font-size: 150%;
}
```

Percentages are based on the default size of ordinary text, so an <h1> tag at 100% is the same size as text in an ordinary paragraph.

```
Em (em)
```

In traditional typesetting, the em is a unit of measurement equivalent to the width of the "m" character in that font. In actual web use, it's really another way of specifying the relative size of a font. For instance, 0.5 ems is half the normal size, and 3 ems is three times the normal size. The term em is used to specify this measurement.

```
p {
font-size: 1.5em;
}
```

Here are the best strategies for font size:

- ✧ Don't change sizes without a good reason. Most of the time, the browser default sizes are perfectly fine, but there may be some times when you want to adjust fonts a little more.
- ✧ Define an overall size for the page. If you want to define a font size for the entire page, do so in the <body> tag. Use a named size, percentage, or ems to avoid the side effects of absolute sizing. The size defined in the body is applied to every element in the body automatically.
- ✧ Modify any other elements. You might want your links a little larger than ordinary text, for example. You can do this by applying a font size attribute to an element. Use relative measurement if possible.

Table 3-3 shows the unit conversion of different font size

Table 3-3 unit conversion of different size

Points	Pixels	Ems	Percent
6pt	**8px**	**0.5em**	**50%**
7pt	**9px**	**0.55em**	**55%**
7.5pt	**10px**	**0.625em**	**62.5%**
8pt	**11px**	**0.7em**	**70%**
9pt	**12px**	**0.75em**	**75%**
10pt	**13px**	**0.8em**	**80%**
10.5pt	**14px**	**0.875em**	**87.5%**
11pt	**15px**	**0.95em**	**95%**
12pt	**16px**	**1em**	**100%**
13pt	**17px**	**1.05em**	**105%**
13.5pt	**18px**	**1.125em**	**112.5%**
14pt	**19px**	**1.2em**	**120%**
14.5pt	**20px**	**1.25em**	**125%**
15pt	**21px**	**1.3em**	**130%**
16pt	**22px**	**1.4em**	**140%**
17pt	**23px**	**1.45em**	**145%**
18pt	**24px**	**1.5em**	**150%**
20pt	**26px**	**1.6em**	**160%**
22pt	**29px**	**1.8em**	**180%**
24pt	**32px**	**2em**	**200%**

3.3.8 Determining Other Font Characteristics

In addition to size and color, you can change fonts in a number of other ways. Figure 3-14 shows a number of common text modifications you can make. The various paragraphs in this page are modified in different ways. You can change the alignment of the text as well as add italic, bold, underline, or strikethrough to the text.

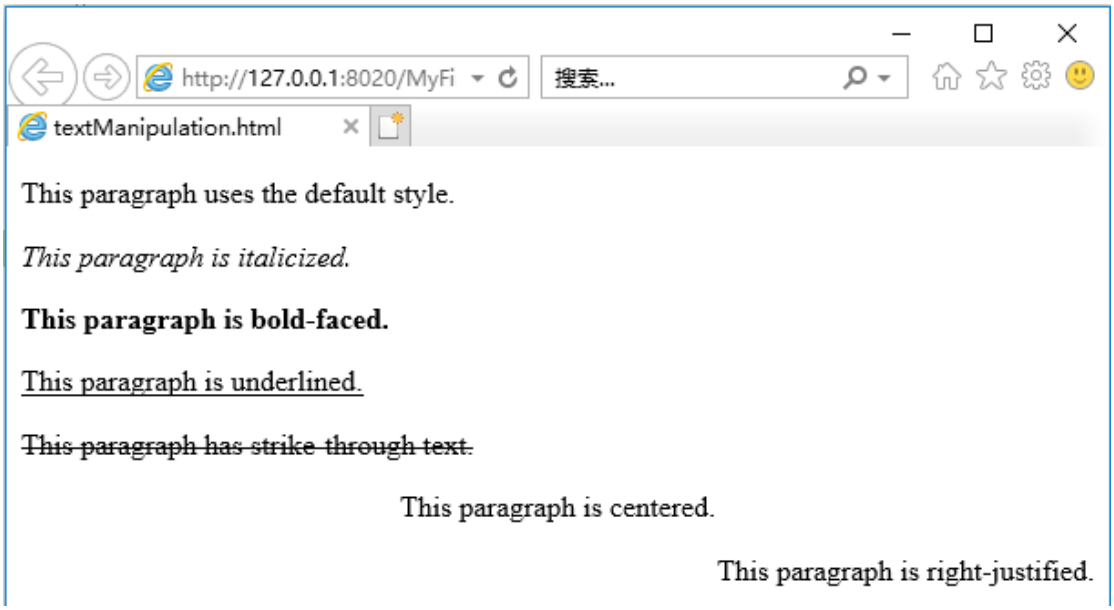


Figure 3-14 other font characters

The source code is shown as Example 3-10.

Example 3-10 other font characters

```
<!DOCTYPE html>
<html >
  <head>
    <meta charset = "UTF-8">
    <title>textManipulation </title>
  </head>
  <body>
    <p>
      This paragraph uses the default style.
    </p>
    <p style="font-style: italic;">
      This paragraph is italicized.
    </p>
    <p style="font-weight: bold;">
      This paragraph is bold-faced.
    </p>
    <p style="text-decoration: underline;">
      This paragraph is underlined.
    </p>
    <p style="text-decoration: line-through;" >
      This paragraph has strike-through text.
    </p>
    <p style="text-align: center;">
      This paragraph is centered.
```

```
    </p>
    <p style="text-align: right;">
      This paragraph is right-justified.
    </p>
  </body>
</html>
```

CSS uses a potentially confusing set of rules for the various font manipulation tools. One rule determines the font style, and another determines boldness.

I describe these techniques in the following sections for clarity.

Using font-style for italics

The font-style attribute allows you to make italic text, as shown in Figure 3-15. Here's some code illustrating how to add italic formatting:

Example 3-11 create italic text with font-style

```
<!DOCTYPE html>
<html >
<head>
<meta charset = "UTF-8">
<title>italics.html</title>
<style type = "text/css">
p {
font-style: italic;
}
</style>
</head>
<body>
<h1>Italics</h1>
<p>This paragraph is in italic form.</p>
</body>
</html>
```

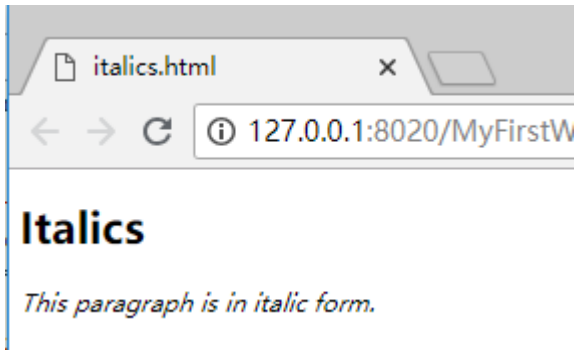


Figure 3-15 create italic text with font-style

The font-style values can be italic, normal, or oblique (tilted toward the left). If you want to set a

particular segment to be set to italic, normal, or oblique style, use the font-style attribute.

Using font-weight for bold

You can make your font bold by using the font-weight CSS attribute, as shown in Example 3-12, the result is displayed in Figure 3-16.

Example 3-12 font-weight for bold

```
<!DOCTYPE html>
<html>
<head>
<meta charset = "UTF-8">
<title>bold.html</title>
<style type = "text/css">
p {
font-weight: bold;
}
</style>
</head>
<body>
<h1>Boldface</h1>
<p>
This paragraph is bold.
</p>
</body>
</html>
```

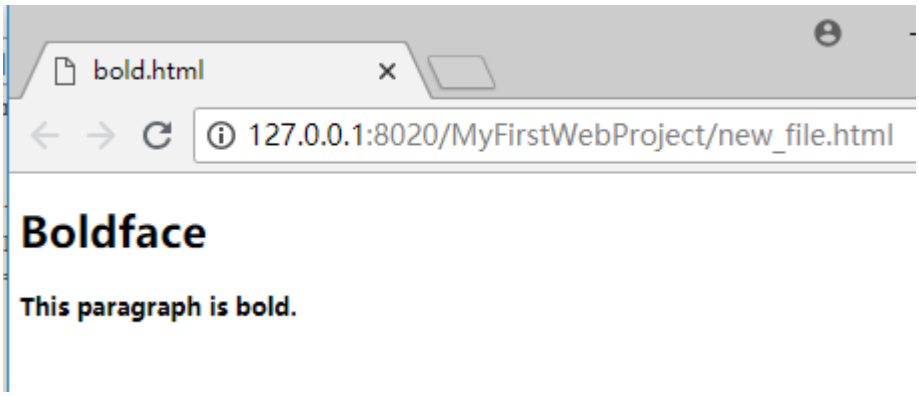


Figure 3-16 font-weight for bold

If you want to make some of your text bold, use the font-weight CSS attribute, Font weight can be defined a couple ways. Normally, you simply indicate bold in the font-weight rule, as I did in this code. You can also use a numeric value from 100 (exceptionally light) to 900 (dark bold).

Using text-decoration

Text-decoration can be used to add a couple other interesting formats to your text, including underline, strikethrough, and overline. For example, the following code produces an underlined paragraph:

Example 3-13 underline text

```
<!DOCTYPE html>
<html lang = "en-US">
<head>
<meta charset = "UTF-8">
<title>underline.html</title>
<style type = "text/css">
p {
text-decoration: underline;
}
</style>
</head>
<body>
<h1>Underline</h1>
<p>
This paragraph is underlined.
</p>
</body>
</html>
```

The result is shown as Figure 3-17.

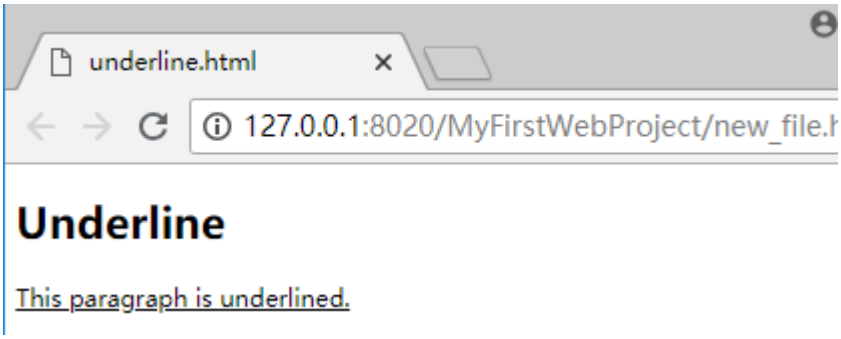


Figure 3-17 underline text

Be careful using underline in web pages. Users have been trained that underlined text is a link, so they may click your underlined text expecting it to take them somewhere.

You can also use text-decoration for other effects, such as strikethrough (called “line-through” in CSS), as shown in the following code:

Example 3-14 line through text

```
<!DOCTYPE html>
```

```
<html lang = "en-US">
<head>
<meta charset = "UTF-8">
<title>strikethrough.html</title>
<style type = "text/css">
p {
text-decoration: line-through;
}
</style>
</head>
<body>
<h1>Strikethrough</h1>
<p>
This paragraph has strikethrough text.
</p>
</body>
</html>
```

This code produces a page similar to Figure 3-18.

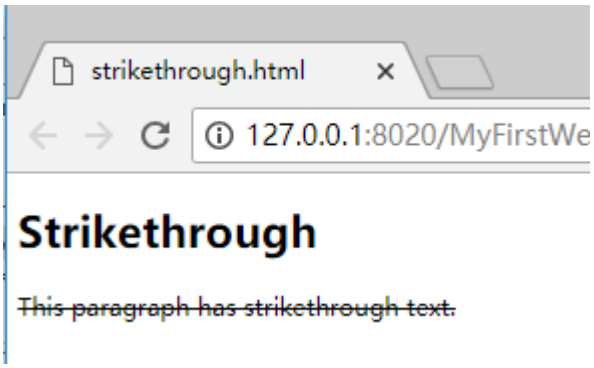


Figure 3-18 line through text

Using text-align for basic alignment

You can use the text-align attribute to center, left-align, or right-align text, as shown in the following code:

Example 3-15 center align text

```
<!DOCTYPE html>
<html lang = "en-US">
<head>
<meta charset = "UTF-8">
<title>center.html</title>
<style type = "text/css">
p {
text-align: center;
}
```

```
</style>
</head>
<body>
<h1>Centered</h1>
<p>This paragraph is centered.</p>
</body>
</html>
```

You can also use the text-align attribute to right- or left-justify your text. The page shown in Figure 3-19 illustrates the text-align attribute.

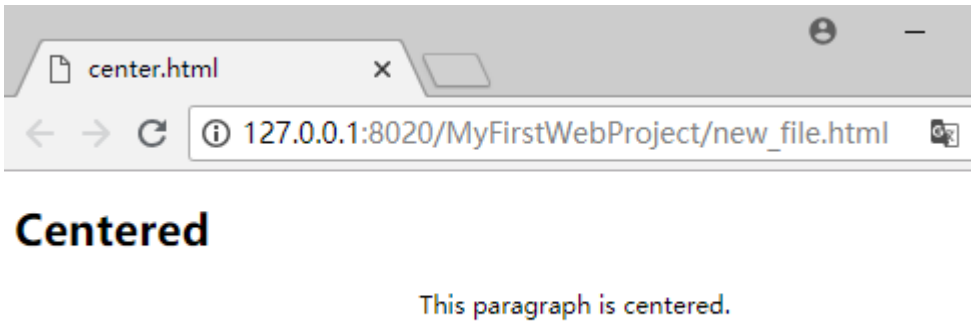


Figure 3-19 center align text

You can apply the text-align attribute only to text. The old tag could be used to center nearly anything (a table, some text, or images), which was pretty easy but caused problems.

Other text attributes

CSS offers a few other text manipulation tools, but they're rarely used:

- ✓ **Font-variant**: Can be set to small-caps to make your text use only capital letters. Lowercase letters are shown in a smaller font size.
- ✓ **Letter-spacing**: Adjusts the spacing between letters. It's usually measured in ems. Fonts are so unpredictable on the web that if you're trying to micromanage this much, you're bound to be disappointed by the results.
- ✓ **Word-spacing**: Allows you to adjust the spacing between words.
- ✓ **Text-indent**: Lets you adjust the indentation of the first line of an element. This value uses the normal units of measurement. Indentation can be set to a negative value, causing an outdent if you prefer.
- ✓ **Vertical-align**: Used when you have an element with a lot of vertical space (often a table cell). You can specify how the text behaves in this situation.
- ✓ **Text-transform**: Helps you convert text into uppercase, lowercase, or capitalized (first letter uppercase) forms.
- ✓ **Line-height**: Indicates the vertical spacing between lines in the element. As with letter and

word spacing, you'll probably be disappointed if you're this concerned about exactly how things are displayed.

Using the font shortcut

It can be tedious to recall all the various font attributes and their possible values. There's another technique often used by the pros. The font rule provides an easy shortcut to a number of useful font attributes. The Example 3-16 shows you how to use the font rule:

Example 3-16 font shortcut

```
<!DOCTYPE html>
<html lang = "en-US">
<head>
<meta charset = "UTF-8">
<title>fontTag.html</title>
<style type = "text/css">
p {
font: bold italic 150% cursive;
}
</style>
</head>
<body>
<h1>Using Font shortcut</h1>
<p>
This paragraph has many settings.
</p>
</body>
</html>
```

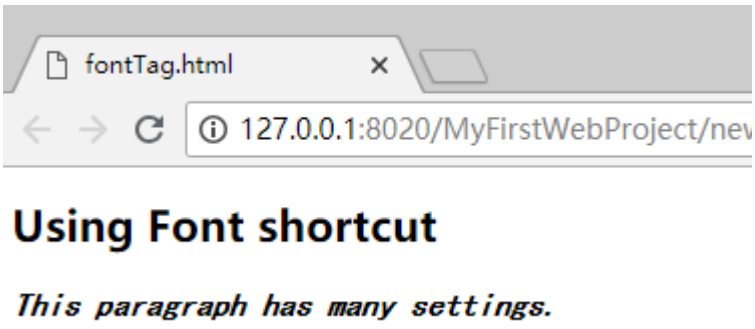


Figure 3-20 the font shortcut

Figure 3-20 illustrates the powerful font rule in action.

The great thing about the font rule is how it combines many of the other font-related rules for a simpler way to handle most text-formatting needs. The font attribute is extremely handy. Essentially, it allows you to roll all the other font attributes into one. Here's how it works:

- ✓ Specify the font rule in the CSS.
- ✓ List any font-style attributes. You can mention any attributes normally used in the font- style rule (italic or oblique). If you don't want either, just move on.
- ✓ List any font-variant attributes. If you want small caps, you can indicate it here. If you don't, just leave this part blank.
- ✓ List any font-weight values. This can be "bold" or a font-weight number (100–900).
- ✓ Specify the font-size value in whatever measurement system you want (but ems or percentages are preferred). Don't forget the measurement unit symbol (em or %) because that's how the font rule recognizes that this is a size value.
- ✓ Indicate a font-family list last. The last element is a list of font families you want the browser to try. This list must be last, or the browser may not interpret the font attribute correctly.

The font rule is great, but it doesn't do everything. You still may need separate CSS rules to define your text colors and alignment. These attributes aren't included in the font shortcut.

Don't use commas to separate values in the font attribute list. Use commas only to separate values in the list of font-family declarations. You can skip any values you want as long as the order is correct. For example,

```
font: italic "Comic Sans MS", cursive;
```

It is completely acceptable, as is

```
font: 70% sans-serif;
```

Working with subscripts and superscripts

Occasionally, you'll need superscripts (characters that appear a little bit higher than normal text, like exponents and footnotes) or subscripts (characters that appear lower, often used in mathematical notation). Figure 3-21demonstrates a page with these techniques.

Figure 3-21 superscripts and subscripts

Surprisingly, you don't need CSS to produce superscripts and subscripts. These properties are managed through HTML tags. You can still style them the way you can any other HTML tag. It is shown as Example 3-17

Example 3-17 producing superscripts and subscripts by tag

```
<!DOCTYPE html>
```

```
<html lang = "en-US">
<head>
<meta charset = "UTF-8">
<title>SuperSub.html</title>
</head>
<body>
<p>
A<sup>2</sup> + B<sup>2</sup> = C<sup>2</sup>
</p>
<p>
i<sub>0</sub> = 0
</p>
</body>
</html>
```

3.4 Selectors: Coding with Class and Style

You know how to use CSS to change all the instances of a particular tag, but what if you want to be more selective? For example, you might want to change the background color of only one paragraph, or you might want to define some special new type of paragraph. Maybe you want to specify a different paragraph color for part of your page, or you want visited links to appear differently from unselected links. The part of the CSS style that indicates what element you want to style is a selector. In this section, you discover powerful new ways to select elements on the page.

3.4.1 Selecting Particular Segments

Apart from its cultural merit, Figure 3-22 is interesting because it has three different paragraph styles. The introductory paragraph is normal. The quote is set in italicized font, and the attribution is monospaced and right-aligned. The quote in the following code was generated by sites on the Internet: the Shakespearean insult generator. Nothing is more satisfying than telling somebody off in iambic pentameter (www.pangloss.com/seidel/Shaker/index.html.)

Figure 3-22 the page has three types paragraphs

The source code of the page is displayed as Example 3-18

Example 3-18 the page has three types paragraphs

```
<!DOCTYPE html>
<html >
<head>
<meta charset = "UTF-8">
<title>quote.html</title>
<style type = "text/css">
#quote {
font: bold italic 130% Garamond, Comic Sans MS, fantasy;
text-align: center;
}
#author {
font-size: 80%;
font-family:monospace;
text-align: right;
}
</style>
</head>
<body>
<h1>Literature Quote of the Day</h1>
<p> How to tell somebody off the classy way: </p>
<p id = "quote">[Thou] leathern-jerkin, crystal-button, knot-pated,
agate-ring, puke-stocking, caddis-garter, smooth-tongue, Spanish
pouch! </p>
<p id = " author"> -William Shakespeare (Henry IV Part I) </p>
</body>
</html>
```

3.4.2 Styling identified paragraphs

Until now, you've used CSS to apply a particular style to an element all across the page. For example, you can add a style to the <p> tag, and that style applies to all the paragraphs on the page.

Sometimes, you want to give one element more than one style. You can do this by naming each element and using the name in the CSS style sheet. Here's how it works:

- ✓ Step 1. Add an id attribute to each HTML element you want to modify. For example, the paragraph with the attribution now has an id attribute with the value author.

  <p id = "author">

- ✓ Step 2. Make a style in CSS. Use a pound sign followed by the element's ID in CSS to specify you're not talking about a tag type any more, but a specific element: For example, the CSS code contains the selector #author, meaning, "Apply this style to an element with the attribution id."

  #author {

- ✓ Step 3. Add the style. Create a style for displaying your named element. In this case, I want the paragraph with the author ID right-aligned, monospace, and a little smaller than normal. This style will be attached only to the specific element.

  #author {

  font-size: 80%;

  font-familiy: monospace;

  text-align: right;

  }

The ID trick works great on any named element. IDs have to be unique (you can't repeat the same ID on one page), so this technique is best when you have a style you want to apply to only one element on the page. It doesn't matter what HTML element it is (it could be a heading 1, a paragraph, a table cell, or whatever). If it has the ID quote, the #quote style is applied to it. You can have both ID selectors and ordinary (element) selectors in the same style sheet.

3.4.3 Using Emphasis and Strong Emphasis

You may be shocked to know that HTML doesn't allow italics or bold. Old style HTML had the <i> tag for italics and the <b> tag for bold. These seem easy to use and understand. Unfortunately, they can trap you. In your HTML5, you shouldn't specify how something should be styled. You should specify instead the purpose of the styling. The <i> and <b> tags in XHTML Strict are removed in HTML5 and replaced with <**em**> and <**strong**>. The <**em**> tag means emphasized. By default, **em** italicizes your text. The <**strong**> tag stands for strong emphasis. It defaults to bold. Figure 3-23 illustrates a page with the default styles for **em** and strong.

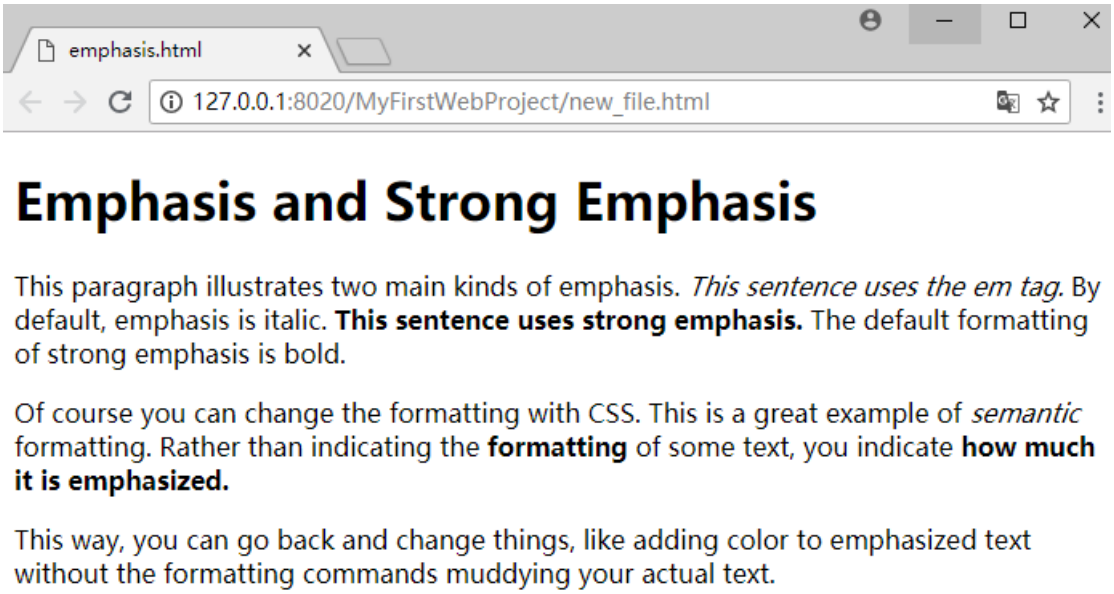


Figure 3-23 Two kinds of emphasis semantic tag

Here it is the source code:

Example 3-19 two kinds of emphasis semantic tag

```
<!DOCTYPE html>
<html >
<head>
<meta charset = "UTF-8">
<title>emphasis.html</title>
</head>
<body>
<h1>Emphasis and Strong Emphasis</h1>
<p> This paragraph illustrates two main kinds of emphasis. <em>This
sentence uses the em tag.</em> By default, emphasis is italic.
<strong>This sentence uses strong emphasis.</strong> The default
formatting of strong emphasis is bold. </p>
<p>Of course you can change the formatting with CSS. This is a great
example of <em>semantic</em> formatting. Rather than indicating the
<strong>formatting</strong> of some text, you indicate <strong>how
much it is emphasized.</strong></p>
<p> This way, you can go back and change things, like adding color
to emphasized text without the formatting commands muddying your
actual text. </p>
</body>
</html>
```

It'd be improper to think that em is just another way to say italic and strong is another way to say bold. In the old scheme, after you define something as italic, you're pretty much stuck with that. The HTML way describes the meaning, and you can define it how you want.

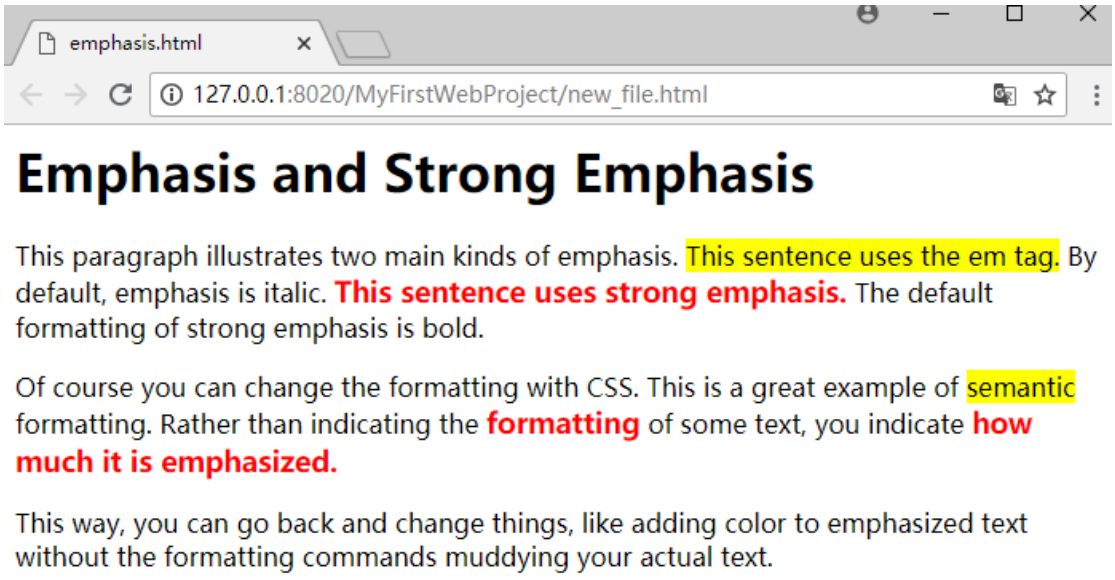


Figure 3-24 you can modify the style of em emphasis

Figure 3-24 shows how you might modify the levels of emphasis. I used yellow highlighting (without italics) for em and a larger red font for strong.

The HTML code is identical to the code for the earlier. The only difference is the addition of a style sheet. The style sheet is embedded in the web page between style tags.

Example 3-20 you can modify the style of emphasis semantic tag

```
<html >
<head>
<meta charset = "UTF-8">
<title>emphasis.html</title>
<style type = "text/css">
em {
font-style: normal;
background-color: yellow;
}
strong {
color: red;
font-size: 110%;
}
</style>
</head>
<body>
<h1>Emphasis and Strong Emphasis</h1>
<p> This paragraph illustrates two main kinds of emphasis. <em>This
sentence uses the em tag.</em> By default, emphasis is italic.
```

```
<strong>This sentence uses strong emphasis.</strong> The default
formatting of strong emphasis is bold. </p>
<p>Of course you can change the formatting with CSS. This is a great
example of <em>semantic</em> formatting. Rather than indicating the
<strong>formatting</strong> of some text, you indicate <strong>how
much it is emphasized.</strong></p>
<p> This way, you can go back and change things, like adding color
to emphasized text without the formatting commands muddying your
actual text. </p>
</body>
</html>
```

The style is used to modify the HTML. The meaning in the HTML stays the same — only the style changes.

The semantic markups are more useful than the older (more literal) tags because they still tell the truth even if the style has been changed. (In the HTML code, the important thing is whether the text is emphasized, not what it means to emphasize the text. That job belongs to CSS.)

3.4.4 Defining Classes

You can easily apply a style to all the elements of a particular type in a page, but sometimes you might want to have tighter control of your styles. For example, you might want to have more than one paragraph style. As an example, look at the page shown in Figure 3-25.

Figure 3-25 using class to define style for different paragraphs

Once again, multiple formats are on this page:

- ✓ Questions have a large italic sans serif font. There's more than one question.
- ✓ Answers are smaller, blue, and in a cursive font. There's more than one answer, too.

Questions and answers are all paragraphs, so you can't simply style the paragraph because you need two distinct styles. There's more than one question and more than one answer, so the ID trick would be problematic. Two different elements can't have the same ID. This is where the notion of classes comes into play. Every ID belongs to a single element, but many elements (even of different types) can share the same class.

CSS allows you to define classes in your HTML and make style definitions that are applied across a class. It works like this:

- ✓ Step 1. Add the class attribute to your HTML questions. Unlike ID, several elements can share the same class. All my questions are defined with this variation of the tag. Setting the class to question indicates these paragraphs will be styled as questions:

  What kind of cow lives in the Arctic?

- ✓ Step 2. Add similar class attributes to the answers by setting the class of the answers to answer:

  An Eskimoo!

  Now you have two different subclasses of paragraph: question and answer.

- ✓ Step 3. Create a class style for the questions. The class style is defined in CSS. Specify a class with the period (.) before the class name. Classes are defined in CSS like this:

  <style type = "text/css">

  .question {

  font: italic 150% arial, sans-serif;

  text-align: left;

  }

  In this situation, the question class is defined as a large sans serif font aligned to the left.

- ✓ Step 4. Define the look of the answers. The answer class uses a right-justified cursive font.

  .answer {

  font: 120% "Comic Sans MS", cursive;

  text-align: right;

  color: #00F;

  }

</style>

Here's the code for the page:

Example 3-21 using class to define style

```
<!DOCTYPE html>
<html>
<head>
<meta charset = "UTF-8">
<title>classes.html</title>
<style type = "text/css">
.question {
font: italic 150% arial, sans-serif;
text-align: left;
}
.answer {
font: 120% "Comic Sans MS", cursive;
text-align: right;
color: #00F;
}
</style>
</head>
<body>
<h1>Favorite Jokes</h1>
<p class = "question">
What kind of cow lives in the Arctic?
</p>
<p class = "answer">
An Eskimoo!
</p>
<p class = "question">
What goes on top of a dog house?
</p>
<p class = "answer">
The woof!
</p>
</body>
</html>
```

Sometimes you see selectors, like

p.fancy

that include both an element and a class name. This style is applied only to paragraphs with the fancy class attached. Generally, I like classes because they can be applied to all kinds of things, so I usually leave the element name out to make the style as reusable as possible.

One element can use more than one class. Figure 3-26 shows an example of this phenomenon.

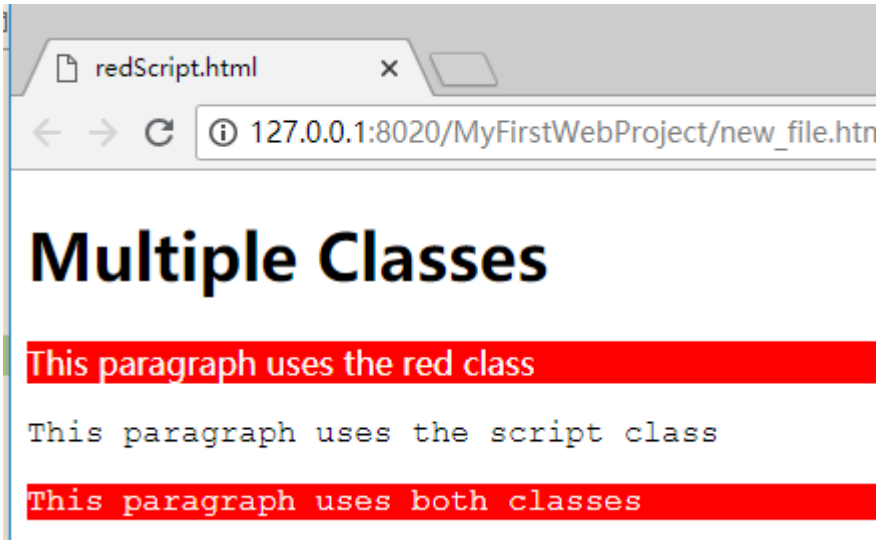


Figure 3-26 combining class style

The paragraphs in Figure 3-26 appear to be in three different styles, but only red and script are defined. The third paragraph uses both classes. Here's the code:

Example 3-22 combining class style

```
<!DOCTYPE html>
<html>
<head>
<meta charset = "UTF-8">
<title>redScript.html</title>
<style type = "text/css">
.red {
color: white;
background-color: red;
}
.script {
font-family: courier;
}
</style>
</head>
<body>
<h1>Multiple Classes</h1>
<p class = "red">
This paragraph uses the red class
</p>
<p class = "script">
This paragraph uses the script class
</p>
```

```
<p class = "red script">
This paragraph uses both classes
</p>
</body>
</html>
```

The style sheet introduces two classes. The red class makes the paragraph red (well, white text with a red background), and the script class applies a courier font to the element. The first two paragraphs each have a class, and the classes act as you'd expect. The interesting part is the third paragraph because it has two classes.

<p class = "red script">

This assigns both the red and script classes to the paragraph. Both styles will be applied to the element in the order they are written. Note that both class names occur inside quotes and no commas are needed (or allowed). You can apply more than two classes to an element if you wish. If the classes have conflicting rules (say one makes the element green and the next makes it blue), the latest class in the list will overwrite earlier values.

An element can also have an ID. The ID style, the element style, and all the class styles are taken into account when the browser tries to display the object.

Normally, I don't like to use colors or other specific formatting instructions as class names. Usually, it's best to name classes based on their meaning (like mainBackgroundColor). You might decide that green is better than red, so you either have to change the class name or you have to have a red class that colored things green. That'd be weird.

3.4.5 Introducing div and span

So far, I've applied CSS styles primarily to paragraphs (with the <p> tag), but you can really use any element you want. In fact, you may want to invent your own elements. Perhaps you want a particular style, but it's not quite a paragraph. Maybe you want a particular style inside a paragraph. HTML has two very useful elements that are designed as generic elements. They don't have any predefined meaning, so they're ideal candidates for modification with the id and class attributes.

- ✓ **div**: A block-level element (like the p element). It acts just like a paragraph. A div usually has carriage returns before and after it. Generally, you use div to group a series of paragraphs.
- ✓ **span**: An inline element. It doesn't usually cause carriage returns because it's meant to be embedded into some other block-level element (usually a paragraph or a div). Usually, a span is used to add some type of special formatting to an element that's contained inside a block-level element.

To see why div and span are useful, take a look at Figure 3-27

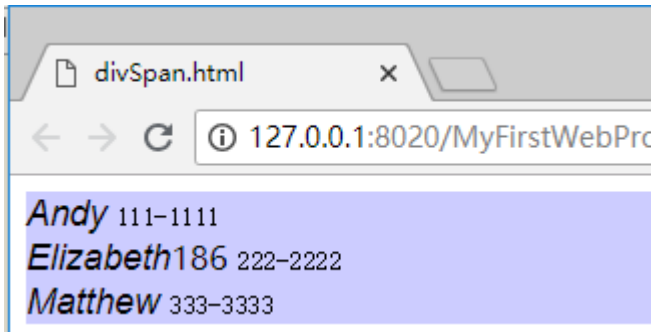


Figure 3-27 formatting text with div and span

The formatting of the page isn't complete (it will be described in the remainder of CSS), but some formatting is in place. Each name and phone number pair is clearly a group of things. Names and phone numbers are formatted differently. The interesting thing about this page is the code:

Example 3-23 formatting text with div and span

```
<!DOCTYPE html>
<html >
<head>
<meta charset = "UTF-8">
<title>divSpan </title>
<style type = "text/css">
.contact {
background-color: #CCCCFF;
}
.name {
font: italic 110% arial, sans-serif;
}
.phone {
font: 100% monospace;
}
</style>
</head>
<body>
<div class = "contact">
<span class = "name">Andy</span>
<span class = "phone">111-1111</span>
</div>
<div class = "contact">
<span class = "name">Elizabeth</span>186
<span class = "phone">222-2222</span>
</div>
<div class = "contact">
<span class = "name">Matthew</span>
<span class = "phone">333-3333</span>
```

```
</div>
</body>
</html>
```

What's exciting about this code is its clarity. When you look at the HTML, it's very clear what type of data you're talking about because the structure describes the data. Each div represents a contact. A contact has a name and a phone number.

The HTML doesn't specify how the data displays, just what it means. That is, the content and display of the page is separated.

3.4.6 Style Links

Now that you have some style going in your web pages, you may be a bit concerned about how ugly links are. The default link styles are useful, but they may not fit with your color scheme.

Adding a style to a link is easy. After all, <a> (the tag that defines links) is just an HTML tag, and you can add a style to any tag. Here's an example, where I make my links black with a yellow background:

```
a {
color: black;
background-color: yellow;
}
```

That works fine, but links are a little more complex than some other elements. Links actually have three states:

- ✓ **Normal**: This is the standard state. With no CSS added, most browsers display unvisited links as blue underlined text.
- ✓ **Visited**: This state is enabled when the user visits a link and returns to the current page. Most browsers use a purple underlined style to indicate that a link has been visited.
- ✓ **Hover**: The hover state is enabled when the user's mouse is lingering over the element. Most browsers don't use the hover state in their default settings.

If you apply a style to the <a> tags in a page, the style is applied to all the states of all the anchors.

You can apply a different style to each state, as illustrated by Figure 3-28.

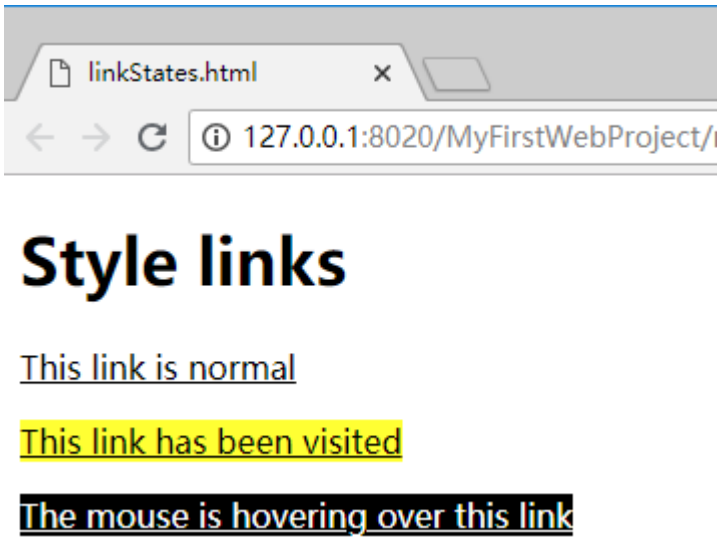


Figure 3-28 style links

In this example, I make ordinary links black on a white background. A visited link is black on yellow; and, if the mouse is hovering over a link, the link is white with a black background.

Take a look at the code and see how it's done:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>linkStates.html</title>
        <style type="text/css">
            a {
                color: black;
                background-color: white;
            }
            a:visited {
                color: black;
                background-color: #FFFF33;
            }
            a:hover {
                color: white;
                background-color: black;
            }
        </style>
    </head>
    <body>
        <h1>Style links</h1>
```

```
      <p> <a href="http://www.wzu.edu.cn">This link is normal</a>
   </p>
      <p> <a href="http://sdxy.wzu.edu.cn/">This link has been
visited</a></p>
      <p> <a href="http://cic.wzu.edu.cn/">The mouse is hovering
over this link</a> </p>
   </body>
</html>
```

Nothing is special about the links in the HTML part of the code. The links change their state dynamically while the user interacts with the page. The style sheet determines what happens in the various states. Here's how you approach putting the code together:

- ✓ Determine the ordinary link style first by making a style for the <a> tag. If you don't define any other styles, all links will follow the ordinary link style.
- ✓ Make a style for visited links. A link will use this style after that site is visited during the current browser session. The a:visited selector indicates links that have been visited.
- ✓ Make a style for hovered links. The a:hover style is applied to the link only when the mouse is hovering over the link. As soon as the mouse leaves the link, the style reverts to the standard or visited style, as appropriate.

Link styles have some special characteristics. You need to be a little bit careful how you apply styles to links. Consider the following issues when applying styles to links:

- ✓ **The order is important.** Be sure to define the ordinary anchor first. The styles are based on the standard anchor style.
- ✓ **Make sure they still look like links.** It's important that users know something is intended to be a link. If you take away the underlining and the color that normally indicates a link, your users might be confused. Generally, you can change colors without trouble, but links should be either underlined text or something that clearly looks like a button.
- ✓ **Test visited links.** Testing visited links is a little tricky because, after you visit a link, it stays visited. Most browsers allow you to delete the browser history, which should also clear the link states to unvisited.
- ✓ **Don't change font size in a hover state.** Unlike most styles, hover changes the page in real time. A hover style with a different font size than the ordinary link can cause problems. The page is automatically reformatted to accept the larger (or smaller) font, which can move a large amount of text on the screen rapidly. This can be frustrating and disconcerting for users. It's safest to change colors or borders on hover but not the font family or font size.

3.4.7 Nested Selection

CSS allows some other nice selection tricks. Take a look at Figure 3-29 and you see a page with two kinds of paragraphs in it.

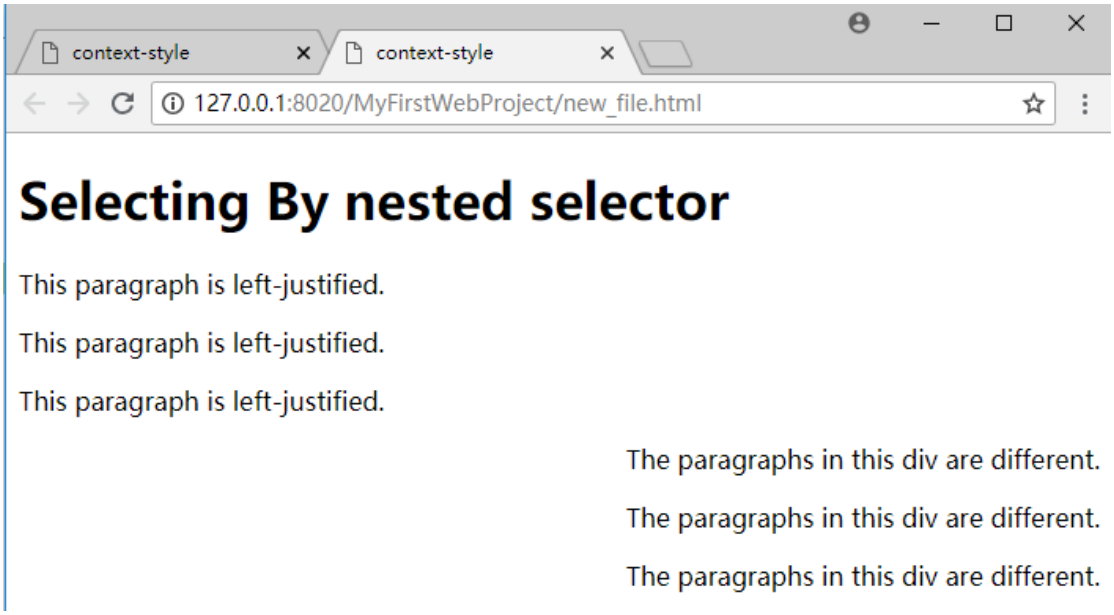


Figure 3-29 a simple example of nested selection

The code for the page is deceptively simple:

Example 3-24 a simple example of nested selection

```
<!DOCTYPE html>
<html >
<head>
<meta charset = "UTF-8">
<title>context-style</title>
<style type = "text/css">
#special p {
text-align: right;
}
</style>
</head>
<body>
<h1>Selecting By nested selector</h1>
<div>
<p>This paragraph is left-justified.</p>
<p>This paragraph is left-justified.</p>
<p>This paragraph is left-justified.</p>
</div>
<div id = "special">
<p>The paragraphs in this div are different.</p>
<p>The paragraphs in this div are different.</p>
<p>The paragraphs in this div are different.</p>
</div>
</body>
```

```
</html>
```

If you look at the code in Example 3-24, you see some interesting things:

- ✓ **The page has two divs.** One div is anonymous, and the other is special.
- ✓ **None of the paragraphs has an ID or class**. The paragraphs in this page don't have names or classes defined, yet they clearly have two different types of behavior. The first three paragraphs are aligned to the left, and the last three are aligned to the right.
- ✓ **The style rule affects paragraphs inside the special div**. Take another look at the style:

  #special p {

  This style rule means, "Apply this style to any paragraph appearing inside something called special." You can also define a rule that could apply to an image inside a list item or emphasized items inside a particular class. When you include a list of style selectors without commas, you're indicating a nested style.
- ✓ **Paragraphs defined outside special aren't affected.** This nested selection technique can help you create very complex style combinations. It becomes especially handy when you start building positioned elements, like menus and columns.

3.4.8 Defining Styles for Multiple Elements

Sometimes, you want a number of elements to share similar styles. As an example, look at Figure 3-30.

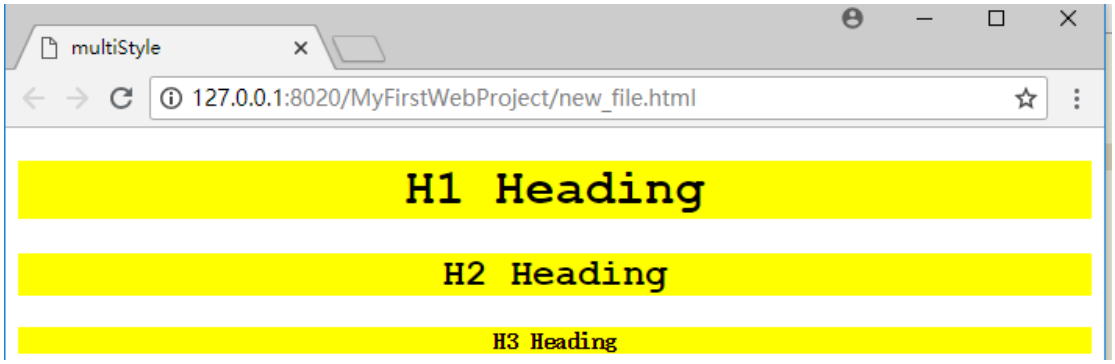


Figure 3-30 style for multiple elements

Here is the code.

Example 3-25 style for multiple elements

```
<!DOCTYPE html>
<html>
<head>
<meta charset = "UTF-8">
```

```
<title>multiStyle</title>
<style type = "text/css">
h1, h2, h3 {
text-align: center;
font-family: "bradley hand itc", courier;
background-color: yellow;
}
h3 {
font-family: monospace;
}
</style>
</head>
<body>
<h1>H1 Heading</h1>
<h2>H2 Heading</h2>
<h3>H3 Heading</h3>
</body>
</html>
```

One style element (the one that begins h1, h2, h3) provides all the information for all three heading types. If you include more than one element in a style selector separated by commas, the style applies to all the elements in the list. In this example, the centered courier font with a yellow background is applied to heading levels 1, 2, and 3 all in the same style.

If you want to make modifications, you can do so. I created a second h3 rule, changing the font-family attribute to monospace. Style rules are applied in order, so you can always start with the general rule and then modify specific elements later in the style if you wish.

If you have multiple elements in a selector rule, it makes a huge difference whether you use commas. If you separate elements with spaces (but no commas), CSS looks for an element nested within another element. If you include commas, CSS applies the rule to all the listed elements.

It's possible to get even more specific about selectors with punctuation. For example, the + selector describes sibling relationship. For example, look at the following rule:

h1+p

This targets only the paragraph that immediately follows a level-one headline. All other paragraphs will be ignored. There are other selectors as well, but the ones mentioned here will suffice for most applications.

CSS3 supports several new selectors with interesting new capabilities, and it is not included in this book.

3.5 Borders and Background

CSS offers some great features for making your elements more colorful, including a flexible and powerful system for adding borders to your elements. You can also add background images to all or part of your page. This section describes how to use borders and backgrounds for maximum effect.

3.5.1 Using the Border Attributes

You can use CSS to draw borders around any HTML element. You have some freedom in the border size, style, and color. Here are two ways to define border properties: using individual border attributes, and using a shortcut. Borders don't actually change the layout, but they do add visual separation that can be appealing, especially when your layouts are more complex.

Figure 3-31 illustrates a page with a simple border drawn around the heading.

Figure 3-31 heading with double border

The code is shown in Example 3-26.

Example 3-26 heading with double border

```
<!DOCTYPE html>
<html lang = "en-US">
<head>
<meta charset = "UTF-8">
<title>borderProps</title>
<style type = "text/css">
h1 {
border-color: red;
border-width: .25em;
border-style: double;
}
</style>
</head>
<body>
<h1>This has a border</h1>
```

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

Each element can have a border defined. Borders require three attributes:

- ✓ **width**: The width of the border. This can be measured in any CSS unit, but border width is normally described in pixels (px) or ems. (Remember: An em is roughly the width of the capital letter "M" in the current font.)
- ✓ **color**: The color used to display the border. The color can be defined like any other color in CSS, with color names or hex values.
- ✓ **style**: CSS supports a number of border styles. For the example, in the example, I chose a double border. This draws a border with two thinner lines around the element.

You must define all three attributes if you want borders to appear properly. You can't rely on the default values to work in all browsers.

Define border styles

CSS has a predetermined list of border styles you can choose from. Figure 3-32 shows a page with all the primary border styles displayed.

Figure 3-32 border styles

Here is the code:

Example 3-27 border styles

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="UTF-8">
      <title>BorderStyle</title>
      <style type="text/css">
         .solid{
            border-color: red;
            border-width: .25em;
            border-style:solid;
         }
         .double
         {
            border-color: red;
            border-width: .25em;
            border-style:double;
         }
         .groove{
            border-color: red;
            border-width: .25em;
            border-style:groove;
         }
         .ridge{
            border-color: red;
            border-width: .25em;
            border-style:ridge;
         }
         .inset{
            border-color: red;
            border-width: .25em;
            border-style:inset;
         }
         .outset{
            border-color: red;
            border-width: .25em;
            border-style:outset;
         }
         .dashed{
            border-color: red;
            border-width: .25em;
            border-style:dashed;
         }
```

```
            .dotted{
            border-color: red;
            border-width: .25em;
            border-style:dotted;
         }
      </style>
   </head>
   <body>
      <h1>Border styles</h1>
      <ul>
         <li class="solid">solid</li><br />
         <li class="double">double</li><br />
         <li class="groove">groove</li><br />
         <li class="ridge">ridge</li><br />
         <li class="inset">inset</li><br />
         <li class="outset">outset</li><br />
         <li class="dashed">dashed</li><br />
         <li class="dotted">dotted</li><br />
      </ul>
   </body>
</html>
```

You can choose any of these styles for any border:

- ✓ Solid: A single solid line around the element.
- ✓ Double: Two lines around the element with a gap between them. The border width is the combined width of both lines and the gap.
- ✓ Groove: Uses shading to simulate a groove etched in the page.
- ✓ Ridge: Uses shading to simulate a ridge drawn on the page.
- ✓ Inset: Uses shading to simulate a pressed-in button.
- ✓ Outset: Uses shading to simulate a button sticking out from the page.
- ✓ Dashed: A dashed line around the element.
- ✓ Dotted: A dotted line around the element.

Shades of danger

Several border styles rely on shading to produce special effects. Here are a couple things to keep in mind when using these shaded styles:

- ✓ **You need a wide border.** The shading effects are typically difficult to see if the border is very thin.

- ✓ **Browsers shade differently.** All the shading tricks modify the base color (the color you indicate with the border-color attribute) to simulate depth. Unfortunately, the browsers don't all do this in the same way. For now, avoid shaded styles if this bothers you.
- ✓ **Black shading doesn't work in IE**. IE makes colors darker to get shading effects. If your base color is black, IE can't make anything darker, so you don't see the shading effects at all. Likewise, white shading doesn't work well on Firefox.

Using the border shortcut

Defining three different CSS attributes for each border is a bit tedious. Fortunately, CSS includes a handy border shortcut that makes borders a lot easier to define, as Figure 3-33 demonstrates.

Figure 3-33 border style shortcut

You can't tell the difference from the output, but the code for Example 3-28 is extremely simple:

Example 3-28 border style shortcut

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>borderShortcut</title>
        <style type="text/css">
            h1 {
                border: red 5px solid;
            }
        </style>
    </head>
    <body>
        <h1>This page uses the border shortcut</h1>
    </body>
</html>
```

The order in which you describe border attributes doesn't matter. Just specify a color, a size, and a border style.

Creating partial borders

If you want, you can have more precise control of each side of a border. There are a number of specialized border shortcuts for each of the sub-borders. Figure 3-34 shows how you can add borders

to the top, bottom, or sides of your element.

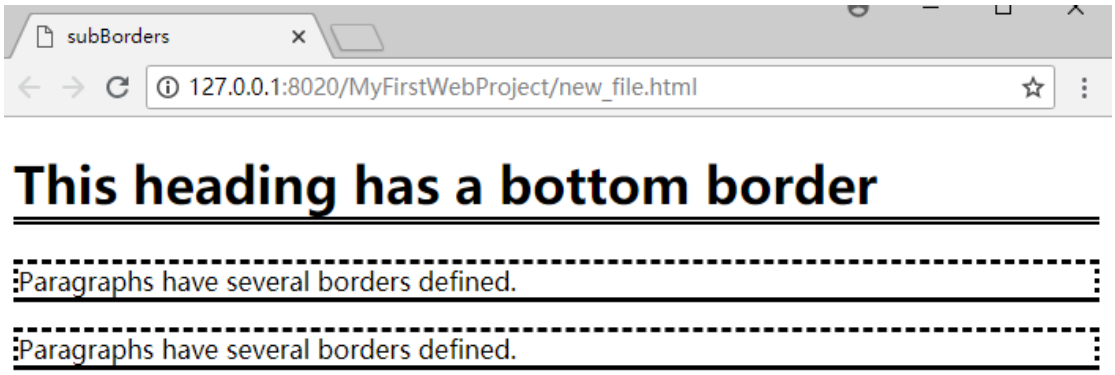


Figure 3-34 specify your border style

Figure 3-34 applies a border style to the bottom of the heading as well as different borders above, below, and to the sides of the paragraphs. Partial borders are pretty easy to build, as you can see from the code listing:

Example 3-29 specify your border style

```
<!DOCTYPE html>
<html >
    <head>
        <meta charset="UTF-8">
        <title>subBorders</title>
        <style type="text/css">
            h1 {
                border-bottom: 5px black double;
            }
            p {
                border-left: 3px black dotted;
                border-right: 3px black dotted;
                border-top: 3px black dashed;
                border-bottom: 3px black groove;
            }
        </style>
    </head>
    <body>
        <h1>This heading has a bottom border</h1>
        <p>
            Paragraphs have several borders defined.
        </p>
        <p>
            Paragraphs have several borders defined.
        </p>
```

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

Notice the border styles. CSS has style rules for each side of the border: border-top, border-bottom, border-left, and border-right. Each of these styles acts like the border shortcut, but it only acts on one side of the border.

3.5.2 Introducing the Box Model

XHTML and CSS use a specific type of formatting called the box model. Understanding how this layout technique works is important. If you don't understand some of the nuances, you'll be surprised by the way your pages flow.

The box model relies on two types of elements: inline and block-level. Block level elements include tags, paragraphs, and all headings (h1– h6), whereas **strong**, **a**, and **image** are examples of inline elements.

The main difference between inline and block-level elements is this: Block-level elements always describe their own space on the screen, whereas inline elements are allowed only within the context of a block level element.

Your overall page is defined in block-level elements, which contain inline elements for detail. Each block-level element (at least in the default setting) takes up the entire width of the parent element. The next block-level element goes directly underneath the last element defined.

Inline elements flow differently. They tend to go immediately to the right of the previous element. If there's no room left on the current line, an inline element drops down to the next line and goes to the far left.

Border, margin, and padding

Each block-level element has several layers of space around it, such as

- ✓ Padding: The space between the content and the border.
- ✓ Border: Goes around the padding.
- ✓ Margin: Space outside the border between the border and the parent element.

Figure 3-35 shows the relationship among margin, padding, and border

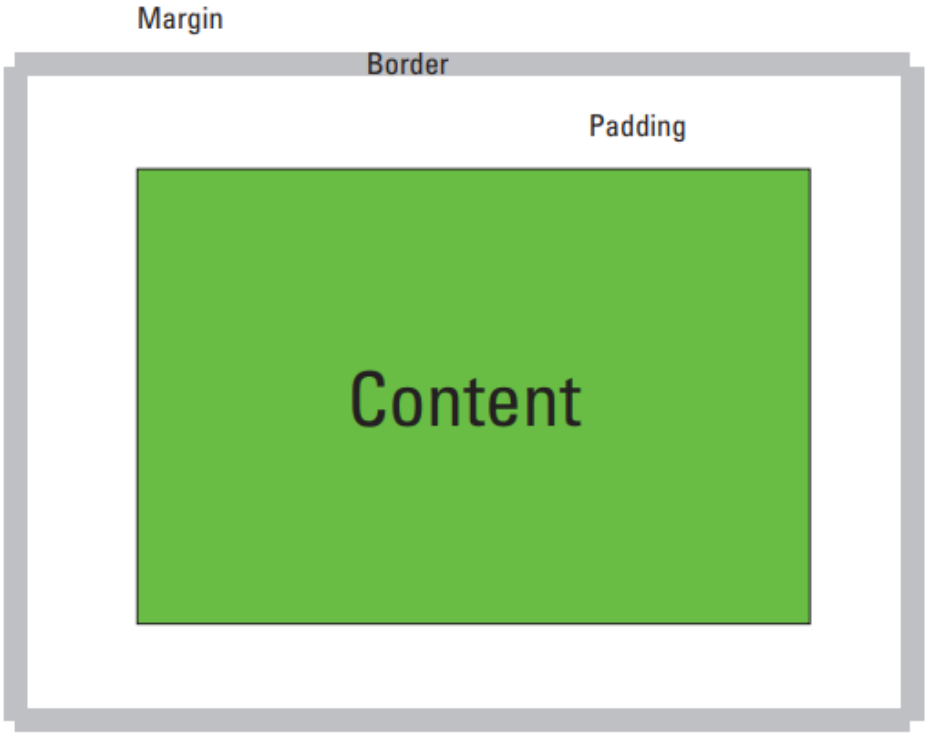


Figure 3-35 relationship among margin, padding, and border

You can change settings for the margin, border, and padding to adjust the space around your elements. The **margin** and **padding** CSS rules are used to set the sizes of these elements, as shown in Figure 3-36.

Figure 3-36 margins and padding

I applied different combinations of margin and padding to a series of paragraphs. To make things easier to visualize, I drew a border around the containing all the paragraphs and each individual paragraph element. You can see how the spacing is affected.

Example 3-30 margins and padding

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="UTF-8">
      <title>marginPadding</title>
      <style type="text/css">
         div {
            border: red 5px solid;
         }
         p {
            border: black 2px solid;
         }
         #margin {
            margin: 5px;
         }
         #padding {
            padding: 5px;
         }
         #both {
            margin: 5px;
            padding: 5px;
         }
      </style>
   </head>
   <body>
      <h1>Margins and padding</h1>
      <div id="main">
         <p>This paragraph has the default margins and padding</p>
         <p  id="margin">This  paragraph  has  a  margin  but  no
padding</p>
         <p  id="padding">This  paragraph  has  padding  but  no
margin</p>
         <p id="both">This paragraph has a margin and padding</p>
      </div>
   </body>
</html>
```

You can determine margin and padding using any of the standard CSS measurement units, but the most common are pixels and ems.

Positioning elements with margins and padding

As with borders, you can use variations of the margin and padding rules to affect spacing on a particular side of the element. One particularly important form of this trick is centering.

In old-style HTML, you could center any element or text with the tag. This was pretty easy, but it violated the principle of separating content from style. The text-align: center rule is a nice alternative, but it only works on the contents of an element. If you want to center an entire block-level element, you need another trick, as you can see in Figure 3-37.

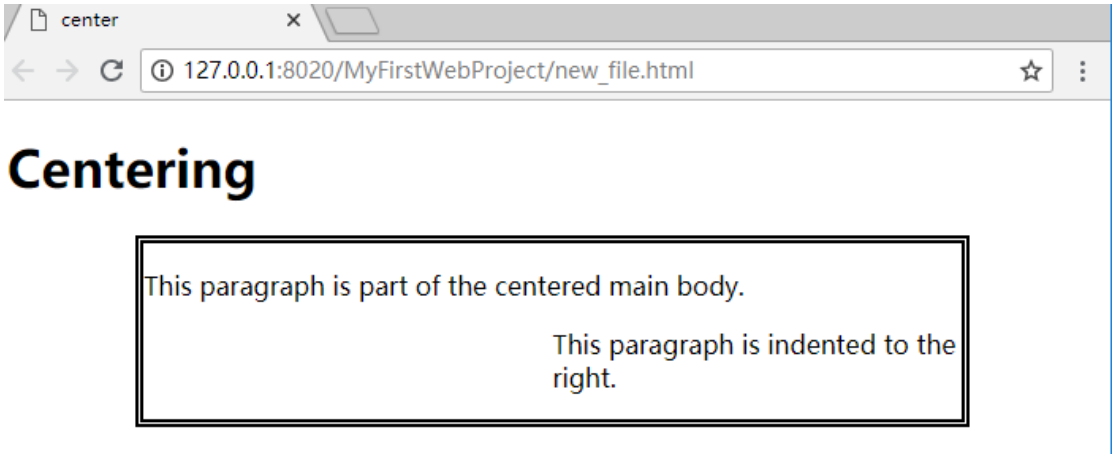


Figure 3-37 adjust the position

Here is the code:

Example 3-31 adjust the position

```
<!DOCTYPE html>
<html >
    <head>
        <meta charset="UTF-8">
        <title>center</title>
        <style type="text/css">
            #mainBody {
                border: 5px double black;
                width: 75%;
                margin-left: auto;
                margin-right: auto;
            }
            .indented {
                margin-left: 50%;
            }
        </style>
    </head>
    <body>
        <h1>Centering</h1>
        <div id="mainBody">
            <p>This paragraph is part of the centered main body.
            </p>
            <p class="indented">
```

```
            This paragraph is indented to the right.
        </p>
    </div>
</body>
</html>
```

You can adjust the width of a block. The main div that contains all the paragraphs has its width set to 75 percent of the page body width.

- ✓ Center an element by setting margin-left and margin-right to auto. Set both the left and right margins to auto to make an element center inside its parent element. This trick is most frequently used to center divs and tables.
- ✓ Use margin-left to indent an entire paragraph. You can use margin-left or margin-right to give extra space between the border and the contents.
- ✓ Percentages refer to percent of the parent element. When you use percentages as the unit measurement for margins and padding, you're referring to the percentage of the parent element; so a margin- left of 50 percent leaves the left half of the element blank.
- ✓ Borders help you see what's happening. I added a border to the main Body div to help you see that the div is centered.
- ✓ Setting the margins to auto doesn't center the text. It centers the div (or other block-level element). Use text-align: center to center text inside the div.

3.5.3 New CSS3 Border Techniques

Borders have been a part of CSS from the beginning, but CSS3 adds some really exciting new options. Modern browsers now support borders made from an image as well as rounded corners and box shadows. These techniques promise to add exciting new capabilities to your designs.

Adding Rounded Corners

Older CSS was known for being very rectangular, so web designers tried to soften their designs by adding rounded corners. This was a difficult effect to achieve. CSS3 greatly simplifies the creation of rounded corners with the border-radius rule.

Figure 3-38 demonstrates a simple page with a rounded border.

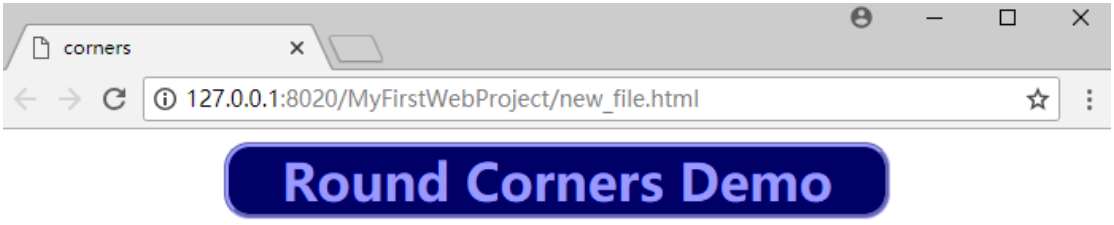


Figure 3-38 corners

It’s pretty easy to get rounded corners on those browsers that support the tag:

```
<!DOCTYPE HTML>
<html>
   <head>
      <title>corners</title>
      <meta charset="UTF-8" />
      <style type="text/css">
         h1 {
            width: 60%;
            background-color: #000066;
            color: #9999ff;
            border: #9999ff 3px groove;
            margin: auto;
            text-align: center;
            border-radius: .5em;
         }
      </style>
   </head>
   <body>
      <h1>Round Corners Demo</h1>
   </body>
</html>
```

The border-radius rule works by cutting an arc from each corner of the element. The arc has the specified radius, so for sharp corners, you’ll want a small radius. You can measure the radius in any of the common measurements, but pixels (px) and character width (em) are the most commonly used.

The border is not visible unless the element has the background-color or border defined. Note that there are variations of each tag to support specific corners: border-top-left-radius and so on. This can be useful if you do not wish to apply the same radius to all four corners of your element. The most recent browsers now support the generic border-radius rule. You can pick up a number of the previous-generation browsers by using the vendor-specific prefix. If your browser does not understand the border radius rule, it will simply create the ordinary squared corners.

Adding a box shadow

Box shadows are often added to elements to create the illusion of depth.

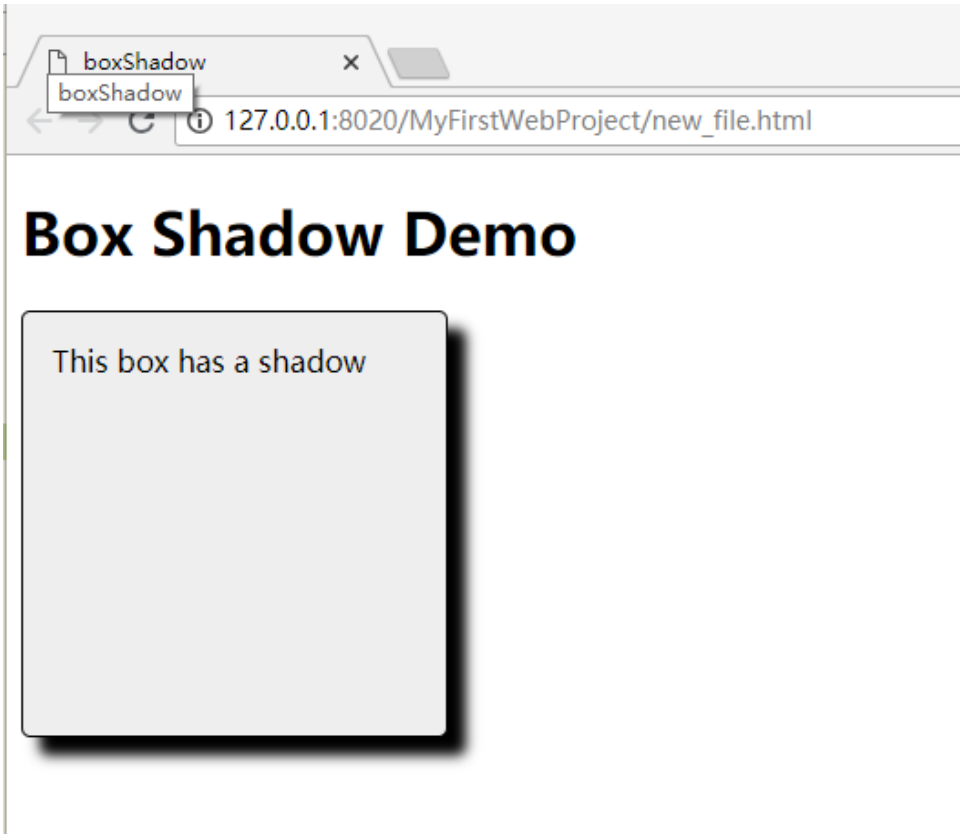


Figure 3-39 box shadow

Figure 3-39 displays a page with a simple box shadow. The box shadow effect is not difficult to achieve, but it is normally done as part of a class definition so it can be re-used throughout the page. Here's some sample code:

Example 3-32 box shadow

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>boxShadow</title>
        <meta charset="UTF-8" />
        <style type="text/css">
            .shadow {
                box-shadow: 10px 10px 10px #000000;
                height: 200px;
                width: 200px;
                padding: 1em;
                border: 1px solid black;
                border-radius: 5px;
                background-color: #EEEEEE;
            }
        </style>
    </head>
```

```
    <body>
        <h1>Box Shadow Demo</h1>
        <div class="shadow">
            This box has a shadow
        </div>
    </body>
</html>
```

Adding a box shadow is much easier in CSS3 than it once was. Here are the steps:

Step 1. Define a class.

Often you'll want to apply the same settings to a number of elements on a page, so the box shadow is often combined with other elements like background-color and border in a CSS class that can be reused throughout the page.

Step 2. Add the box-shadow rule.

The latest browsers support the standard box-shadow rule, but you may also want to include browser prefixes to accommodate older browses.

Step 3. Specify the offset.

A shadow is typically offset from the rectangle it belongs to. The first two values indicate the horizontal and vertical offset. Measure using any of the standard CSS measurements (normally pixels or ems).

Step 4. Determine the blur and spread distances.

You can further modify the behavior of the shadow by specifying how quickly the shadow blurs and how far it spreads. These are optional parameters.

Step 5. Indicate the shadow color.

You can make the shadow any color you wish. Black and gray are common, but you can get interesting effects by picking other colors. Many other shadow effects are possible. You can add multiple shadows, and you can also use the inset keyword to produce an interior shadow to make it look like part of the page is cut out.

There is a similar rule called text-shadow. It has the same general behavior as box-shadow, but it's designed to work on text. It's possible to get some really nice effects with this tool, but be careful not to impede readability.

Changing the Background Image

You can use the **img** tag to add an image to your page, but sometimes you want to use images as a background for a specific element or for the entire page. You can the background-image CSS rule to apply a background image to a page or elements on a page. Figure 3-40 shows a page with this feature.

Figure 3-40 background image

Background images are easy to apply. The code for Figure 3-40 shows how:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>backgroundImage</title>
        <style type="text/css">
            body {
                background-image: url("clouds.png");
                background-repeat:no-repeat;
                background-attachment: fixed;
                background-size:cover;
            }
            h1 {
                background-image: url("star.png");
            }
            p {
                background-color: white;
```

```
            background-color: rgba(255, 255, 255, .85);
         }
      </style>
   </head>
   <body>
      <h1>Using Background Images</h1>
      <p>
         The heading uses a lighter version of the background, and
the paragraph uses a solid color background with light transparency.
      </p>
      <p>
         The heading uses a lighter version of the background, and
the paragraph uses a solid color background with light transparency.
      </p>
   </body>
</html>
```

Attaching the background image to an element through CSS isn't difficult.

Here are the general steps:

- ✓ Step 1. Find or create an appropriate image and place it in the same directory as the page so it's easy to find.
- ✓ Step 2. Attach the background-image style rule to the page you want to apply the image to. If you want to apply the image to the entire page, use the body element.
- ✓ Step 3. Tell CSS where background-image is by adding a url identifier. Use the keyword url() to indicate that the next thing is an address.
- ✓ Step 4. Enter the address of the image. It's easiest if the image is in the same directory as the page. If that's the case, you can simply type the image name. Make sure you surround the URL with quotes.
- ✓ Step 5. Test your background image by viewing the web page in your browser. A lot can go wrong with background images. The image may not be in the right directory, you might have misspelled its name, or you may have forgotten the url() bit. (I do all those things sometimes.)

With this element, I kept the ropes image, but I made a much brighter background so the dark text would show up well underneath. This technique allows you to use the background image even underneath text, but here are a few things to keep in mind if you use it:

- ✓ Make the image very dark or very light. Use the Adjust Colors command in your favorite image editor to make your image dark or light. Don't be shy. If you're creating a lighter version, make it very light.
- ✓ Set the foreground to a color that contrasts with the background. If you have a very light version of the background image, you can use dark text on it. A dark background requires light text. Adjust the text color with your CSS code.

- ✓ Set a background color. Make the background color representative of the image. Background images can take some time to appear, but the background color appears immediately because it is defined in CSS. This is especially important for light text because white text on the default white background is invisible. After the background image appears, it overrides the background color. Be sure the text color contrasts with the background, whether that background is an image or a solid color.
- ✓ Use this method for large text. Headlines are usually larger than body text, and they can be easier to read, even if they have a background behind them. Try to avoid putting background images behind smaller body text. This can make the text much harder to read.

3.6 Advanced CSS: Page Layout in CSS

Once you've designed a few websites, you begin to notice a pattern. Most web pages have a similar structure, with a header, footer, navigation, and sections. The authors of the HTML5 specification added support for additional structural tags to make it easier for authors to create web pages with a tag structure that matches the meaning behind those tags. Most of the new tags in HTML5 are new structural elements.

3.6.1 A Short History of HTML Page Layout

On the early Web, most pages tended to be a single column of text that ran down the page from top to bottom, mainly because that was the only option available. Early browsers didn't even support tables, so you could create paragraphs, lists, and other basic elements that were laid out left to right and top to bottom.

Back in the prehistoric (well, pre-CSS) days, no good option was built into HTML for creating a layout that worked well. Clever web developers and designers found some ways to make things work, but these proposed solutions all had problems.

Problems with frames

Frames were a feature of the early versions of HTML. They allowed you to break a page into several segments. Each segment was filled with a different page from the server. You could change pages independently of each other, to make a very flexible system. You could also specify the width and height of each frame.

At first glance, frames sound like an ideal solution to layout problems. In practice, they had a lot of disadvantages, such as

- ✓ **Complexity**: If you had a master page with four segments, you had to keep track of five web pages. A master page kept track of the relative positions of each section, but had no content of its own. Each of the other pages had content but no built-in awareness of the other pages.

- ✓ **Linking issues**: The default link action caused content to pop up in the same frame as the original link, which isn't usually what you want. Often, you'd put a menu in one frame and have the results of that menu pop up in another frame. This meant most anchors had to be modified to make them act properly.
- ✓ **Backup nightmares**: If the user navigated to a page with frames and then caused one of the frames to change, what should the backup button do? Should it return to the previous state (with only the one segment returned to its previous state) or was the user's intent to move entirely off the master page to what came before? There are good arguments for either and no good way to determine the user's intention. Nobody ever came up with a reasonable compromise for this problem.
- ✓ **Ugliness**: Although it's possible to make frames harder to see, they did become obvious when the user changed the screen size and scroll bars would automatically pop up.
- ✓ **Search engine problems:** Search engines had a lot of problems with frame-based pages. The search engine might only index part of a frame based site, and the visitor might get incomplete websites missing navigation or sidebars. For all these reasons, frames are no longer supported in HTML5.

The layout techniques you read about in this chapter more than compensate for the loss of frames as layout tools. HTML5 does allow one limited type of frame called the iFrame, but even it is not necessary.

Problems with tables

When it became clear that frames weren't the answer, web designers turned to tables. HTML has a flexible and powerful table tool, and it's possible to do all kinds of creative things with that tool to create layouts. A few web developers still do this, but you'll see that flow-based layout is cleaner and easier. Tables are meant for tabular data, not as a layout tool. When you use tables to set up the visual layout of your site, you'll encounter these problems:

- ✓ **Complexity**: Although table syntax isn't that difficult, a lot of nested tags are in a typical table definition. To get exactly the look you want, you probably won't use an ordinary table but tricks, like rowspan and colspan, special spacer images, and tables inside tables. It doesn't take long for the code to become bulky and confusing.
- ✓ **Content and display merging**: Using a table for layout violates the principle of separating content from display. If your content is buried inside a complicated mess of table tags, it'll be difficult to move and update.
- ✓ **Inflexibility**: If you create a table-based layout and then decide you don't like it, you basically have to redesign the entire page from scratch. It's no simple matter to move a menu from the left to the top in a table based design, for example. Tables are great for displaying tabular data. Avoid using them for layout because you have better tools available.

Problems with huge images

Some designers skip HTML altogether and create web pages as huge images. Tools, like Photoshop,

include features for creating links in a large image. Again, this seems ideal because a skilled artist can have control over exactly what is displayed. Like the other techniques, this has some major drawbacks, such as

- ✓ **Size and shape limitations**: When your page is based on a large image, you're committed to the size and shape of that image for your page. If a person wants to view your page on a cellphone or PDA, it's unlikely to work well, if at all.
- ✓ **Content issues**: If you create all the text in your graphic editor, it isn't really stored to the web page as text. In fact, the web page will have no text at all. This means that search engines can't index your page, and screen-readers for people with disabilities won't work.
- ✓ **Difficult updating**: If you find an error on your page, you have to modify the image, not just a piece of text. This makes updating your page more challenging than it would be with a plain HTML document.
- ✓ **File size issues**: An image large enough to fill a modern browser window will be extremely large and slow to download. Using this technique will all but eliminate users with slower access from using your site.

Problems with Flash

Another tool that's gained great popularity is the Flash animation tool from Adobe. This tool allows great flexibility in how you position things on a page and supports techniques that were once difficult or impossible in ordinary HTML, such as sound and video integration, automatic motion tweening, and path-based animation. Flash certainly had an important place in web development. Even though Flash has historic significance, you should avoid using it for ordinary web development for the following reasons:

- ✓ **Cost**: The Flash editor isn't cheap, and it doesn't look like it'll get cheaper. The tool is great, but if free or low-cost alternatives work just as well, it's hard to justify the cost.
- ✓ **Binary encoding**: All text in a Flash web page is stored in the Flash file itself. It's not visible to the browser. Flash pages (like image-based pages) don't work in web searches and aren't useful for people with screen-readers.
- ✓ **Updating issues:** If you need to change your Flash-based page, you need the Flash editor installed. This can make it more difficult to keep your page up-to-date.
- ✓ **No separation of content:** As far as the browser is concerned, there's no content but the Flash element, so there's absolutely no separation of content and layout. If you want to make a change, you have to change the Flash application.
- ✓ **Search engine problems**: Code written in Flash can't always be read by search engines (though Google is working on the problem).
- ✓ **Technical issues**: Flash is not integrated directly into the browser, which leads to a number of small complications. The Forward and Back buttons don't work as expected, printing can be problematic, and support is not universal.

- ✓ **Limited mobile access**: Flash is not supported on iPhones and iPads, and support is limited on other mobile platforms. As the mobile platform becomes more and more important, it's hard to justify working with a system that is not supported on these platforms.
- ✓ **It's no longer necessary**: HTML5, CSS3, and JavaScript have now addressed many of the shortcomings that once made Flash such a compelling alternative. You no longer need a plug-in to play audio and video, or to program games.

3.6.2 Introducing the Floating Layout Mechanism

CSS supplies a couple techniques for layout. The preferred technique for most applications is a floating layout. The basic idea of this technique is to leave the HTML layout as simple as possible, but to provide style hints that tell the various elements how to interact with each other on the screen.

In a floating layout, you don't legislate exactly where everything will go. Instead, you provide hints and let the browser manage things for you. This ensures flexibility because the browser will try to follow your intentions, no matter what size or shape the browser window becomes. If the user resizes the browser, the page will flex to fit to the new size and shape, if possible.

Floating layouts typically involve less code than other kinds of layouts because only a few elements need specialized CSS. In most of the other layout techniques, you need to provide CSS for every single element to make things work as you expect.

The most common place to use the float attribute is with images. Figure 3-41 has a paragraph with an image embedded inside

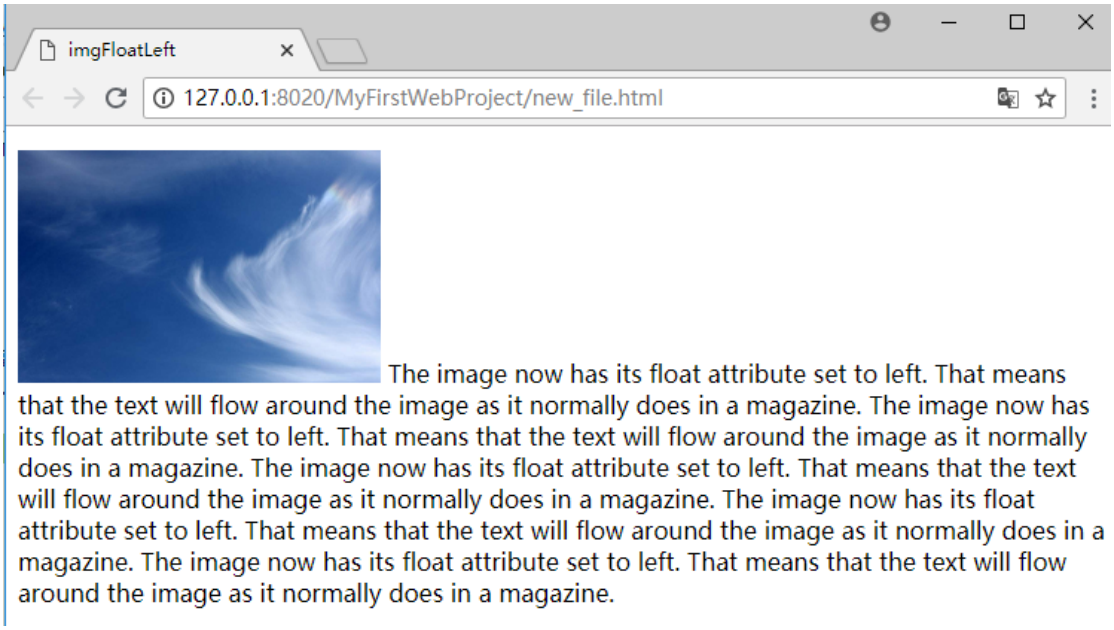


Figure 3-41 image and paragraphs

Here is the code:

Example 3-33 image and paragraph

```
<!DOCTYPE html>
<html>
<head>
<meta charset = "UTF-8">
<title>imgFloatLeft</title>
</head>
<body>
<p>
<img src = "clouds.PNG"
alt = "clouds" />
The image now has its float attribute set to left. That means
that the text will flow around the image as it normally does
in a magazine.
The image now has its float attribute set to left. That means
that the text will flow around the image as it normally does
in a magazine.
The image now has its float attribute set to left. That means
that the text will flow around the image as it normally does
in a magazine.
The image now has its float attribute set to left. That means
that the text will flow around the image as it normally does
in a magazine.
The image now has its float attribute set to left. That means
that the text will flow around the image as it normally does
in a magazine.
</p>
</body>
</html>
```

It's more likely that you want the image to take up the entire left part of the paragraph. The text should flow around the paragraph, similar to Figure 3-42 .

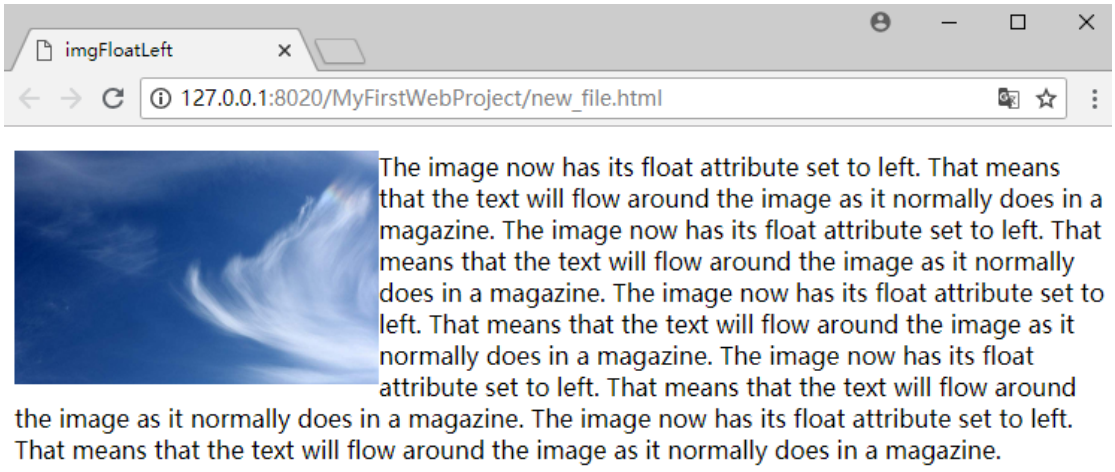


Figure 3-42 paragraph and float image

When you add a float:left attribute to the img element, the image tends to move to the left, pushing other content to the right. Now, the text flows around the image. The image is actually removed from the normal flow of the page layout, so the paragraph takes up all the space. Inside the paragraph, the text avoids overwriting the image.

Here is the style of the img.

Example 3-34 float attribute

```
<style type = "text/css">
img {
float: left;
}
</style>
```

3.6.3 Using Float with Block-Level Elements

The float attribute isn't only for images. You can also use it with any element (typically p or div) to create new layouts. Using the float attribute to set the page layout is easy after you understand how things really work.

Floating a paragraph

Paragraphs and other block-level elements have a well-defined default behavior. They take the entire width of the page, and the next element appears below. When you apply the float element to a paragraph, the behavior of that paragraph doesn't change much, but the behavior of **succeeding** paragraphs is altered.

To illustrate, I take you all the way through the process of building two side-by-side paragraphs.

Begin by looking at a page with three paragraphs. Paragraph 2 has its float property set to left. Figure 3-43 illustrates such a page.

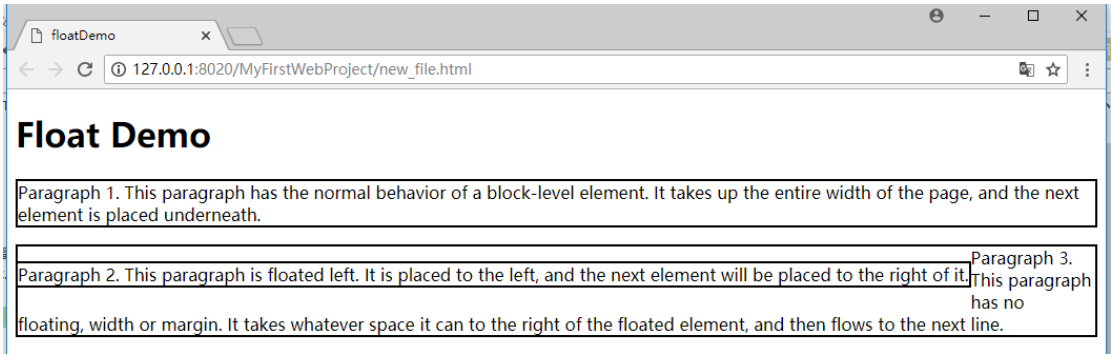


Figure 3-43 float paragraph

Here is the code.

Example 3-35 float paragraph

```
<!DOCTYPE html>
<html lang = "en-US">
<head>
<meta charset = "UTF-8">
<title>floatDemo</title>
<style type = "text/css">
p {
border: 2px black solid;
}
.floated {
float: left;
}
</style>
</head>
<body>
<h1>Float Demo</h1>
<p>
Paragraph 1.
This paragraph has the normal behavior of a block-level element.
It takes up the entire width of the page, and the next element is
placed underneath.
</p>
<p class = "floated">
Paragraph 2.
This paragraph is floated left. It is placed to the left, and the
next element will be placed to the right of it.
</p>
<p>
Paragraph 3.
This paragraph has no floating, width or margin. It takes whatever
space it can to the right of the floated element, and then flows
to the next line.
</p>
</body>
</html>
```

As you can see, some strange formatting is going on here. I improve on things later to make the beginnings of a two-column layout, but for now, just take a look at what's going on:

- ✓ I've added borders to the paragraphs. As you'll see, the width of an element isn't always obvious by looking at its contents. When I'm messing around with float, I often put temporary borders on key elements so I can see what's going on. You can always remove the borders when you have it working right.

- ✓ The first paragraph acts normally. The first paragraph has the same behavior you see in all block-style elements. It takes the entire width of the page, and the next element will be placed below it.
- ✓ The second paragraph is pretty normal. The second paragraph has its float attribute set to left. This means that the paragraph will be placed in its normal position, but that other text will be placed to the right of this element.
- ✓ The third paragraph seems skinny. The third paragraph seems to surround the second, but the text is pushed to the right. The float parameter in the previous paragraph causes this one to be placed in any remaining space (which currently isn't much). The remaining space is on the right and eventually underneath the second paragraph.

As you can see from the code, I have a simple class called floated with the float property set to left. The paragraphs are defined in the ordinary way. Even though paragraph 2 seems to be embedded inside paragraph 3 in the screen shot, the code clearly shows that this isn't the case. The two paragraphs are completely separate.

Adjusting the width

When you float an element, the behavior of succeeding elements is highly dependent on the width of the first element. This leads to a primary principle of float-based layout:

If you float an element, you must also define its width. The exception to this rule is elements with a predefined width, such as images and many form elements. These elements already have an implicit width, so you don't need to define width in the CSS. If in doubt, try setting the width at various values until you get the layout you're looking for.

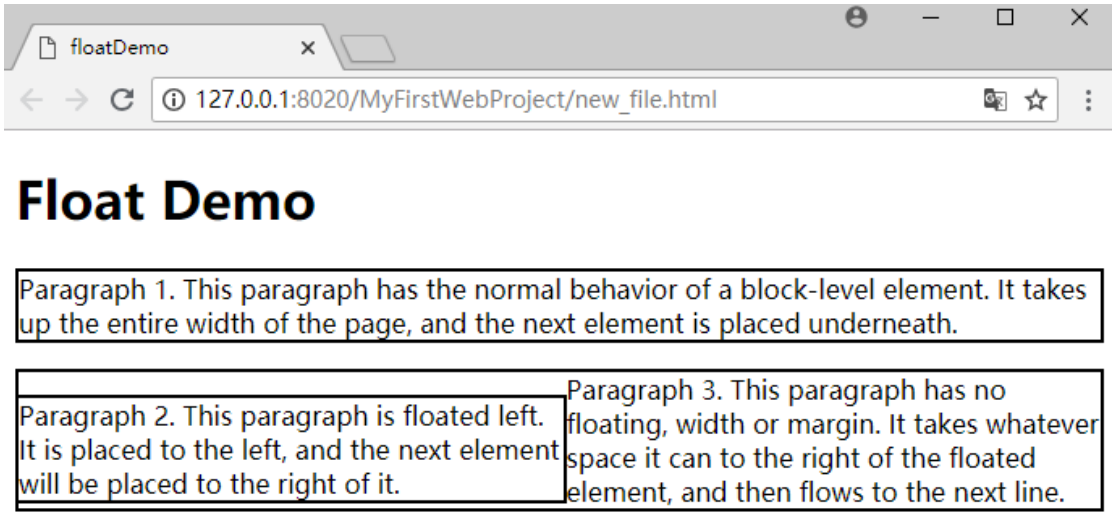


Figure 3-44 with 50% width

Figure 3-44 shows the page after I adjusted the width of the floated paragraph to 50 percent of the page width. Here is the styles:

Example 3-36 width style of float paragraph

```
<style type = "text/css">
p {
border: 2px black solid;
}
.floated {
float: left;
width: 50%;
}
</style>
```

Things look better in Figure 3-44, but paragraph 2 still seems to be embedded inside paragraph 3. The only significant change is in the CSS style.

I've added a width property to the floated element. Elements that have the float attribute enabled generally also have a width defined, except for images or other elements with an inherent width.

When you use a percentage value in the context of width, you're expressing a percentage of the parent element (in this case, the body because the paragraph is embedded in the document body). Setting the width to 50% means I want this paragraph to span half the width of the document body.

Setting the next margin

Things still don't look quite right. I added the borders around each paragraph so you can see an important characteristic of floating elements. Even though the text of paragraph 3 wraps to the right of paragraph 2, the actual paragraph element still extends all the way to the left side of the page. The element doesn't necessarily flow around the floated element, but its contents do. The background color and border of paragraph 3 still take as much space as they normally would if paragraph 2 didn't exist.

This is because a floated element is removed from the normal flow of the page. Paragraph 3 has access to the space once occupied by paragraph 2, but the text in paragraph 3 will try to find its own space without stepping on text from paragraph 2. Somehow, you need to tell paragraph 3 to move away from the paragraph 2 space. This isn't a difficult problem to solve after you recognize it. Figure 3-45 shows a solution.

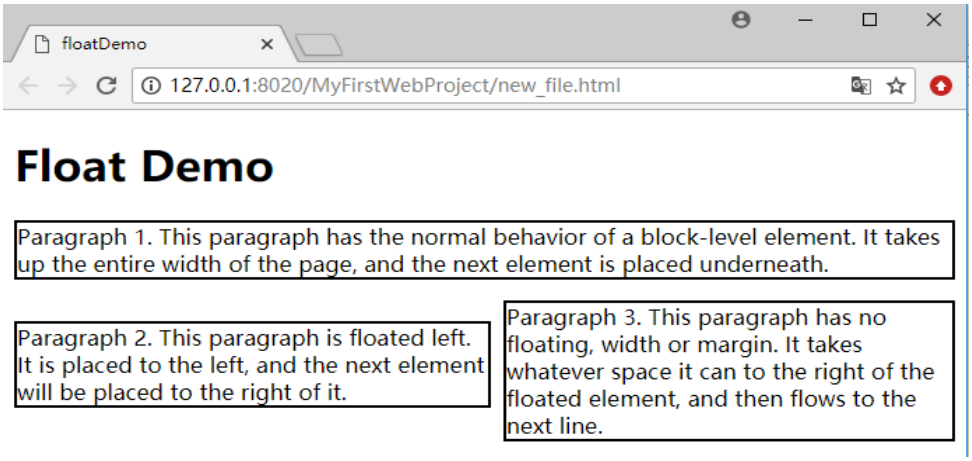


Figure 3-45 with margin left style

Here is the code.

Example 3-37 setting the margin left

```
<!DOCTYPE html>
<html lang = "en-US">
<head>
<meta charset = "UTF-8">
<title>floatDemo</title>
<style type = "text/css">
p {
border: 2px black solid;
}
.floated {
float: left;
width: 50%;
}
.right
{
margin-left: 52%;
}
</style>
</head>
<body>
<h1>Float Demo</h1>
<p>
Paragraph 1.
This paragraph has the normal behavior of a block-level element.
It takes up the entire width of the page, and the next element
is placed underneath.
</p>
<p class = "floated">
Paragraph 2.
This paragraph is floated left. It is placed to the left, and the
next element will be placed to the right of it.
</p>
<p class="right" >
Paragraph 3.
This paragraph has no floating, width or margin. It takes whatever
space it can to the right of the floated element, and then flows
to the next line.
</p>
</body>
</html>
```

3.6.4 Using Float to Style Forms

Many page layout problems appear to require tables. Some clever use of the CSS float can help elements with multiple columns without the overhead of tables. Forms cause a particular headache because a form often involves labels in a left column followed by input elements in the right column. You'd probably be tempted to put such a form in a table. Adding table tags makes the HTML much more complex and isn't required. It's much better to use CSS to manage the layout.

You can float elements to create attractive forms without requiring tables. Figure 3-46 shows a form with float used to line up the various elements. As page design gets more involved, it makes more sense to think of the HTML and the CSS separately. The HTML will give you a sense of the overall intent of the page, and the CSS can be modified separately. Using external CSS is a natural extension of this philosophy. Begin by looking at float form and concentrate on the HTML structure before worrying about style:

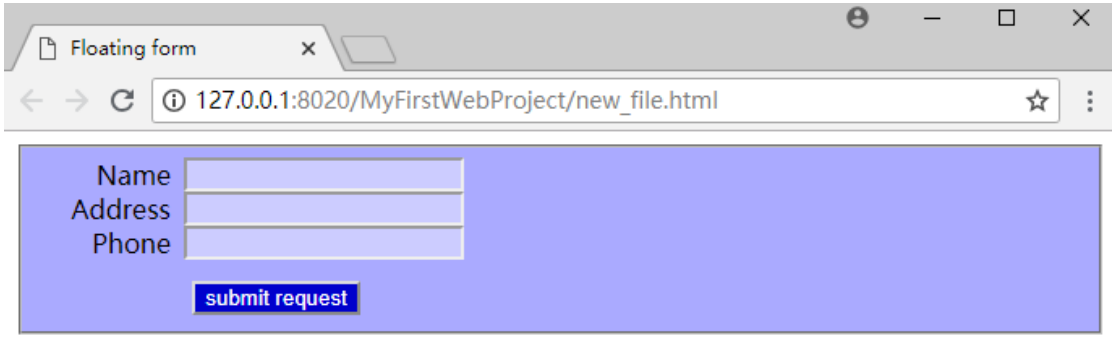


Figure 3-46 Floating form

Here is the HTML and CSS code.

Example 3-38 HTML code of the floating a from

```
<!DOCTYPE html>
<html>

  <head>
    <meta charset = "UTF-8">
    <title>Floating form</title>
    <link rel = "stylesheet"
          type = "text/css"
          href = "floatForm.css" />
  </head>
  <body>
    <form action = "">
      <fieldset>
        <label>Name</label>
        <input type = "text" id = "txtName" />
```

```
      <label>Address</label>
      <input type = "text"  id = "txtAddress" />
      <label>Phone</label>
      <input type = "text"  id = "txtPhone" />
      <button type = "button">
        submit request
      </button>
    </fieldset>
  </form>
 </body>
</html>
```

While you look over Example 3-38, note several interesting things about how the page is designed:

- ✓ **The CSS is external**. CSS is defined in an external document. This makes it easy to change the style and helps you to focus on the HTML document in isolation.
- ✓ **The HTML code is minimal**. The code is very clean. It includes a form with a fieldset. The fieldset contains labels, input elements, and a button.
- ✓ **There isn't a table**. There's no need to add a table as an artificial organization scheme. A table wouldn't add to the clarity of the page. The form elements themselves provide enough structure to allow all the formatting you need.
- ✓ **Labels are part of the design**. I used the label element throughout the form, giving me an element that can be styled however I wish.
- ✓ **Everything is selectable**. I'll want to apply one CSS style to labels, another to input elements, and a third style to the button. I've set up the HTML so I can use CSS selectors without requiring any id or class attributes.
- ✓ **There's a button**. I used a button element instead of <input type = "button"> on purpose. This way, I can apply one style to all the input elements and a different style to the button element.

Designing a page like this one so its internal structure provides all the selectors you need is wonderful. This keeps the page very clean and easy to read. Still, don't be afraid to add classes or IDs if you need them.

It's often a good idea to look at your page with straight HTML before you start messing around with CSS.

Figure 3-47 demonstrates how the page looks with no CSS

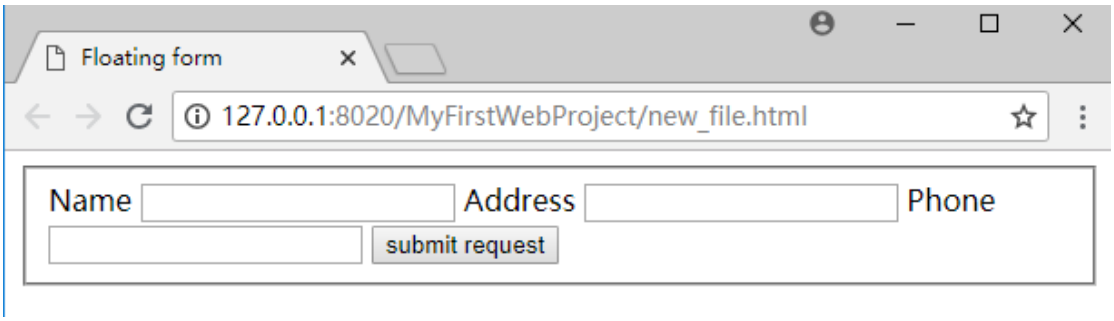


Figure 3-47 floating form without CSS

It'd be very nice to give the form a tabular feel, with each row containing a label and its associated input element. My first attempt at a CSS file for this page looked like this:

Example 3-39 CSS styles of floating form

```
fieldset {
background-color: #AAAAFF;
}
label {
float: left;
width: 5em;
text-align: right;
margin-right: .5em;
}
input {
background-color: #CCCCFF;
float: left;
}
button {
float: left;
width: 10em;
margin-left: 7em;
margin-top: 1em;
background-color: #0000CC;
color: #FFFFFF;
}
```

This CSS looks reasonable, but you'll find it doesn't quite work right. Here are the steps to build the CSS:

- ✓ Step 1. Add colors to each element. Colors are a great first step. For one thing, they ensure that your selectors are working correctly so that everything's where you think it is. This color scheme has a nice modern feel to it, with a lot of blues.
- ✓ Step 2. Float the labels to the left. Labels are all floated to the left, meaning they should move as far left as possible, and other things should be placed to the right of them.

- ✓ Step 3. Set the label width to 5em. This gives you plenty of space for the text the labels will contain.
- ✓ Step 4. Set the labels to be right-aligned. Right-aligning the labels makes the text snug up to the input elements but gives them a little margin-right so the text isn't too close.
- ✓ Step 5. Set the input's float to left. This tells each input element to go as far to the left (toward its label) as it can. The input element goes next to the label if possible and on the next line, if necessary. Like images, input elements have a default width, so it isn't absolutely necessary to define the width in CSS.
- ✓ Step 6. Float the button, too, but give the button a little top margin so it has a respectable space at the top. Set the width to 10em.

This seems to be a pretty good CSS file. It follows all the rules, but if you apply it to HTML, you'll be surprised by the results shown in Figure 3-48

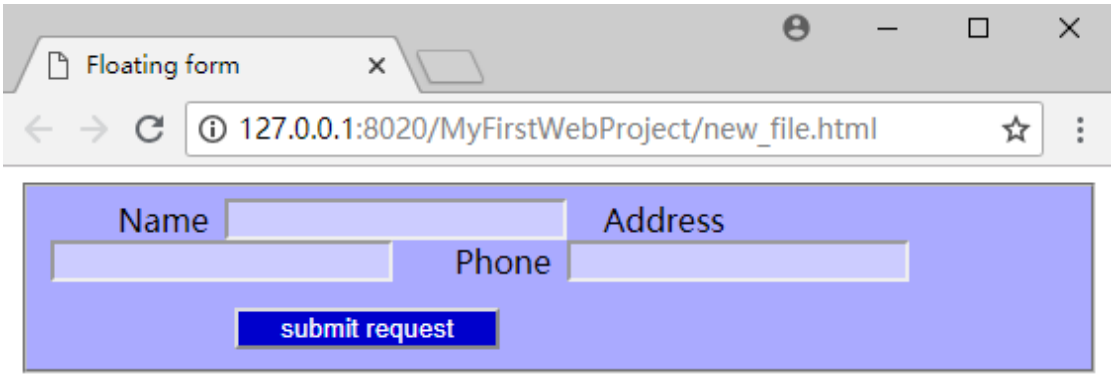


Figure 3-48 the first attempt of floating form

After all that talk about how nice float-based layout is, you're probably expecting something a bit neater. If you play around with the page in your browser, you'll find that everything works well when the browser is narrow, but when you expand the width of the browser, it gets ugly.

When CSS doesn't do what you want, it's usually acting on some false assumptions, which is the case here. Floating left causes an element to go as far to the left as possible and on the next line, if necessary. However, that's not really what you want on this page. The inputs should float next to the labels, but each label should begin its own line. The labels should float all the way to the left margin with the inputs floating

Using the clear attribute to control page layout

Adjusting the width of the container is a suitable solution, but it does feel like a bit of a hack. There should be some way to make the form work right, regardless of the container's width. There is exactly such a mechanism. The clear attribute is used on elements with a float attribute. The clear attribute can be set to left, right, or both. Setting the clear attribute to left means you want nothing to the left of this element. In other words, the element should be on the left margin of its container. That's exactly what you want here. Each label should begin its own line, so set its clear attribute to left.

To force the button onto its own line, set its clear attribute to both. This means that the button should have no elements to the left or the right. It should occupy a line all its own.

If you want an element to start a new line, set both its float and clear attributes to left. If you want an element to be on a line alone, set float to left and clear to both.

Using the clear attribute allows you to have a flexible-width container and still maintain reasonable control of the form design. Figure 1-14 shows that the form can be the same width as the page and still work correctly. This version works, no matter the width of the page.

Here's the final CSS code, including clear attributes in the labels and button:

Example 3-40 final CSS code of floating form

```
/* floatForm.css
   CSS file to go with float form
*/
fieldset {
  background-color: #AAAAFF;
}
label {
  clear: left;
  float: left;
  width: 5em;
  text-align: right;
  margin-right: .5em;
}
input {
  float: left;
  background-color: #CCCCFF;
}

button {
  float: left;
  clear: both;
  margin-left: 7em;
  margin-top: 1em;
  background-color: #0000CC;
  color: #FFFFFF;
}
```

3.6.5 Creating a Basic Two-Column Design

Many pages today use a two-column design with a header and footer. Such a page is quite easy to build with the techniques you read about in this section.

Designing the page

It's best to do your basic design work on paper, not on the computer. Here's my original sketch in Figure 3-49.

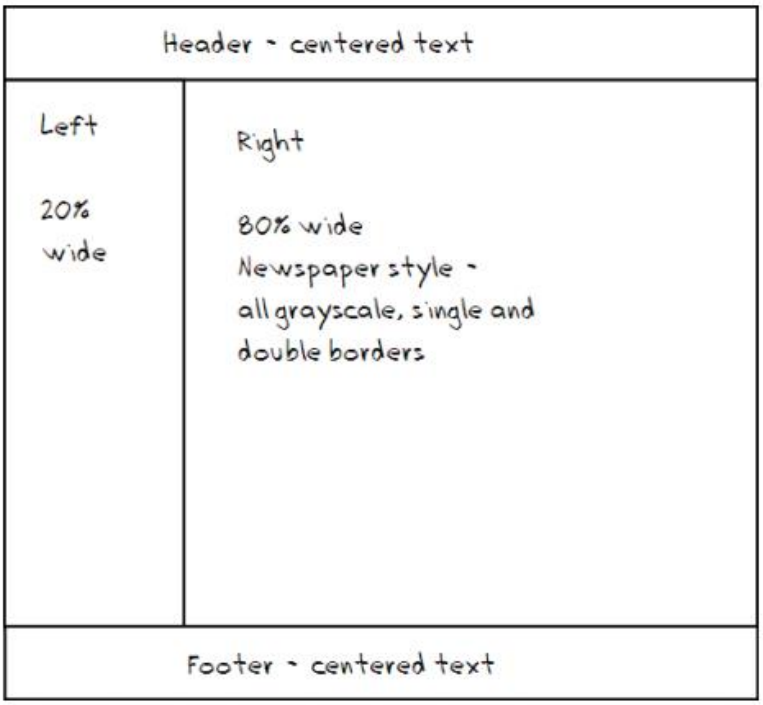


Figure 3-49 two column page

Draw the sketch first so you have some idea what you're aiming for. Your sketch should include the following information:

- ✓ Overall page flow: How many columns do you want? Will it have a header and footer?
- ✓ Section names: Each section needs an ID, which will be used in both the HTML and the CSS.
- ✓ Width indicators: How wide will each column be? (Of course, these widths should add up to 100 percent or less.)
- ✓ Fixed or percentage widths: Are the widths measured in percentages (of the browser size) or in a fixed measurement (pixels)? This has important implications. For this example, I'm using a dynamic width with percentage measurements.
- ✓ Font considerations: Do any of the sections require any specific font styles, faces, or colors?
- ✓ Color scheme: What are the main colors of your site? What will be the color and background color of each section?

Building the HTML

After you have a basic design in place, you're ready to start building the HTML code that will be

the framework. Start with basic CSS, but create a div for each section that will be in your final work. You can put a placeholder for the CSS, but don't add any CSS yet. Here's my basic code. I removed some of the redundant text to save space:

```
<!DOCTYPE html>
<html>
<head>
<meta charset = "UTF-8">
<title>twoColumn</title>
<link rel = "stylesheet"
type = "text/css"
href = "twoCol.css" />
</head>
<body>
<div id = "head">
<h1>Two Columns with Float</h1>
</div>
<div id = "left">
<h2>Left Column</h2>
<p>
This..here..has.a...paragraph...will...be.
</p>
</div>
<div id = "right">
<h2>Right Column</h2>
<p>
This..here..has.a...paragraph...will...be.
</p>
</div>
<div id = "footer">
<h3>Footer</h3>
</div>
</body>
</html>
```

Nothing at all is remarkable about this HTML code, but it has a few important features, such as:

- ✓ Its standards compliant. It's good to check and make sure the basic HTML code is well formed before you do a lot of CSS work with it. Sloppy HTML can cause you major headaches later.
- ✓ It contains four divs. The parts of the page that will be moved later are all encased in div elements.
- ✓ Each div has an ID. All the divs have an ID determined from the sketch.
- ✓ No formatting is in the HTML. The HTML code contains no formatting at all. That's left to the CSS.

✓ It has no style yet. Although a <link> tag is pointing to a style sheet, the style is currently empty.

Using temporary background colors

You won't keep these background colors, but they provide some very useful cues while you're working with the layout:

✓ **Testing the selectors**: While you change the background of each selector, you can see whether you've remembered the selector's name correctly. It's amazing how many times I've written code that I thought was broken just because I didn't write the selector properly.

✓ **Identifying the divs**: If you make each div a different color, it'll be easier to see which div is which when they are not acting the way you want.

✓ **Specifying the size of each div**: The text inside a div isn't always a good indicator of the actual size. The background color tells you what's really going on.

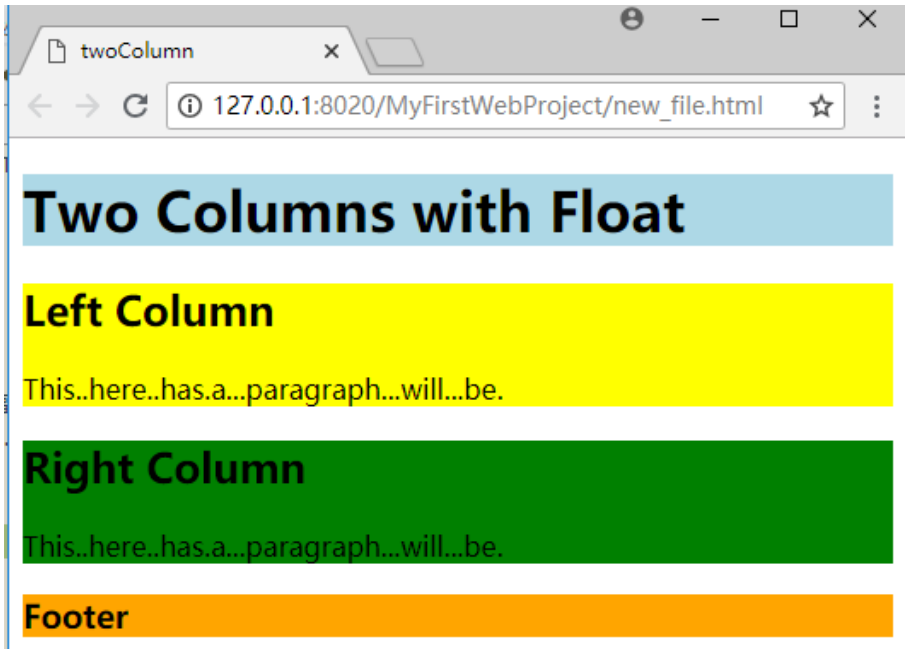


Figure 3-50 with background color

Figure 3-50 displays how the page looks with the background colors turned on.

Setting up the floating columns

```
#head {
border: 3px black solid;
}
#left {
border: 3px red solid;
float: left;
```

```
width: 20%;
}
#right {
border: 3px blue solid;
float: left;
width: 75%
}
#footer {
border: 3px green solid;
clear: both;
}
```

I made the following changes to the CSS:

- ✓ Float the #left div. Set the #left div's float property to left so other divs (specifically the #right div) are moved to the right of it.
- ✓ Set the #left width. When you float a div, you must also set its width. I've set the left div width to 20 percent of the page width as a starting point.
- ✓ Float the #right div, too. The right div can also be floated left, and it'll end up snug to the left div. Don't forget to add a width. I set the width of #right to 75 percent, leaving another 5 percent available for padding, margins, and borders.
- ✓ Clear the footer. The footer should take up the entire width of the page, so set its clear property to both.

Figure 3-51 shows how the page looks with this style sheet in place

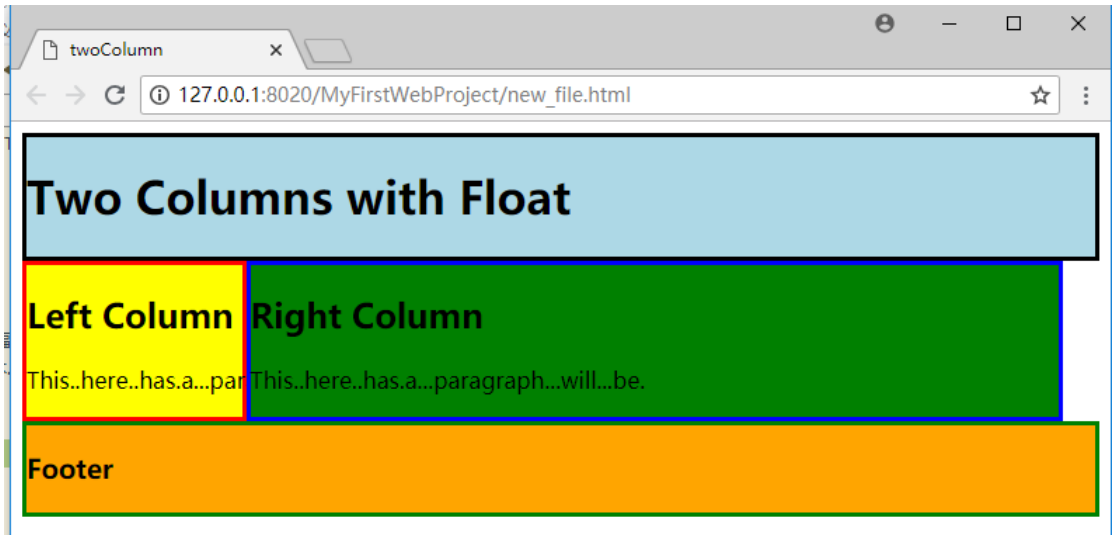


Figure 3-51 float the left

Tuning up the borders

The colored backgrounds in Figure 3-51 point out some important features of this layout scheme.

For instance, the two columns are not the same height. This can have important implications.

You can change the borders to make the page look more like a column layout. I'm going for a newspaper-style look, so I use simple double borders. I put a black border under the header, a gray border to the left of the right column, and a gray border on top of the bottom segment. Tweaking the padding and centering the footer complete the look. Here's the complete CSS:

```
#head {
border-bottom: 3px double black;
}
#left {
float: left;
width: 20%;
}
#right {
float: left;
width: 75%;
border-left: 3px double gray;
}
#footer {
clear: both;
text-align: center;
border-top: 3px double gray;
}
```

The final effect is shown in Figure 3-52

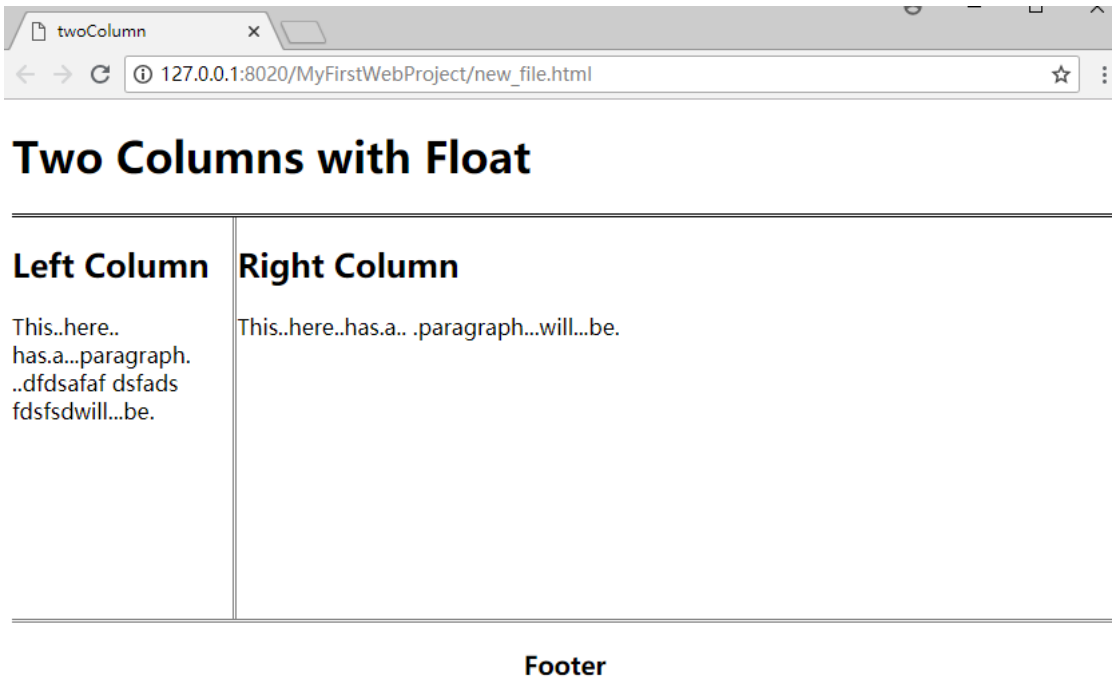


Figure 3-52 final effect of two-column

3.7 Quiz

1. What three ways are used to add CSS to HTML in this chapter, and describe the advantage and disadvantage of these ways?

2. What three parts are consisted of the selector of style tag?

3. Describe the href attribute of link tag. And give an example for each way.

4. What is the two main ways to define colors?

5. What is your impression about RGB color?

6. List the generic fonts and what is it used for.

7. What is fonts list? Create a fonts list that the browser choose Times New Roman when it's a windows machine, Times in Mac, and FreeSerif in Linux

8. Lists the font file types supported in browser.

9. How to decide the size of text in a page?

10. Lists the detail step to style a paragraph using class

11. List three selector

12. Describe the detail steps of linked style.

13. What is the meaning of h1+p?

14. What are the three attributes that borders require?

15. Lists the predefined border styles.

16. What are the layers of block-level?

17. Describe the history of page layout.

Tips: frames, table, image, flash, div + CSS, and so on.

18. Describe how to create a two-column page.

3.8 Practice

1. Create a page to set different inline style for each list item.

```
<ul>
<li><em>The Rainbow Returns</em> by E. Smith</li>
<li><em>Seven Steps to Immeasurable Wealth</em> by R. U. Needy</li>
<li><em>The Food-Lovers Guide to Weight Loss</em> by L. Goode</li>
<li><em>The Silly Person's Guide to Seriousness</em> by M. Nott</li>
</ul>
```

2. Create a page, use embedded style to change the style of the two tags. Of course, you can change the text of tags.

```
<h1>Red text on a yellow background</h1>
<p>Yellow text on a red background</p>
```

3. Create a page, use style sheet file to change the style of the two tags. Of course, you can change the text of tags.

```
<h1>Red text on a yellow background</h1>
<p> Yellow text on a red background</p>
```

4. Create a simple page to display all the named colors

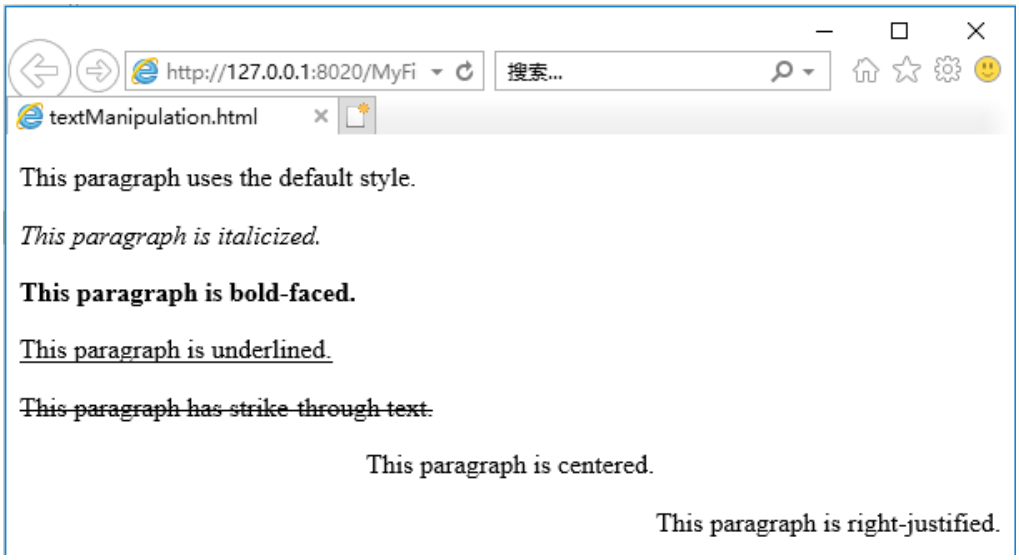
5. Changing your color with Chrome developer tools.

```
<h1>Red text on a yellow background</h1>
<p>Yellow text on a red background</p>
```

6. Create a page and show all the generic fonts. The result is shown as the following.

7. Create a page using embedded fonts which you can download from website.

8. Create a page as the following figure.



10. Create a page using ID selector

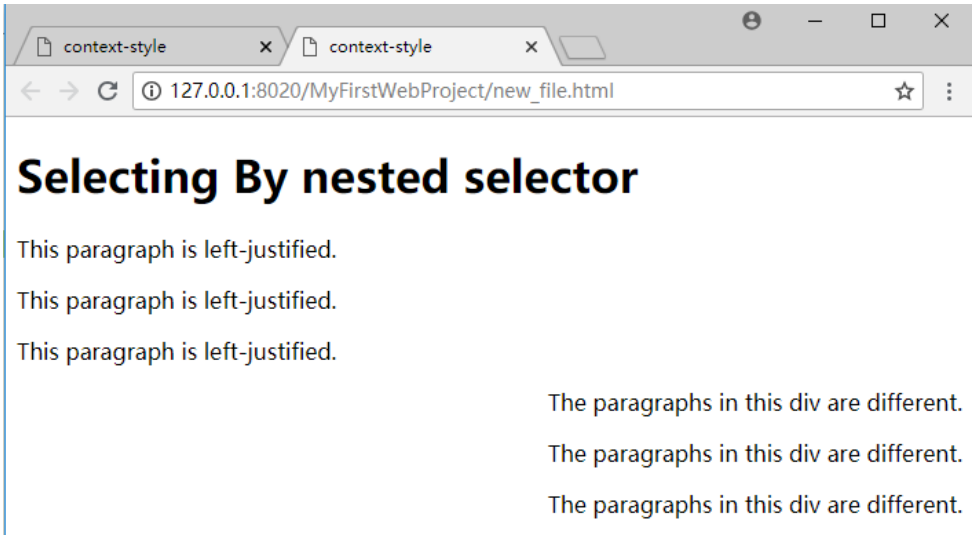
11. Create a page using class selector

12. Create a page using combining selector that include an element and a class name, for example: p.textColor

13. Create a page using combining class. The page include three paragraphs, the first two paragraphs have different style, the last paragraph has the two style defined in the first two paragraphs.

14. Create your favorite linked styles, and then create a page using these styles.

15 Create a page using nested selection. The page may like the following page.



16. Creating a partial borders like the example.

Paragraphs have several borders defined.

Paragraphs have several borders defined.

17. Creating a page with three paragraphs which has different padding, margin and border.

18. Create a round corner for a paragraph.

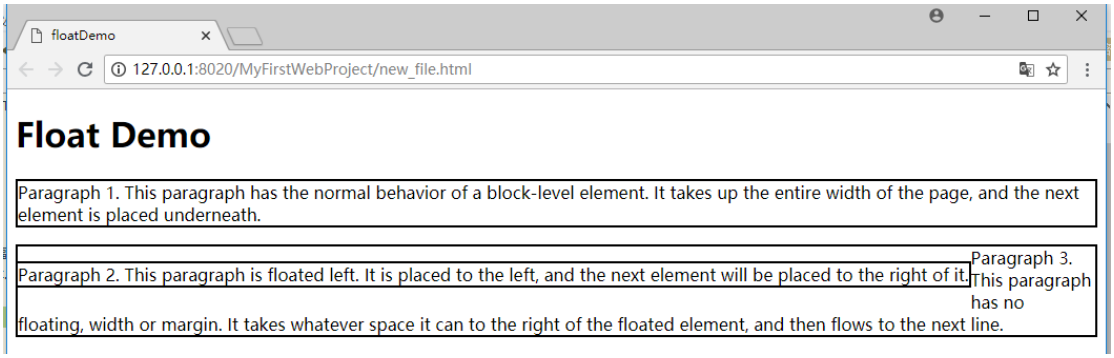
19. Create a shadow for a div and describe the detail steps.

20 Create a page with background image:

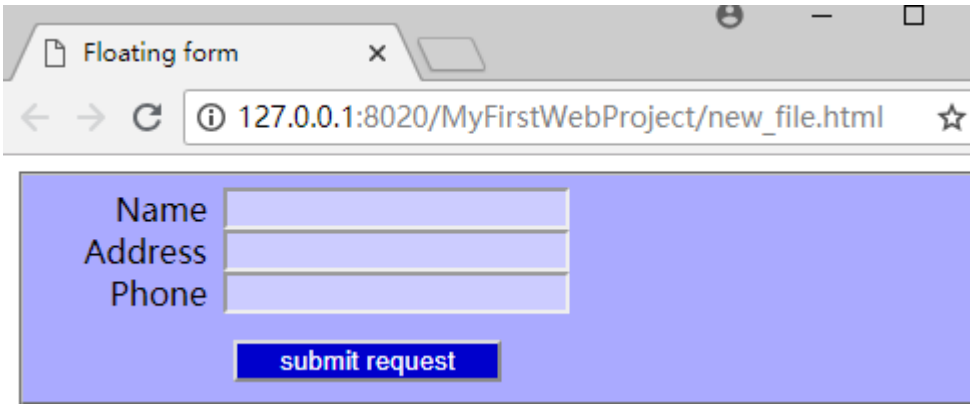
(1) When you resize the page size, the image fit for the page.

(2) Describe the detail steps.

21 Create a float paragraph and describe the detail steps.



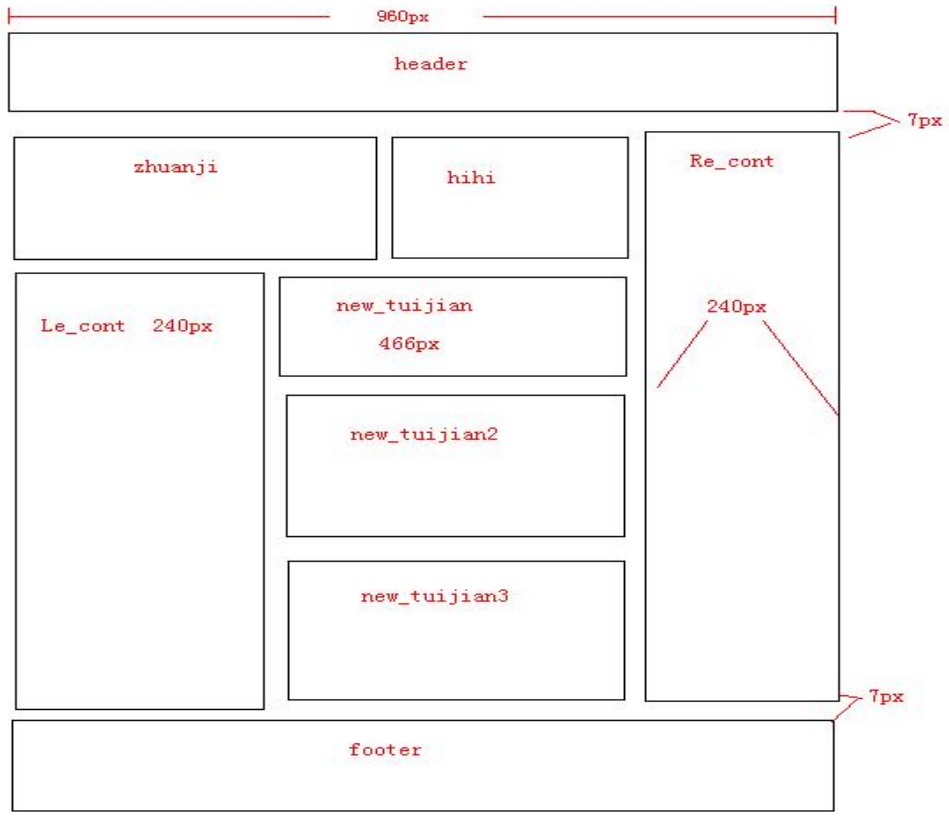
22. Create a page with floating form.



23. Create a two-column page as you like.

24. Create a three-column page as you like.

25. Create a page as the figure. (Tips: the source code is shown after the figure.)




```
<!DOCTYPE html>
<html>
    <head>
        <meta charset=utf-8" />
        <title>An example of page layout</title>
        <style type="text/css">
            * {
                margin: 0;
                padding: 0;
            }
            body {
                width: 960px;
                position: relative;
                left: 50%;
                margin-left: -480px;
            }
            div {
                border: 1px solid black;
            }
            #header,#fooder {
                height: 100px;
            }
            #container {
                float: left;
```

```
            border: none;
        }
        .floatL {
            float: left;
            width: 713px;
            border: none;
        }
        .floatR {
            float: right;
            width: 240px;
        }
        .zhuanji {
            width: 464px;
            float: left;
            height: 180px;
        }
        .hihi {
            width: 240px;
            height: 180px;
            float: left
        }
        .ri_cont {
            width: 240px;
            height: 500px;
```

```
            float: left;
        }
        .le_cont {
            float: left;
            width: 240px;
            height: 315px;
        }
        .new_tuijian01, .new_tuijian02, .new_tuijian03 {
            width: 464px;
            height: 100px;
            float: right;
        }
        .clearB {
            clear: both;
        }
        .mb5 {
            margin-bottom: 5px;
        }
        .mb7 {
            margin-bottom: 7px;
        }
        .mr5 {
            margin-right: 5px;
        }
```

```
        </style>
    </head>

    <body>
        <div id="header" class="mb7">header</div>
        <div id="container" class="mb7">
            <div class="floatL mr5">
                <div class="zhuanji mr5 mb5">zhuanji</div>
                <div class="hihi">hihi</div>
                <div class="le_cont mr5">le_cont</div>
                <div class="new_tuijian01 mb5">new_tuijian01</div>
                <div class="new_tuijian02 mb5">new_tuijian02</div>
                <div class="new_tuijian03">new_tuijian03</div>
            </div>
            <div class="ri_cont floatR">ri_cont</div>
        </div>
        <div id="fooder" class="clearB"></div>
    </body>
</html>
```

4 Using JavaScript

4.1 Getting Started with JavaScript

Web pages are defined by the HTML code and fleshed out by CSS. But to make them move and breathe, sing, and dance, you need to add a programming language or two. If you thought building web pages was cool, you're going to love what you can do with a little programming. Programming is what makes pages interact with the user. Learn to program, and your pages come alive.

Sometimes people are nervous about programming. It seems difficult and mysterious, and only super-geeks do it. That's a bunch of nonsense. Programming is no more difficult than HTML and CSS. It's a natural extension, and you're going to like it.

In this chapter, you discover how to add code to your web pages. You use a language called JavaScript, which is already built into most web browsers.

4.1.1 Working in JavaScript

JavaScript is a programming language first developed by Netscape Communications. It is now standard on nearly every browser. You should know a few things about JavaScript right away:

- ✓ **It's a real programming language.** JavaScript doesn't have all the same features as a monster, such as C++ or VB.NET, but it still has all the hallmarks of a complete programming language.
- ✓ **It's not Java.** Sun Microsystems developed a language called Java, which is also sometimes used in web programming. Despite the similar names, Java and JavaScript are completely different languages. The original plan was for JavaScript to be a simpler language for controlling more complex Java applets, but that never really panned out. Don't go telling people you're programming in Java. Java people love to act all superior and condescending when JavaScript programmers make this mistake.
- ✓ **It's a scripting language.** As programming languages go, JavaScript's pretty friendly. It's not quite as strict or wordy as some other languages. It also doesn't require any special steps (such as compilation), so it's pretty easy to use. These things make JavaScript a great first language.

4.1.2 Writing Your First JavaScript Program

The foundation of any JavaScript program is a standard web page like the ones featured in the first three chapters.

To create your first JavaScript program, you need to add JavaScript code to your pages. Figure 4-1 shows the classic first program in any language.

Figure 4-1 the first JavaScript program

This page has a very simple JavaScript program in it that pops up the phrase “Hello, World!” in a special element called a dialog box.

Example 4-1 the first JavaScript program

```
<!DOCTYPE html>
<html>
<head>
<meta charset = "UTF-8">
<title>HelloWorld</title>
<script type = "text/javascript">
// Hello, world!
alert("Hello, World!");
</script>
</head>
<body>
</body>
</html>
```

As you can see, this page contains nothing in the HTML body. You can incorporate JavaScript with HTML content. For now, though, you can simply place JavaScript code in the head area in a special tag and make it work.

Embedding your JavaScript code

JavaScript code is placed in your web page via the <script> tag. JavaScript code is placed inside the <script></script> pair. The <script> tag has one required attribute, type, which will usually be text/javascript. (Other types are possible, but they’re rarely used.)

Creating comments

Just like HTML and CSS, comments are important. Because programming code can be more difficult to decipher than HTML or CSS, it’s even more important to comment your code in JavaScript than it is in these environments. The comment character in JavaScript is two slashes (//).The browser ignores everything from the two slashes to the end of the line. You can also use a

multi-line comment (/* */) just like the one in CSS.

Using the alert() method for output

You can output data in JavaScript in a number of ways. In this section, I focus on the simplest to implement and understand — the alert(). This technique pops up a small dialog box containing text for the user to read. The alert box is an example of a modal dialog. Modal dialogs interrupt the flow of the program until the user pays attention to them. Nothing else will happen in the program until the user acknowledges the dialog by clicking the OK button. The user can't interact with the page until he clicks the button.

Adding the semicolon

Each command in JavaScript ends with a semicolon (;) character. The semicolon in most computer languages acts like the period in English. It indicates the end of a logical thought. Usually, each line of code is also one line in the editor.

To tell the truth, JavaScript will usually work fine if you leave out the semicolons. However, you should add them anyway because they help clarify your meaning. Besides, many other languages, including PHP, require semicolons. You may as well start a good habit now

4.1.3 Introducing Variables

Computer programs get their power by working with information. Figure 4-2 shows a program that gets user data from the user to include in a customized greeting.

Figure 4-2 ask user for user name

This program introduces a new kind of dialog that allows the user to enter some data. The information is stored in the program for later use. After the user enters her name, she gets a greeting, as shown in Figure 4-3.

Figure 4-3 the first greeting for asking user name

The rest of the greeting happens in a second dialog box, shown in Figure 4-4. It incorporates the username supplied in the first dialog box as Figure 4-2.

Figure 4-4 the second greeting for asking user name

The output may not seem that incredible, but take a look at the source code to see what's happening:

Example 4-2 Create the first variable

```
<!DOCTYPE html>
<html>
<head>
<meta charset = "UTF-8">
<title>prompt</title>
<script type = "text/javascript">
// from prompt
var person = "";
person = prompt("What is your name?");
alert("Hi");
alert(person);
</script>
</head>
<body>
</body>
</html>
```

Variable

This program is interesting because it allows user interaction. The user can enter a name, which is stored in the computer and then returned in a greeting. The key to this program is a special element called a variable.

Variables are simply places in memory for holding data. Any time you want a computer program to "remember" something, you can create a variable and store your information in it.

Variables typically have the following characteristics:

- ✓ The var statement: You can indicate that you're creating a variable with the var command.
- ✓ A name: When you create a variable, you're required to give it a name.
- ✓ An initial value: It's useful to give each variable a value immediately.
- ✓ A data type: JavaScript automatically determines the type of data in a variable, but you should still be clear in your mind what type of data you expect a variable to contain.

Asking the user for information

The prompt statement does several interesting things:

- ✓ Pops up a dialog box. This modal dialog box is much like the one the alert() method creates.
- ✓ Asks a question. The prompt() command expects you to ask the user a question.
- ✓ Provides space for a response. The dialog box contains a space for the user to type a response and buttons for the user to click when he's finished or wants to cancel the operation.
- ✓ Passes the information to a variable. The purpose of a prompt() command is to get data from the user, so prompts are nearly always connected to a variable. When the code is finished, the variable contains the indicated value.

Responding to the user

This program uses the alert() statement to begin a greeting to the user. The first alert works just like the one from the helloWorld program, described earlier in this chapter:

```
alert("Hi");
```

The content of the parentheses is the text you want the user to see. In this case, you want the user to see the literal value "Hi". The second alert() statement is a little bit different:

```
alert(person);
```

This alert() statement has a parameter with no quotes. Because the parameter has no quotes, JavaScript understands that you don't really want to say the text person. Instead, it looks for a variable named person and returns the value of that variable.

The variable can take any name, store it, and return a customized greeting

4.1.4 Understanding the String Object

To have a greeting and a person's name on two different dialogs seems a little awkward. Figure 4-5 shows a better solution.

Figure 4-5 user name and greeting

The program asks for a name again and stores it in a variable. This time, the greeting is combined into one alert, which looks a lot better.

The secret to Figure 4-5 is one of those wonderful gems of the computing world: a really simple idea with a really complicated name. The term concatenation is a delightfully complicated word for a basic process. Look at the following code, and you see that combining variables with text is not all that complicated:

Example 4-3 username and greeting

```
<script type = "text/javascript">
var person = "";
person = prompt("What is your name?");
alert("Hi, "+ person);
</script>
```

For the sake of brevity, I include only the script tag and its contents throughout this section. The rest of this page is a standard blank HTML page.

The program contains two kinds of text. "Hi, " is a literal text value. That is, you really mean to say “Hi, " (including the comma and the space). person is a variable.

You can combine literal values and variables in one phrase if you want:

```
alert("Hi there, " + person + "!");
```

The secret to this code is to follow the quotes. "Hi, " is a literal value because it is in quotes. On the other hand, person is a variable name because it is not in quotes; "!" is a literal value. You can combine any number of text snippets together with the plus sign. Using the plus sign to combine text is called concatenation.

You may be curious about the extra space between the comma and the quote in the output line:

alert("Hi, " + person + "!");

This extra space is important because you want the output to look like a normal sentence. If you don’t have the space, the computer doesn’t add one, and the output looks like this:

Hi ,Tom!

You need to construct the output as it should look, including spaces and punctuation.

The person variable used in the previous program is designed to hold text. Programmers (being

programmers) devised their own mysterious term to refer to text. In programming, text is referred to as string data.

The term string comes from the way text is stored in computer memory. Each character is stored in its own cell in memory, and all the characters in a word or phrase reminded the early programmers of beads on a string.

JavaScript (and many other modern programming languages) uses a powerful model called object-oriented programming (OOP). This style of programming has a number of advantages. Most important for beginners, it allows you access to some very powerful objects that do interesting things out of the box.

Objects are used to describe complicated things that can have a lot of characteristics — like a cow. You can't really put an adequate description of a cow in an integer variable.

In many object-oriented environments, objects can have the following characteristics. (Imagine a cow object for the examples.)

- ✓ Properties: Characteristics about the object, such as breed and age
- ✓ Methods: Things the objects can do, such as moo() and giveMilk()
- ✓ Events: Stimuli the object responds to, such as onTip

If you have a variable of type cow, it describes a pretty complicated thing. This thing might have properties, methods, and events, all of which can be used together to build a good representation of a cow.

Most variable types in Java are actually objects, and most JavaScript objects have a full complement of properties and methods; many even have event handlers. Master how these things work and you've got a powerful and compelling programming environment.

When you assign text to a variable, JavaScript automatically treats the variable as a string object. The object instantly takes on the characteristics of a string object. Strings have a couple of properties and a bunch of methods. The one interesting property (at least for beginners) is length. Look at the example in Figure 4-6 to see the length property in action.

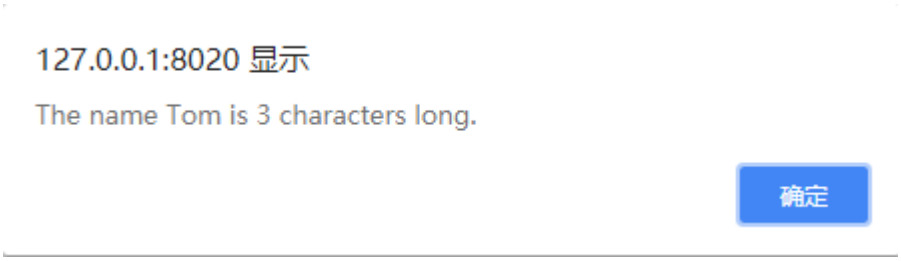


Figure 4-6 length of the String

Here is the code:

Example 4-4 length of the Sting

```
<script type = "text/javascript">
var person = prompt("Please enter your name.");
var length = person.length;
alert("The name " + person + " is " + length + " characters long.");
</script>
```

A property is used like a special subvariable. For example, person is a variable in the previous example. person.length is the length property of the person variable. In JavaScript, an object and a variable are connected by a period (with no spaces).

The string object in JavaScript has only two other properties (constructor and prototype). Both of these properties are needed only for advanced programming, so I skip them for now.

Using string methods to manipulate text

The length property is kind of cool, but the string object has a lot more up its sleeve. Objects also have methods (things the object can do). Strings in JavaScript have all kinds of methods. Here are a few of my favorites:

toUpperCase() makes an entirely uppercase copy of the string.

toLowerCase() makes an entirely lowercase copy of the string.

substring() returns a specific part of the string.

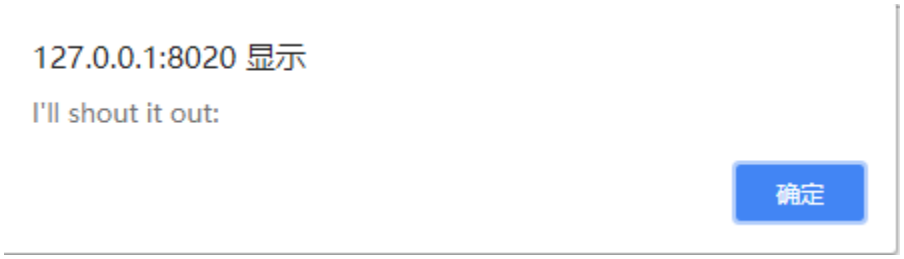
indexOf() determines whether one string occurs within another.

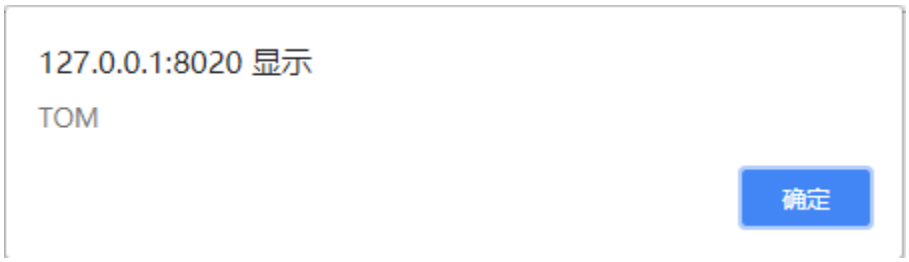
The string object has many other methods, but I'm highlighting the preceding because they're useful for beginners. Many string methods, such as big() and fontColor(), simply add HTML code to text. They aren't used very often because they produce HTML code that won't validate, and they don't really save a lot of effort anyway. Some other methods, such as search(), replace(), and slice(), use advanced constructs like arrays and regular expressions that aren't necessary for beginners.

The best way to see how methods work is to look at some in action. Look at the code for Example 4-5 string methods, and Figure 4-7 displays the result.

Example 4-5 string methods

```
<script type = "text/javascript">
var text = new String;
text = prompt("Please enter some text.");
alert("I'll shout it out:");
alert(text.toUpperCase());
alert("Now in lowercase...");
alert(text.toLowerCase());
alert("The first 'm' is at letter...");
alert(text.indexOf("m"));
alert("The first three letters are ...");
alert(text.substring(0, 3));
</script>
```


127.0.0.1:8020 显示
I'll shout it out:
确定


127.0.0.1:8020 显示
TOM
确定


127.0.0.1:8020 显示
Now in lowercase...
确定


127.0.0.1:8020 显示
tom
确定


127.0.0.1:8020 显示
The first 'm' is at letter...
确定

Figure 4-7 result of string methods

In this example, I explicitly defined text as a string variable by saying

```
var text = new String;
```

JavaScript does not require you to explicitly determine the type of a variable, but you can do so, and this is sometimes helpful.

4.1.5 Understanding Variable Types

JavaScript isn't too fussy about whether a variable contains text or a number, but the distinction is still important because it can cause some surprising problems. To illustrate, take a look at a program that adds two numbers together, and then see what happens when you try to get numbers from the user to add.

Adding numbers

First, take a look at the following program:

Example 4-6 add number

```
<script type = "text/javascript">
var x = 5;
var y = 3;
```

```
var sum = x + y;
alert(x + " plus " + y + " equals " + sum);
</script>
```

(As usual for this section, I'm only showing the script part because the rest of the page is blank.)

This program features three variables. I've assigned the value 5 to x and 3 to y. I then add x + y and assign the result to a third variable, sum. The last line prints the results, which are also shown in Figure 4-8.

Figure 4-8 add number

Note a few important things from this example:

- ✓ You can assign values to variables. It's best to read the equal sign as "gets" so that the first assignment is read as "variable x gets the value 5."

  ```
  var x = 5;
  ```

- ✓ Numeric values aren't enclosed in quotes. When you refer to a text literal value, it's always enclosed in quotes. Numeric data, such as the value 5, isn't placed in quotes.
- ✓ You can add numeric values. Because x and y both contain numeric values, you can add them together.
- ✓ You can replace the results of an operation in a variable. The result of the calculation x + y is placed in a variable called sum.
- ✓ Everything works as expected. The behavior of this program works as expected. That's important because it's not always true.

Adding the user's numbers

The natural extension of the addNumbers.html program is a feature that allows the user to input two values and then returns the sum. This program can be the basis for a simple adding machine. Here's the JavaScript code:

Example 4-7 add input number

```
<script type = "text/javascript">
var x = prompt("first number:");
var y = prompt("second number:");
var sum = x + y;
alert(x + " plus " + y + " equals " + sum);
```

```
</script>
```

This code seems reasonable enough. It asks for each value and stores them in variables. It then adds the variables and returns the results, right? Well, look at Figure 4-9 to see a surprise.

Figure 4-9 add two input number

Something's obviously not right here. To understand the problem, you need to see how JavaScript makes guesses about data types

The trouble with dynamic data

Ultimately, all the information stored in a computer, from music videos to e-mails, is stored as a bunch of ones and zeroes. The same value 01000001 can mean all kinds of things: It may mean the number 65 or the character A. (In fact, it does mean both those things in the right context.) The same binary value may mean something entirely different if it's interpreted as a real number, a color, or a part of a sound file. The theory isn't critical here, but one point is really important: Somehow the computer has to know what kind of data is stored in a specific variable.

Many languages, such as C and Java, have all kinds of rules about defining data. If you create a variable in one of these languages, you have to define exactly what kind of data will go in the

variable, and you can't change it.

JavaScript is much more easygoing about variable types. When you make a variable, you can put any kind of data in it that you want. In fact, the datatype can change. A variable can contain an integer at one point, and the same variable may contain text in another part of the program.

JavaScript uses the context to determine how to interpret the data in a particular variable. When you assign a value to a variable, JavaScript puts the data in one of the following categories:

- ✓ Integers are whole numbers (no decimal part). They can be positive or negative values.
- ✓ A floating point number has a decimal point — for example, 3.14. You can also express floating point values in scientific notation, such as 6.02e23 (Avogadro's number –6.02 times 10 to the 23rd). Floating point numbers can also be negative.
- ✓ A Boolean value can only be true or false.
- ✓ Text is usually referred to as string data in programming languages. String values are usually enclosed in quotes.
- ✓ Arrays and objects are more complex data types that you can ignore for now.

Most of the time, when you make a variable, JavaScript guesses right, and you have no problems. But sometimes, JavaScript makes some faulty assumptions, and things go wrong.

The pesky plus sign

I use the plus sign in two ways throughout this chapter. The following code uses the plus sign in one way (concatenating two string values):

```
var x = "Hi, ";
var y = "there!";
result = x + y;
alert(result);
```

In this code, x and y are text variables. The result = x + y line is interpreted as "concatenate x and y," and the result is "Hi, there!" Here's the strange thing: The following code is almost identical.

```
var x = 3;
var y = 5;
result = x + y;
alert(result);
```

Strangely, the behavior of the plus sign is different here, even though the statement result = x + y is identical in the two code snippets. In this second case, x and y are numbers. The plus operator has two entirely different jobs. If it's surrounded by numbers, it adds. If it's surrounded by text, it concatenates. That's what happened to the first adding machine program. When the user enters data in prompt dialogs, JavaScript assumes that the data is text. When I try to add x and y, it "helpfully" concatenates instead.

Changing Variables to the Desired Type

If JavaScript is having a hard time figuring out what type of data is in a variable, you can give it a

friendly push in the right direction with some handy conversion functions, as shown in Table 4-1.

Table 4-1 Variable Conversion Functions

Function	From	To	Example	Result
parseInt()	String	Integer	parseInt(“23”)	23
parseFloat()	String	Floating point	parseFloat(“21.5”)	21.5
toString()	Any variable	String	myVar.toString()	varies
eval()	Expression	Result	eval(“5 + 3”)	8
Math.ceil()	Floating point	Integer	Math.ceil(5.2)	6
Math. floor ()	Floating point	Integer	Math. floor (5.2)	5
Math.round()	Floating point	Integer	Math. round (5.2)	5

Using variable conversion tools

The conversion functions are incredibly powerful, but you only need them if the automatic conversion causes you problems. Here’s how they work:

- ✓ parseInt() is used to convert text to an integer. If you put a text value inside the parentheses, the function returns an integer value. If the string has a floating-point representation (“4.3” for example), an integer value (4) is returned.
- ✓ parseFloat() converts text to a floating-point value.
- ✓ toString() takes any variable type and creates a string representation. Usually, using this function isn’t necessary to use because it’s invoked automatically when needed.
- ✓ eval() is a special method that accepts a string as input. It then attempts to evaluate the string as JavaScript code and return the output. You can use this method for variable conversion or as a simple calculator — eval(“5 + 3”) returns the integer 8.
- ✓ Math.ceil() is one of several methods of converting a floating-point number to an integer. This technique always rounds upward, so Math. ceil(1.2) is 2, and Math.ceil(1.8) is also 2.
- ✓ Math.floor() is similar to Math.ceil(), except it always rounds downward, so Math.floor(1.2) and Math.floor(1.8) will both evaluate to 1.
- ✓ Math.round() works like the standard rounding technique used in grade school. Any fractional value less than .5 rounds down, and greater than or equal to .5 rounds up, so Math.round(1.2) is 1, and Math. round(1.8) is 2.

4.2 Talking to the Page

JavaScript is fun and all, but it lives in web browsers for a reason: to let you change web pages. The best thing about JavaScript is how it helps you control the page. You can use JavaScript to read useful information from the user and to change the page on the fly.

4.2.1 Understanding the Document Object Model

JavaScript programs usually live in the context of a web page. The contents of the page are available to the JavaScript programs through a mechanism called the Document Object Model (DOM).

The DOM is a special set of complex variables that encapsulates the entire contents of the web page. You can use JavaScript to read from the DOM and determine the status of an element. You can also modify a DOM variable and change the page from within JavaScript code.

The easiest way to get a feel for the DOM is to load a page in Chrome and play around in the console. Follow these steps to get a feel for the DOM:

- ✓ Step 1. Use the Chrome browser. Most browsers have something like the web developer console used in this example, but Chrome's is very easy to use and comes built-in, so you should begin with that one.
- ✓ Step 2. Load any page you want. It's probably easiest to start with a page that's relatively simple, so you can get a sense of what's happening
- ✓ Step 3. Turn on the web developer toolbar. Use the F12 key, View ➪ Developer ➪ Developer Tools or Tools ➪ Developer Tools from the menu. (It may vary based on your version of Chrome or your operating system.)
- ✓ Step 4. Go to the Console tab. The Developer Tools window has many tabs, but the console tab is the one we need for now (and it will continue to be useful as you get more advanced).
- ✓ Step 5. Type document. Don't forget the period at the end. When you type a period, Chrome's auto-complete describes all the various elements related to the document. document is a very fancy variable (called an object) that contains a ton of sub-variables. You can scroll through this list to see all the things related to document.
- ✓ Step 6. Change the page's background color. Try typing this in the console: document.body.style.backgroundColor = "green" You can use this trick to (temporarily) change all kinds of features.
- ✓ Step 7. Play around with the document tree a bit more. It's fine if you don't know exactly what's going on yet, but use this technique to get a general feel for the complexity of the page and all the interesting things you can do with it.

Figure 4-10 illustrates a simple web page being dynamically modified through the console tab.

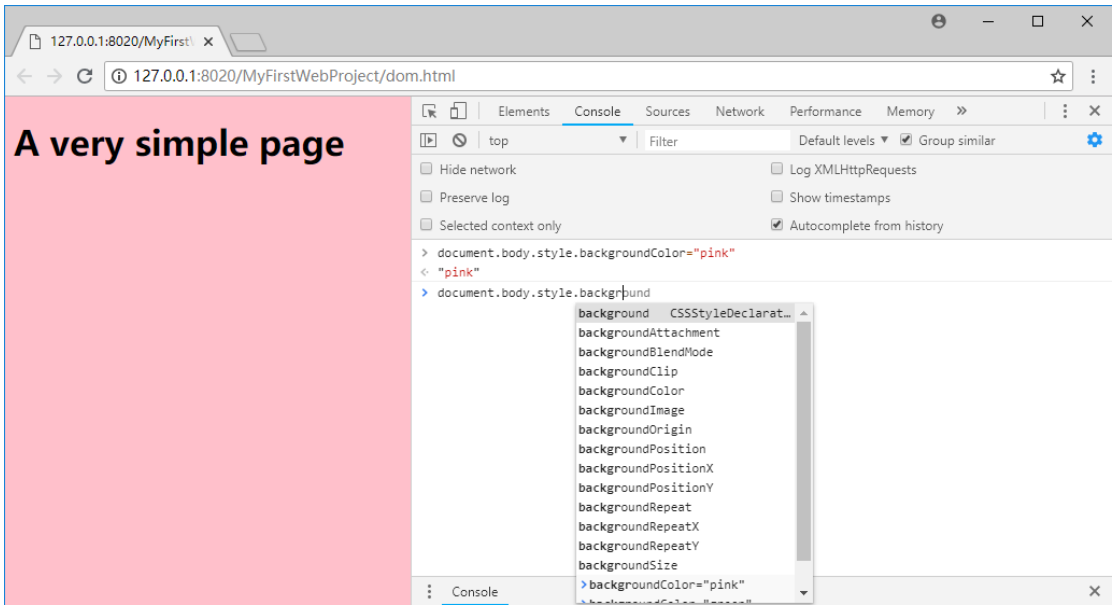


Figure 4-10 console of DOM tree

The Console tab is far more involved and powerful than I'm letting on here. The remainder of this chapter goes into all kinds of details about how to use this powerful tool to figure out what's going on in your page.

When you look over the DOM of a simple page, you can easily get overwhelmed. You'll see a lot of variables listed. Technically, these variables are all elements of a special object called window. The window object has a huge number of sub objects, all listed in the DOM view. Table 4-2 describes a few important window variables

Table 4-2 primary DOM objects

Variable	Description	Notes
document	Represents HTML page	Most commonly scripted element
location	Describes current URL	Change location.href to move to a new page
history	A list of recently visited pages	Access this to view previous pages
status	The browser status bar	Change this to set a message in the status bar

It all gets fun when you start to write JavaScript code to access the DOM.

Example 4-8 JavaScript style

```
<!DOCTYPE html>
<html>
<head>
```

```
<meta charset = "UTF-8">
<title>blue</title>
</head>
<body>
<h1>I've got the JavaScript Blues</h1>
<script type = "text/javascript">
document.body.style.color = "white";`
document.body.style.backgroundColor = "peachpuff ";
</script>
</body>
</html>
```

Take a look at Example 4-8 in Figure 4-11.

Figure 4-11 JavaScript style

The page has white text on a peachpuff background, but there's no CSS! Instead, it has a small script that changes the DOM directly, controlling the page colors through code.

4.2.2 Managing Button Events

The page shown in Figure 4-12 is pretty simple, but it has a few unique features.

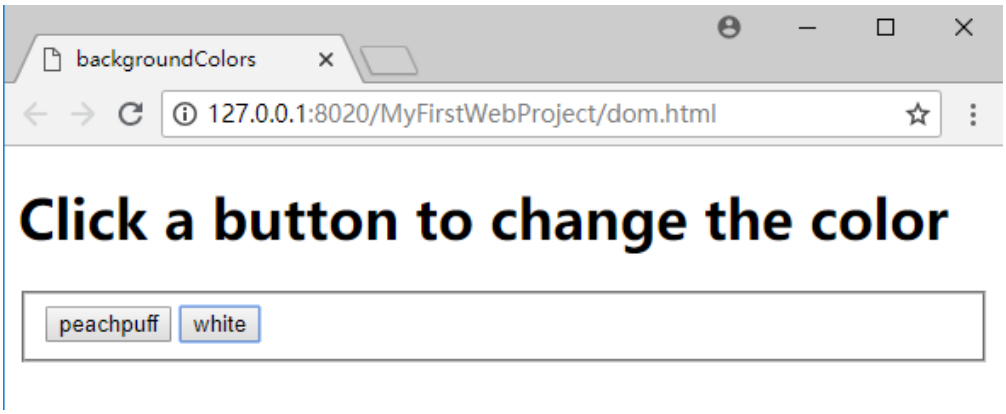


Figure 4-12 change background color

Example 4-9 change background colo

```
<!DOCTYPE html>
<html >
  <head>
    <meta charset = "UTF-8">
    <title>backgroundColors</title>
    <script type = "text/javascript">
      // from backgroundColors
      function makePeachpuff(){
        document.body.style.backgroundColor = "peachpuff";
      } // end changeColor
      function makeWhite(){
        document.body.style.backgroundColor = "white";
      }
    </script>
  </head>
  <body>
    <h1>Click a button to change the color</h1>
    <form action = "">
      <fieldset>
        <input type = "button"  value = "peachpuff"
              onclick = "makePeachpuff()"/>
        <input type = "button" value = "white"
              onclick = "makeWhite()" />
      </fieldset>
    </form>
  </body>
</html>
```

From the Example 4-9, we know as follow.

- ✓ **It has no CSS**. A form of CSS is dynamically created through the code.
- ✓ **The script is in the body.** I can't place this particular script in the header because it refers to the body. When the browser first sees the script, there must be a body for the text to change. If I put the script in the head, no body exists when the browser reads the code, so it gets confused. If I place the script in the body, there is a body, so the script can change it. (It's really okay if you don't get this discussion. This example is probably the only time you'll see this trick because I show a better way in the next example.)
- ✓ **Use a DOM reference to change the style colors.** That long "trail of breadcrumbs" syntax (document.body.style.color) takes you all the way from the document through the body to the style and finally the color. It's tedious but thorough.
- ✓ **Set the foreground color to white.** You can change the color property to any valid CSS color value (a color name or a hex value). It's just like CSS because you are affecting the CSS.

✓ **Set the background color to peachpuff.** Again, this adjustment is just like setting CSS.

Of course, there's no good reason to write code. You will find that it's just as easy to build CSS as it is to write JavaScript. The advantage comes when you use the DOM dynamically to change the page's behavior after it has finished loading.

In Figure 4-12, the page is set up with the default white background color. It has two buttons on it, which should change the body's background color. Click the peachpuff button, and you see that it works, as verified in Figure 4-13.

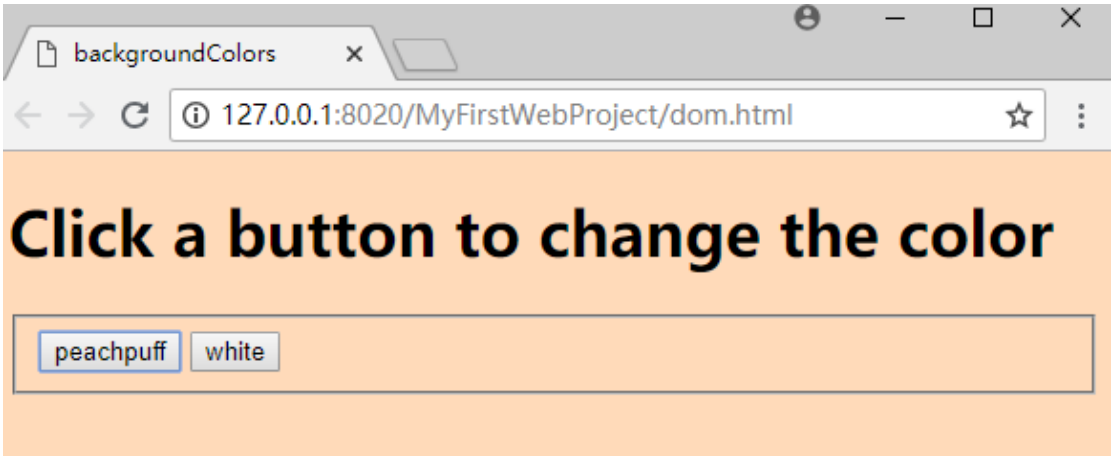


Figure 4-13 peachpuff background color

Some really exciting things just happened.

✓ **The page has a form.**

✓ **The buttons do something.** Plain-old HTML forms don't really do anything. You've got to write some kind of programming code to accomplish a task. This program does it. Twice. All for free.

✓ **Each button has a special attribute called onclick**: The onclick attribute is an event handler. This is special because it allows you to apply some sort of action to the button press. The action (a single line of JavaScript code) assigned to onclick will happen each time the button is clicked.

✓ **Each button changes the background to a different color**: The peachpuff button makes the background peachpuff and the White one — well, you get it. I simply used the code from the console example to change the background colors.

✓ **The code is integrated directly into the buttons**: You can attach one line of JavaScript code to a button's onclick event. I use that line to change the background colors.

4.2.3 Function

Functions are simply a collection of code lines with a name. Functions can also be sent an optional parameter, and they can return output. You learn much more about functions in section 3 of this chapter, but look at this basic version for now.

Something interesting is happening here. Take a look at how this program has changed from the first one.

- ✓ There's a function called makePeachpuff()in the script area. The function keyword allows you to collect one or more commands and give them a name. In this case, I'm giving that nasty document. body.style nonsense a much more sensible name —makePeachpuff ().
- ✓ The parentheses are necessary. Whenever you define a function, you have to include parentheses, but sometimes (as in this simple example), they're empty. You see how to add something to the parentheses in the next example.
- ✓ One or more lines of code go inside the function. Mark a function with squiggle braces ({}). This example has only one line of code in the function, but you can have as many code lines as you want.
- ✓ The function name describes what the function does. Functions are used to simplify code, so it's really important that a function name describes what the function does.
- ✓ Another function makes the background white. One button makes the background peachpuff, and the other makes it white, so I'll make a function to go with each button.
- ✓ Attach the functions to the buttons. Now the buttons each call a function rather than doing the work directly.

You might wonder if all this business of making a function is worth the effort — after all, these programs seem exactly the same — but the new one is a bit more work. In this very simple example, the functions are a little more work but clarify the code a smidge. As your programs get more complex, there's no doubt that functions improve things, especially as you learn more about how functions work.

The version of the code that uses functions doesn't seem a lot easier than adding the code directly, and it isn't. But functions are much more powerful than simply renaming a line of code. If you think about the two functions in that example, you quickly realize they're almost exactly the same. It would be awesome if you could write one simple function and have it change the background to any color you want. That's exactly what happens in the next example. Here's the code:

Example 4-10 flexible function

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>backgroundColors</title>
        <script type="text/javascript">
            function changeColor(color) {
                document.body.style.backgroundColor =color;
            }
        </script>
    </head>
```

```
    <body>
        <h1>Click a button to change the color</h1>
        <form >
            <fieldset>
                <input         type="button"         value="peachpuff"
onclick="changeColor('peachpuff')" />
                <input           type="button"           value="white"
onclick="changeColor('white')" />
            </fieldset>
        </form>
    </body>
</html>
```

Once again, this program will seem to the casual user to be exactly like the programs in Figure 4-12 and Figure 4-13, so I'm not including a screen shot. This is an important part of computer programming. Often the most important changes are not visible to the user. If you've ever hired a programmer, you're no doubt aware of this issue.

- ✓ The page has a single changeColor()function. The page has only one function called changeColor() defined in the header.
- ✓ The changeColor()function includes a color parameter. This time, there's a value inside the parentheses:

  ```
  function changeColor(color){
  ```

  The term color inside the parentheses is called a parameter. A parameter is a value sent to a function. Inside the function, color is available as a variable. When you call a function, you can send a value to it, like this:

  ```
  changeColor('white');
  ```

  This sends the text value 'white' to the function, where it becomes the value of the color variable.

  You'll sometimes see the terms argument and parameter used interchangeably to reference the stuff passed to a function, but these terms are not exactly the same. Technically, the parameter is the variable name (color) and the argument is the value of that variable ('white'). You can design a function with as many parameters as you wish, but you need to name each one. After a function is designed with parameters, you must supply an argument for each parameter when you call the function.
- ✓ Both buttons pass information to changeColor: Both of the buttons call the changeColor() function, but they each pass a different color value. This is one of the most useful characteristics of functions. They allow you to repeat code that's similar but not identical. That makes it possible to build very powerful functions that can be reused easily

Embedding quotes within quotes

Take a careful look at the onclick lines in the code in the preceding section. You may not have noticed one important issue:

onclick is an HTML parameter, and its value must be encased in quotes. The parameter happens to be a function call, which sends a string value. String values must also be in quotes. This setup can become confusing if you use double quotes everywhere because the browser has no way to know the quotes are nested. Look at this incorrect line of code:

```
onclick = "changeColor("white")" />
```

HTML thinks the onclick parameter contains the value "changeColor(" and it will have no idea what white")" is. Fortunately, JavaScript has an easy fix for this problem. If you want to embed a quote inside another quote, just switch to single quotes. The line is written with the parameter inside single quotes:

```
onclick = "changeColor('white')" />
```

Writing the changeColor function

The changeColor() function is pretty easy to write.

```
<script type = "text/javascript">
// from backgroundColors
function changeColor(color){
document.body.style.backgroundColor = color;
} // end changeColor
</script>
```

It goes in the header area as normal. It's simply a function accepting one parameter called color. The body's backgroundColor property is set to color.

I can write JavaScript in the header that refers to the body because the header code is all in a function. The function is read before the body is in place, but it isn't activated until the user clicks the button. By the time the user activates the code by clicking on the button, there is a body, and there's no problem.

4.2.4 Managing Text Input and Output

Perhaps the most intriguing application of the DOM is the ability to let the user communicate with the program through the web page, without all those annoying dialog boxes. Figure 4-14 shows a page with a web form containing two textboxes and a button.

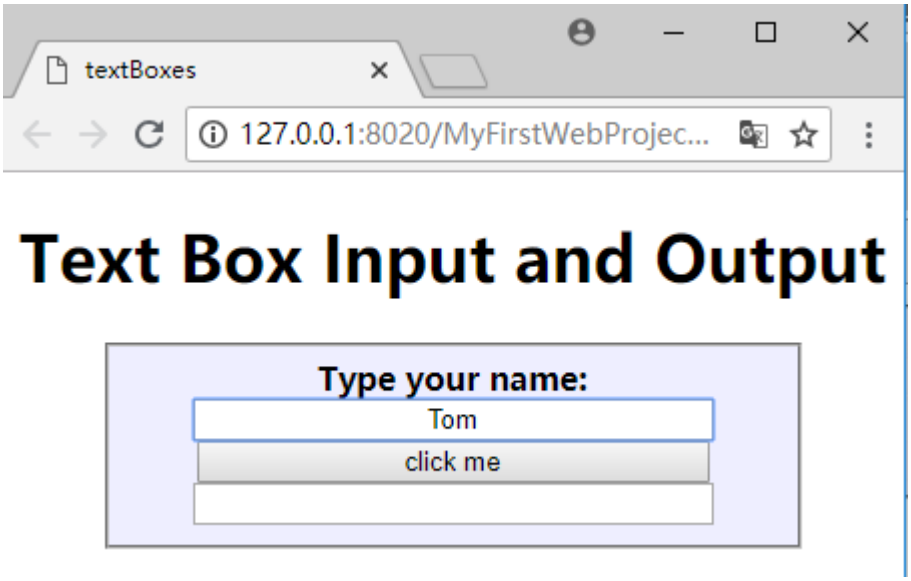


Figure 4-14 type the name into the textbook

When you click the button, something exciting happens, demonstrated by Figure 4-15.

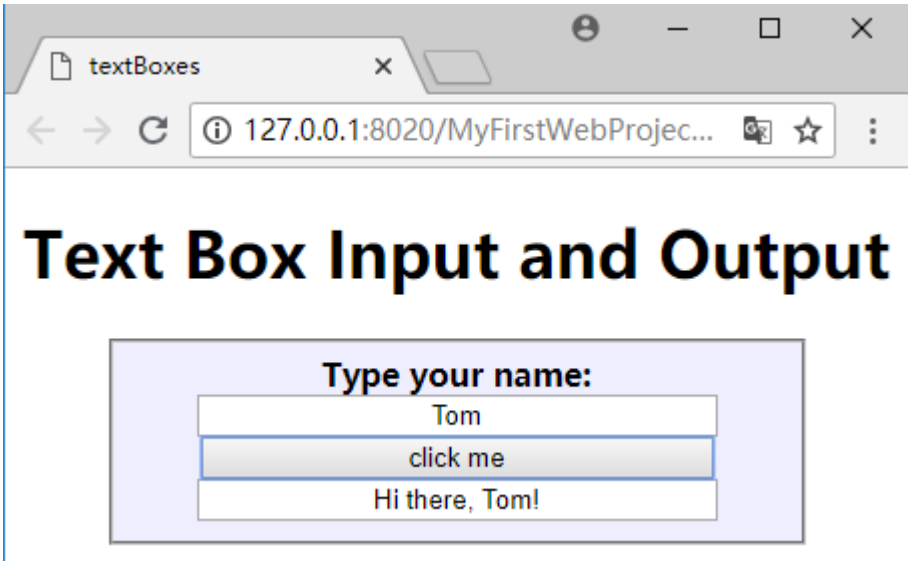


Figure 4-15 get the greeting from the textbook

Clearly, form-based input and output is preferable to the constant interruption of dialog boxes.

Introducing event-driven programming

Graphic user interfaces usually use a technique called event-driven programming. The idea is simple.

- ✓ Step 1. Create a user interface. In web pages, the user interface is usually built of HTML and CSS.
- ✓ Step 2. Identify events the program should respond to. If you have a button, users will click it. (If you want to guarantee they click it, put the text “Launch the Missiles” on the button. I don’t know why, but it always works.) Buttons almost always have events. Some other elements do,

too.

- ✓ Step 3. Write a function to respond to each event. For each event you want to test, write a function that does whatever needs to happen.
- ✓ Step 4. Get information from form elements. Now you're accessing the contents of form elements to get information from the user. You need a mechanism for getting information from a text field and other form elements.
- ✓ Step 5. Use form elements for output. For this simple example, I also use form elements for output. The output goes in a second textbox, even though I don't intend the user to type any text there.

The first step in building a program that can manage text input and output is to create the HTML framework. Here's the HTML code:

Example 4-11 manage input and output

```
<!DOCTYPE html>
<html>
<head>
<meta charset = "UTF-8">
<title>textBoxes</title>
<script type = "text/javascript">
function sayHi(){
var txtName = document.getElementById("txtName");
var txtOutput = document.getElementById("txtOutput");
var name = txtName.value;
txtOutput.value = "Hi there, " + name + "!"
} // end sayHi
</script>
<link rel="stylesheet" href="textButton.css" />
</head>
<body>
<h1>Text Box Input and Output</h1>
<form >
<fieldset>
<label>Type your name: </label>
<input type = "text" id = "txtName" />
<input type = "button" value = "click me"
onclick = "sayHi()"/>
<input type = "text" id = "txtOutput" />
</fieldset>
</form>
</body>
</html>
```

And here is the CSS file:

Example 4-12 CSS file of manage input and output

```
h1 {
  text-align: center;
}
fieldset{
  width: 20em;
  background-color: #EEEEFF;
  margin-left: auto;
  margin-right: auto;
}
input {
    display: block;
    margin-left: auto;
    margin-right: auto;
    width: 80%;
    text-align: center;
}
label {
  display: block;
  font-weight: bold;
  margin-left: auto;
  margin-right: auto;
  width: 100%;
  text-align: center;
}
```

As you look over the code, note a few important ideas:

- ✓ The page uses external CSS. The CSS style is nice, but it's not important in the discussion here. It stays safely encapsulated in its own file. Of course, you're welcome to look it over or change it.
- ✓ Most of the page is a form. All form elements must be inside a form.
- ✓ A fieldset is used to contain form elements. input elements need to be inside some sort of block-level element, and a fieldset is a natural choice.
- ✓ There's a text field named txtName. This text field contains the name. I begin with the phrase txt to remind myself that this field is a textbox.
- ✓ The second element is a button. You don't need to give the button an ID (as it won't be referred to in code), but it does have an onclick() event.
- ✓ The button's onclick event refers to a (yet undefined) function. In this example, it's named "sayHi()".

- ✓ A second textbox contains the greeting. This second textbox is called txtOutput because it's the text field meant for output.

After you set up the HTML page, the function becomes pretty easy to write because you've already identified all the major constructs. You know you need a function called sayHi(), and this function reads text from the txtName field and writes to the txtOutput field.

Using getElementById to get access to the page

HTML is one thing, and JavaScript is another. You need some way to turn an HTML form element into something JavaScript can read. The magical getElementById() method does exactly that. First, look at the first two lines of the sayHi() function (defined in the header as usual).

```
function sayHi(){
var txtName = document.getElementById("txtName");
var txtOutput = document.getElementById("txtOutput");
```

You can extract every element created in your web page by digging through the DOM. Modern browsers have the wonderful getElementById() function instead. This beauty searches through the DOM and returns a reference to an object with the requested ID.

A reference is simply an indicator where the specified object is in memory. You can store a reference in a variable. Manipulating this variable manipulates the object it represents. If you want, you can think of it as making the textbox into a variable.

Note that I call the variable txtName, just like the original textbox. This variable refers to the text field from the form, not the value of that text field. After I have a reference to the text field object, I can use its methods and properties to extract data from it and send new values to it.

Manipulating the text fields

After you have access to the text fields, you can manipulate the values of these fields with the value property:

```
var name = txtName.value;
txtOutput.value = "Hi there, " + name + "!"
```

Text fields (and, in fact, all input fields) have a value property. You can read this value as an ordinary string variable. You can also write to this property, and the text field will be updated on the fly.

This code handles the data input and output:

1. Create a variable for the name. This is an ordinary string variable.

2. Copy the value of the textbox into the variable. Now that you have a variable representing the textbox, you can access its value property to get the value typed in by the user.

3. Create a message for the user. Use ordinary string concatenation.

4. Send the message to the output textbox. You can also write text to the value property, which changes the contents of the text field on the screen. Text fields always return string values (like prompts do). If you want to pull a numeric value from a text field, you may have to convert it with the parseInt() or parseFloat() functions.

Writing to the Document

Form elements are great for getting input from the user, but they're not ideal for output. Placing the output in an editable field really doesn't make much sense. Changing the web document is a much better approach.

The DOM supports exactly such a technique. Most HTML elements feature an innerHTML property. This property describes the HTML code inside the element. In most cases, it can be read from and written to.

So what are the exceptions? Single-element tags (like <img> and <input>) don't contain any HTML, so obviously reading or changing their inner HTML doesn't make sense. Table elements can often be read from but not changed directly.

Figure 4-16 shows a program with a basic form.

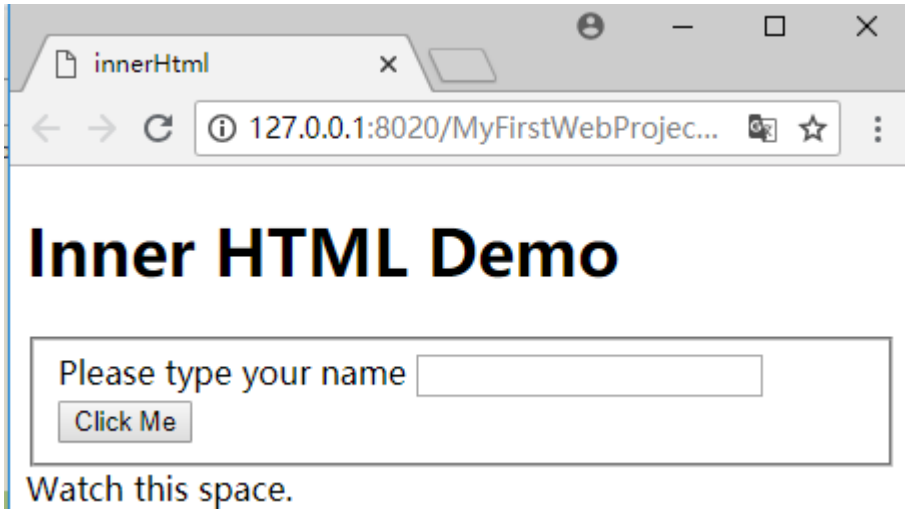


Figure 4-16 innerHTML

This form doesn't have a form element for the output. Enter a name and clickthe button, and you see the results in Figure 4-17

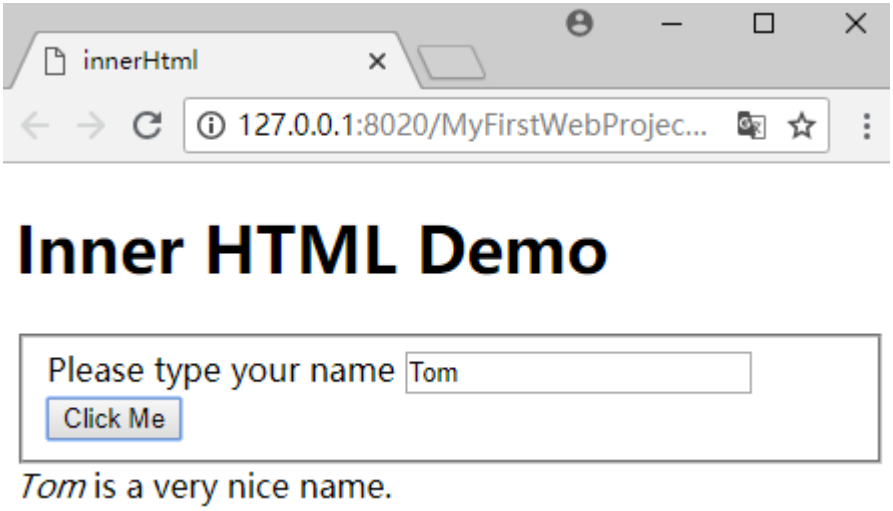


Figure 4-17 output by innerHTML

Amazingly enough, this page can make changes to itself dynamically. It isn't simply changing the values of form fields, but changing the HTML.

Preparing the HTML framework

To see how the page changes itself dynamically, begin by looking at the HTML body for innerHTML.html:

Example 4-13 HTML framework for innerHTML

```
<body>
<h1>Inner HTML Demo</h1>
<form>
<fieldset>
<label>Please type your name</label><p>
<input type = "text" id = "txtName" />
<button type = "button" onclick = "sayHi()"> Click Me </button>
</fieldset>
</form>
<div id = "divOutput">
Watch this space.
</div>
</body>
```

The code body has a couple of interesting features:

- ✓ The program has a form. The form is pretty standard. It has a text field for input and a button, but no output elements.
- ✓ The button will call a sayHi()function. The page requires a function with this name. Presumably, it says hi somehow.
- ✓ There's a div for output. A div element in the main body is designated for output.
- ✓ The div has an ID. The id attribute is often used for CSS styling, but the DOM can also use it. Any HTML elements that will be dynamically scripted should have an id field.

Writing the JavaScript

The JavaScript code for modifying innerHTML isn't very hard:

Example 4-14 script for writing innerHTML

```
<script type = "text/javascript">
function sayHi(){
txtName = document.getElementById("txtName");
divOutput = document.getElementById("divOutput");
name = txtName.value;
divOutput.innerHTML = "<em>" + name + "</em>";
```

```
divOutput.innerHTML += " is a very nice name.";
}
</script>
```

The first step (as usual with web forms) is to extract data from the input elements. Note that I can create a variable representation of any DOM element, not just form elements. The divOutput variable is a JavaScript representation of the DOM div.

Finding your innerHTML

Like form elements, divs have other interesting properties you can modify. The innerHTML property allows you to change the HTML code displayed by the div. You can put any valid HTML code you want inside the innerHTML property, even HTML tags. Be sure that you still follow the HTML rules so that your code will be valid.

4.2.5 Working with Other Text Elements

When you know how to work with text fields, you've mastered about half of the form elements. Several other form elements work exactly like text fields, including these:

- ✓ Password fields obscure the user's input with asterisks, but preserve the text.
- ✓ Hidden fields allow you to store information in a page without revealing it to the user. (They're used a little bit in client-side coding, but almost never in JavaScript.)
- ✓ Text areas are a special variation of textboxes designed to handle multiple lines of input.

Figure 4-18 is a page with all these elements available on the same form.

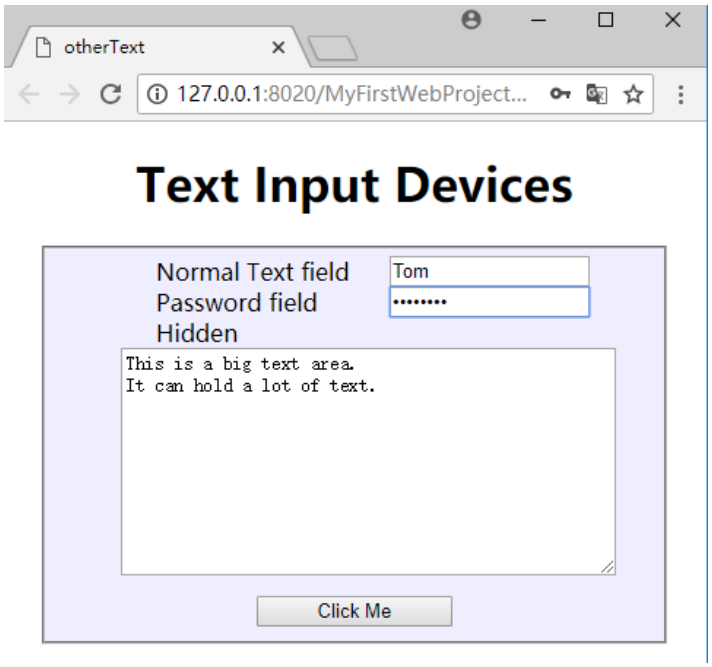


Figure 4-18 other elements

When the user clicks the button, the contents of all the fields (even the password and hidden fields)

appear on the bottom of the page, as shown in Figure 4-19 .

Figure 4-19 JavaScript in other elements

Building the form

Here's the HTML (otherText.html) that generates the form shown in Figure 4-18 and Figure 4-19:

Example 4-15 other elements form

```
<body>
    <h1>Text Input Devices</h1>
    <form>
      <fieldset>
        <label>Normal Text field</label>
        <input type = "text" id = "txtNormal" />
        <label>Password field</label>
        <input type = "password" id = "pwd" />
        <label>Hidden</label>
        <input type = "hidden" id = "hidden"
              value = "I can't tell you" />
        <textarea id = "txtArea"rows = "10" cols = "40">
```

```
            This is a big text area.
            It can hold a lot of text.
      </textarea>
      <button type = "button"  onclick = "processForm()">
        Click Me </button>
    </fieldset>
   </form>
  <div id = "output">
  </div>
  </body>
```

And here is the CSS of the form.

Example 4-16 CSS file of other elements form

```
h1 {
  text-align: center;
}

fieldSet{
  background-color: #EEEEFF;
  width: 25em;
  margin-left: auto;
  margin-right: auto;
}

label {
  float: left;
  width: 10em;
  clear: left;
  margin-left: 4em;
}

input {
  float: left;
  width: 10em;
}

textArea {
  float: left;
  clear: left;
  width: 25em;
  margin-left: 3em;
}

button {
```

```
  float: left;
  clear: left;
  width: 10em;
  margin-left: 10em;
  margin-top: 1em;
}

dt {
  font-weight: bold;
  border-top: 3px black groove;
}
```

The code may be familiar to you if you read about form elements. A few things are worth noting for this example:

- ✓ An ordinary text field appears, just for comparison purposes. It has an id so that it can be identified in the JavaScript.
- ✓ The next field is a password field. Passwords display asterisks, but store the actual text that was entered. This password has an id of pwd.
- ✓ The hidden field is a bit strange. You can use hidden fields to store information on the page without displaying that information to the user. Unlike the other kinds of text fields, the user can't modify a hidden field. (She usually doesn't even know it's there.) This hidden field has an id of secret and a value ("I can't tell you").
- ✓ The text area has a different format. The input elements are all singletag elements, but the textarea is designed to contain a large amount of text, so it has beginning and end tags. The text area's id is txtArea.
- ✓ A button starts all the fun. As usual, most of the elements just sit there gathering data, but the button has an onclick event associated with it, which calls a function.
- ✓ External CSS gussies it all up. The page has some minimal CSS to clean it up. The CSS isn't central to this discussion, so I don't reproduce it. Note that the page will potentially have a dl on it, so I have a CSS style for it, even though it doesn't appear by default.

Writing the function

After you build the form, all you need is a function. Here's the good news: JavaScript treats all these elements in exactly the same way! The way you handle a password, hidden field, or text area is identical to the technique for a regular text field. Here's the code:

```
<script type = "text/javascript">
     function processForm(){
       //grab input from form
       var txtNormal = document.getElementById("txtNormal");
       var pwd = document.getElementById("pwd");
```

```
        var hidden = document.getElementById("hidden");
        var txtArea = document.getElementById("txtArea");
        var normal = txtNormal.value;
        var password = pwd.value;
        var secret = hidden.value;
        var bigText = txtArea.value;
        //create output
        var result = ""
        result += "<dl> \n";
        result += "  <dt>normal<\/dt> \n";
        result += "  <dd>" + normal + "<\/dd> \n";
        result += " \n";
        result += "  <dt>password<\/dt> \n";
        result += "  <dd>" + password + "<\/dd> \n";
        result += " \n";
        result += "  <dt>secret<\/dt> \n";
        result += "  <dd>" + secret + "<\/dt> \n";
        result += "   \n";
        result += "  <dt>big text<\/dt> \n";
        result += "  <dd>" + bigText + "<\/dt> \n";
        result += "<\/dl> \n";
        var output = document.getElementById("output");
        output.innerHTML = result;
      } // end function
</script>
```

The function is a bit longer than the others in this chapter, but it follows exactly the same pattern: It extracts data from the fields, constructs a string for output, and writes that output to the innerHTML attribute of a div in the page.

The code has nothing new, but it still has a few features you should consider:

- ✓ Create a variable for each form element. Use the document.getElementById mechanism.
- ✓ Create a string variable containing the contents of each element. Don't forget: The getElementById trick returns an object. You need to extract the value property to see what's inside the object.
- ✓ Make a big string variable to manage the output. When output gets long and messy like this one, concatenate a big variable and then just output it in one swoop.
- ✓ HTML is your friend. This output is a bit complex, but innerHTML is HTML, so you can use any HTML styles you want to format your code. The return string is actually a complete definition list. Whatever is inside the textbox is (in this case) reproduced as HTML text, so if I want carriage returns or formatting, I have to add them with code.

- ✓ Newline characters (\n) clean up the output. If I were writing an ordinary definition list in HTML, I'd put each line on a new line. I try to make my programs write code just like I do, so I add newline characters everywhere I'd add a carriage return if I were writing the HTML by hand.

Understanding generated source

When you run the program in the preceding section, your JavaScript code actually changes the page it lives on. The code that doesn't come from your server (but is created by your program) is sometimes called generated source. The generated code technique is powerful, but it can have a significant problem. Try this experiment to see what I mean:

- ✓ Reload the page. You want to view it without the form contents showing so that you can view the source. Everything will be as expected; the source code shows exactly what you wrote.
- ✓ Click the Click Me button. Your function runs, and the page changes. You clearly added HTML to the output div because you can see the output right on the screen.
- ✓ View the source again. You'll be amazed. The output div is empty, even though you can clearly see that it has changed.
- ✓ Check generated code. Using the HTML validator extension or the W3 validator doesn't check for errors in your generated code. You have to check it yourself, but it's hard to see the code!

4.3 Control Structure and Debugging

Computer programs are complex. They involve information. The other key component of programming is control — that is, managing the instructions needed to solve interesting complex problems. In this section, you will learn the key control structures — if statements and looping structures. With increased control comes increased opportunity for error, so you also learn how to manage problems in your code.

4.3.1 Making Choices with if

Sometimes you'll need your code to make decisions. For example, if somebody famous typed their name in your website, you might want to create a custom greeting for them. Take a look at the Example 4-17 in Figure 4-20 and Figure 4-21.

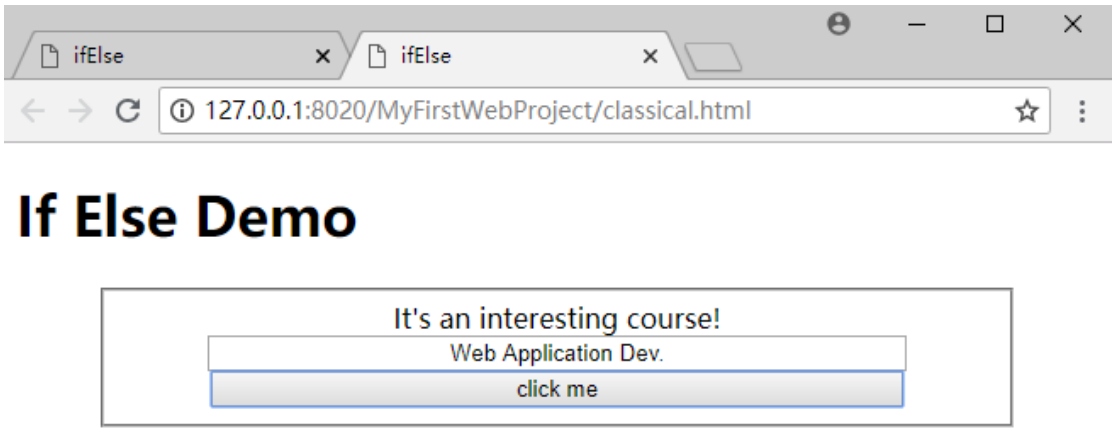


Figure 4-20 well, it's web dev course

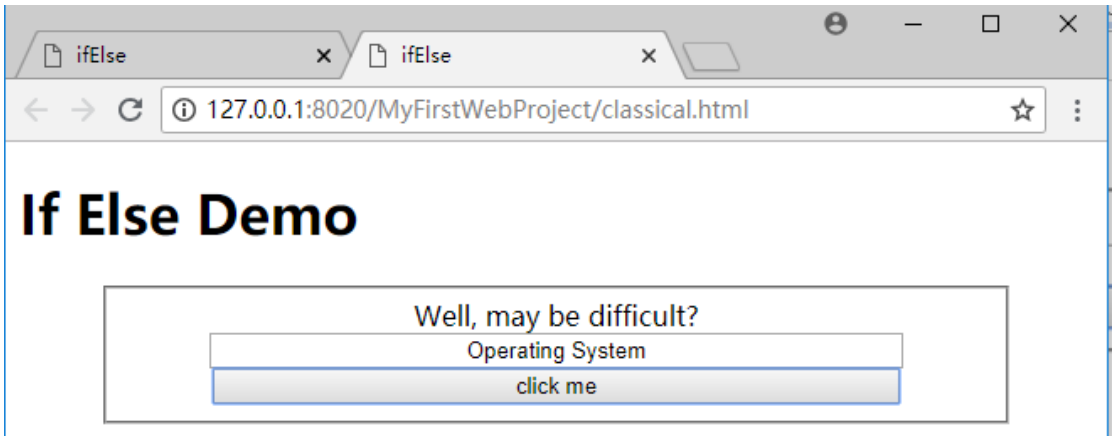


Figure 4-21 other course

Example 4-17 the first if statement

```
<!DOCTYPE HTML>
<html>
<head>
  <title>ifElse</title>
  <meta charset = "UTF-8" />
  <script type = "text/javascript">
    function checkName(){
      lblOutput = document.getElementById("lblOutput");
      txtInput = document.getElementById("txtInput");
      courseName = txtInput.value;
      if (courseName == "Web Application Dev."){
        lblOutput.innerHTML = "It's an interesting course!";
      } else {
```

```
      lblOutput.innerHTML = "Well, may be difficult?";
    } // end if
  } // end function
 </script>
 <style>
  label, input, button {
    display: block;
    width: 80%;
    margin: auto;
    text-align: center;
  }
  fieldset {
    width: 80%;
    margin: auto;
  }
 </style>
</head>

<body>
 <h1>If Else Demo</h1>
 <form >
  <fieldset>
    <label id = "lblOutput">Please enter course name</label>
    <input type = "text" id = "txtInput"
          value = "Web Application Dev." />
    <button type = "button"onclick = "checkName()">click me
    </button>
  </fieldset>
 </form>
</body>
</html>
```

This program (and the next few) uses a basic HTML set up to take information from a text field, respond to a button click, and print output in a designated area. As you can see, the program looks at the input in the text box and changes behavior based on the value of the text field.

Changing the greeting with if

This code uses an important idea called a **condition** inside a construct called an if statement. Here's how to do it:

- ✓ Set up the web page as usual. The HTML code has elements called lblOutput and txtInput. It also has a button that calls checkName() when it is clicked.
- ✓ Create variables for important page elements. You're getting data from txtInput and changing the HTML code in lblOutput, so create variables for these two elements.

- ✓ Get userName from txtInput. Use the txtInput.value trick to get the value of the input element called txtInput and place it in the variable courseName.
- ✓ Set up a condition. The key to this program is a special element called a condition — an expression that can be evaluated as true or false. Conditions are often (as in this case) comparisons. Note that the double equals sign (==) is used to represent equality. In this example, I'm asking whether the courseName variable equals the value "Web Application Dev.".
- ✓ Place the condition in an if structure. The if statement is one of a number of programming constructs which use conditions. It contains the keyword if followed by a condition (in parentheses). If the condition is true, all of the code in the following set of braces is executed.
- ✓ Write code to execute if the condition is true. Create a set of squiggly braces after the condition. Any code inside these braces will execute if the condition is true. Be sure to indent your code, and use the right squiggle brace (}) to end the block of code. In this example, I give a special greeting to Web Application Dev. (because This course is web dev.).
- ✓ Build an else clause. You can build an if statement with a single code block, but often you want the code to do something else if the condition was false. Use the else construct to indicate you will have a second code block that will execute only if the condition is false.
- ✓ Write the code to happen when the condition is false. The code block following the else clause will execute only if the condition is false. In this particular example, I have a greeting for every course except Web Application Dev.

4.3.2 The different if statement

If statements are extremely powerful, and there are a number of variations. You can actually have one, two, or any number of branches. You can write code like this:

```
if (courseName == " Web Application Dev."){
lblOutput.innerHTML = "It's an interesting course!"
} // end if
```

With this structure, the greeting will occur if courseName is "Web Application Dev." and nothing will happen if the courseName is anything else. You can also use the if-else structure (this is the form used in the actual code):

```
if (courseName == " Web Application Dev."){
lblOutput.innerHTML = "It's an interesting course!";
} else {
lblOutput.innerHTML = "Well, may be difficult?";
} // end if
```

One more alternative lets you compare as many results as you wish by adding new conditions:

```
if (courseName == "Web Application Dev."){
lblOutput.innerHTML = "It's an interesting course!";
} else if (courseName == "Java Program") {
```

```
lblOutput.innerHTML = "Well, may be difficult?";
} else if (courseName == "Operating System") {
lblOutput.innerHTML = "Basic Course!";
```

Conditional operators

The == operator checks to see if two values are identical, but JavaScript supports a number of other operators as well:

Table 4-3 conditional operator

Operator	Meaning
a == b	a is equal to b
a < b	a is less than b
a > b	a is greater than b
a <= b	a is less than or equal to b
a >= b	a is greater than or equal to b
a != b	a is not equal to b

If you're coming from another programming language like Java, C++, or PHP you might wonder how string comparisons work because they require different operators in these languages. JavaScript uses exactly the same comparison operators for types of data, so there's no need to learn different operators. Yeah, JavaScript!

4.3.3 Nesting your if statements

There are a few other variations of the if structure you'll sometimes run across. One variation is the nested if statement. This simply means you can put if statements inside each other for more complex options. For example, look at the following code:

Example 4-18 nested if statement

```
function checkTemp() {
    //from nestedIf.html
    var temp = prompt("What temperature is it outside?");
    temp = parseInt(temp);
    if(temp < 60) {
        //less than 60
        if(temp < 32) {
            /less than 32
```

```
            alert("It's freezing!");
        } else {
            //between 32 and 60
            alert("It's cold.");
        } // end 'freezing if'
    } else {
        //We're over 60
        if(temp < 75) {
            //between 60 and 75
            alert("It's cool.");
        } else {
            //temp is higher than 75
            if(temp > 90) {
                //over 90
                alert("It's really hot.");
            } else {
                //between 75 and 90
                alert("It's warm.");
            } // end 'over 90' if
        } // end 'over 75' if
    } // end 'over 60' if
} // end function
```

This code looks complicated, but it really isn't. It simply takes in a temperature and looks for a range of values. Here's what's happening:

- ✓ **Get a temperature value from the user.** Ask the user for a temperature. I'm using the simple prompt statement here, but you could also grab the value from a form field.
- ✓ **Convert the temperature to a numeric type.** Recall that computers are fussy about data types, and sometimes you need to nudge a variable to the right type. The parseInt() function forces any value into an integer, which is perfect for our needs.
- ✓ **Use an if statement to chop the possibilities in half.** The outer (most encompassing) if statement separates all the cooler temperatures from the warmer ones.
- ✓ **Use an inner if statement to clarify more if needed.** Within the cool (less than 60 degree) temperatures, you might want to know if it's cold or below freezing, so place a second condition to determine the temperatures.
- ✓ **The upper bound is determined by the outer if statement.** The first else clause in the code is triggered when the temperature is between 32 and 60 degrees because it's inside two if statements: temp < 60 is true, and temp < 32 is false, so the temperature is between 32 and 60 degrees.
- ✓ **Indentation and comments are not optional.** As the code becomes more complex, indentation and comment characters become more critical. Make sure your indentation accurately reflects the beginning and end of each if statement, and the code is clearly

commented so you know what will happen (or what you expect will happen — the truth may be different).

- ✓ **You can nest as deeply as you wish.** As you can see in this structure, there are three different possibilities for temperatures higher than 60 degrees. Simply add more if statements to get the behavior you wish.
- ✓ **Test your code.** When you build this kind of structure, you need to run your program several times to ensure it does what you expect.

4.3.4 Making decisions with switch

JavaScript, like a number of languages, supports another decision-making structure called switch. This is a useful alternative when you have a number of discrete values you want to compare against a single variable. Take a look at this variation of the name program from earlier in this chapter:

Example 4-19 switch statement

```
function checkName(){
var name = prompt("Input your course name?");
switch(name){
case " Web Application Dev.":
alert("It's an interesting course!");
break;
case " Java Program ":
alert("Well, may be difficult?");
break;
default:
alert("Basic Course!");
} // end
} // end checkName
```

The switch code is similar to an if-else if structure in its behavior, but it uses a different syntax:

- ✓ **Indicate a variable in the switch statement.** In the switch statement's parentheses, place a variable or other expression. The switch statement is followed by a code block encased in squiggly braces ({}).
- ✓ **Use the case statement to indicate a case.** The case statement is followed by a potential value of the variable, followed by a colon. It's up to the programmer to ensure the value type matches the variable type.
- ✓ **End each case with the break statement.** End each case with the break statement. This indicates that you're done thinking about cases, and it's time to pop out of this data structure. If you don't explicitly include the break statement, you'll get strange behavior (all the subsequent cases will evaluate true as well).
- ✓ **Define a default case to catch other behavior.** Just like you normally add a default else to an

if–elseif structure to catch any unanticipated values, the default keyword traps for any values of the variable that were not explicitly caught.

A number of the newer languages (like Python) don't support switch at all, so at some point you're likely to be in a language that cannot do switch. You might as well learn alternatives now. For these reasons, I rarely use switch in my own programming

4.3.5 Managing Repetition with for Loops

Computers are well-known for repetitive behavior. It's pretty easy to get a computer to do something many times. The main way to get this behavior is to use a mechanism called a **loop**. The **for** loop is a standard kind of loop that is used when you know how often something will happen. Figure 4-22 and Figure 4-23 shows the most basic form of the for loop:

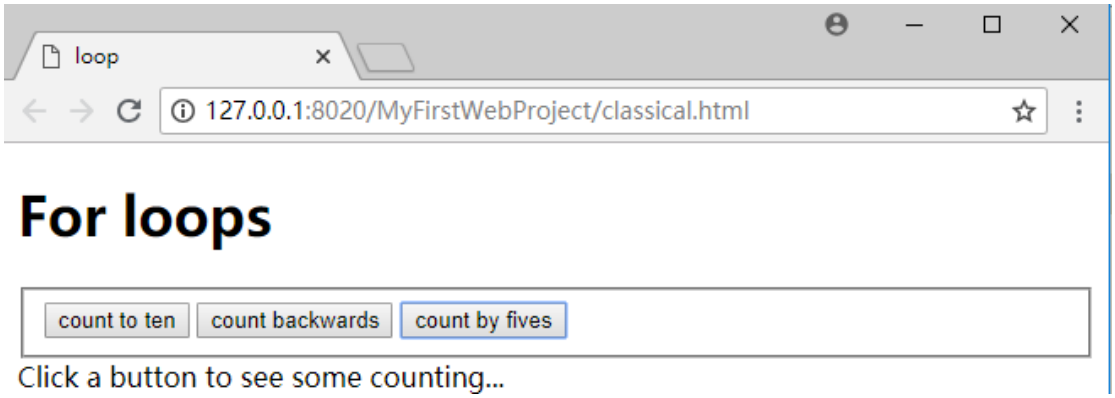


Figure 4-22 the first for loop

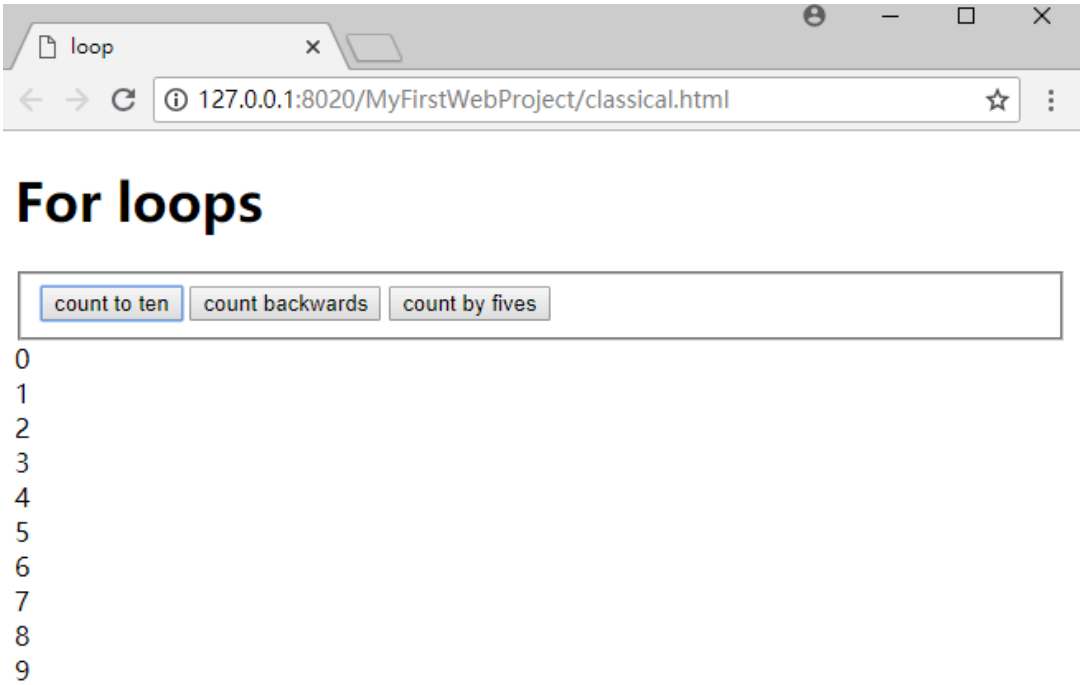


Figure 4-23 result of the first for loop

Setting up the web page

The same web page is used to demonstrate three different kinds of for loops. As usual, the HTML code sets everything up. Here's the HTML code that creates the basic framework:

Example 4-20 HTML code of the first for loop

```
<body onload="init()">
      <h1>For loops</h1>
      <form>
         <fieldset>
            <button type="button" onclick="count()">
                  count to ten</button>
            <button type="button" onclick="back()">
                  count backwards</button>
            <button type="button" onclick="byFive()">
                  count by fives</button>
         </fieldset>
      </form>
     <div id="output">Click a button to see some counting...</div>
   </body>
```

While the HTML is pretty straightforward, it does have some important features:

- ✓ **The body calls an initialization function.** Often you'll want some code to happen when the page first loads. One common way to do this is to attach a function call to the onload attribute of the body element. In this example, I call the init() function as soon as the body is finished loading. The contents of the init() function will be described in the next section.
- ✓ **The page is mostly an HTML form.** The most important part of this page is the form with three buttons on it. Each button calls a different JavaScript function.
- ✓ **A special div is created for output.** It's a good idea to put some default text in the div so you can see where the output should go and so you can ensure the div is actually changing when it's supposed to.

From this example, it's easy to see why it's a good idea to write the HTML first. The HTML code gives me a solid base for the program, and it also provides a good outline of what JavaScript code I'll need. Clearly this page calls for four JavaScript functions, init(), count(), back(), and byFive(). The names of all the functions are pretty self-explanatory, so it's pretty easy to see what each one is supposed to do. It's also clear that the div named output is intended as an output area. When you design the HTML page well, the JavaScript code becomes very easy to start.

Initializing the output

This program illustrates a situation that frequently comes up in JavaScript programming: All three of the main functions will refer to the same output area. It seems a waste to create a variable for output three different times. Instead, I make a single global output variable available to all functions, and attach the variable to that element once when the page loads. In order to understand why this is

necessary, it's important to discuss an idea called variable scope. Generally, variables are created inside functions. As long as the function is running, the variable still exists. However, when a function is done running, all the variables created inside that function are instantly destroyed. This prevents functions from accidentally changing the variables in other functions. Practically, it means you can think of each function as a separate program. However, sometimes you want a variable to live in more than one function. The output variable in the forLoop.html page is a great example because all of the functions will need it. One solution is to create the variable outside any functions. Then all the functions will have access to it.

You can create the output variable without being in a function, but you can't attach it to the actual div in the web page until the web page has finished forming. The init() function is called when the body loads. Inside that function, I assign a value to the global output variable. Here's how the main JavaScript and the init() method code looks:

```
var output;
function init(){
output = document.getElementById("output");
} // end init
```

This code creates output as a global variable, and then attaches it to the output div after the page has finished loading.

Creating the basic for loop

The standard for loop counts the values between 1 and 10. The "count to ten" button triggers the count() function.

Here's the code for count():

```
function count(){
output.innerHTML = "";
for (i = 1; i <= 10; i++){
output.innerHTML += i + "<br />";
} // end for loop
} // end count
```

Although the count() function clearly prints ten lines, it only has one line that modifies the output div. The main code repeats many times to create the long output.

- ✓ Step 1. You can use the output var immediately. Because output is a global variable and it has already been created, you can use it instantly. There's no need to initialize it in the function.
- ✓ Step 2. Clear the output. Set output.value to the empty string ("") to clear the output. This will destroy whatever text is currently in the div.
- ✓ Step 3. Start a for loop. The for loop is a special loop used to repeat something a certain number of times. For loops have three components: initialization, comparison, and update.
- ✓ Step 4. Initialize your counting variable. A for loop works by changing the value of an integer many times. The first part of a for loop initializes this variable (often called i) to a starting value (usually 0 or 1).

- ✓ Step 5. Specify a condition for staying in the loop. The second part of a for statement is a condition. As long as the condition is true, the loop will continue. As soon as the condition is evaluated as false, the loop will exit.
- ✓ Step 6. Change the variable. The third part of a for statement somehow changes the counting variable. The most common way to change the variable is to add one to it. The i++ syntax is a shortcut for "add one to i."
- ✓ Step 7. Build a code block for repeated code. Use braces and indentation to indicate which code repeats. All code inside the braces repeats.
- ✓ Step 8. Inside the loop, write to the output. On each iteration of the loop, add the current value of i to the output div's innerHTML. Also add a break (<br />) to make the output look better. When you add to an innerHTML property, you're writing HTML code, so if you want the output to occur on different lines, you need to write the HTML to make this happen. (See the section "Introducing shortcut operators" in this chapter for an explanation of the += statement.)
- ✓ Step 9. Close the loop. Don't forget to end the loop, or your program will not run correctly

You might have noticed a couple of new operators in the code. These are some shortcut tools that allow you to express common ideas more compactly. For example, consider the following code:

i = i + 1;

This means, "Add one to i, and store the result back in i." It's a pretty standard statement, even if it does drive algebra teachers bananas. The statement is so common that it is often abbreviated, like this:

i += 1;

This statement means exactly the same as the last one; add one to i. You can use this to add any amount to the variable i. Because the + sign is used to concatenate (combine) strings, you can use the += shortcut with string manipulation, so consider this variation:

Counting backwards

After you understand basic for loops, it's not difficult to make a loop that counts backwards. Here's the back() function (called by the Count Backwards button):

```
function back(){
output.innerHTML = "";
for (i = 10; i > 0; i--){
output.innerHTML += i + "<br />";
} // end for loop
} // end back
```

When the user activates this function, she gets the result shown in Figure 4-24.

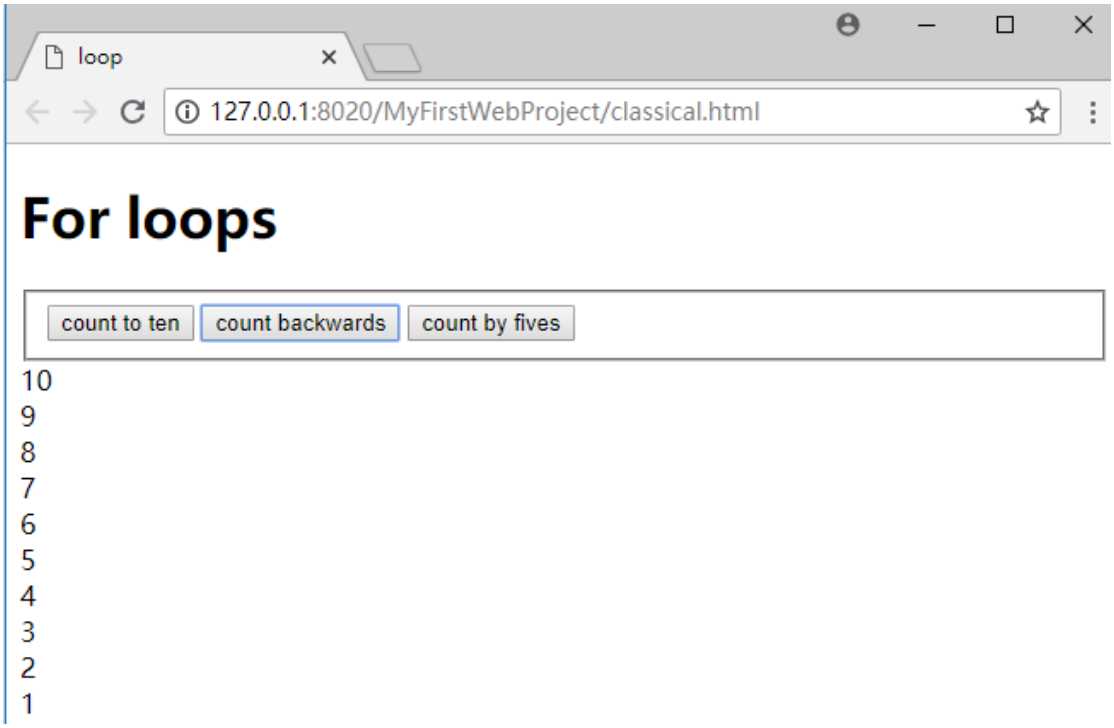


Figure 4-24 count backwards of the first for loop

This code is almost exactly like the first loop, but look carefully at how the loop is created:

- ✓ Step 1. Initialize i to a high value. This time I want to count backwards from 10 to 1, so start i with the value 10.
- ✓ Step 2. Keep going as long as i is greater than 0. It's important to note that the logic changes here. If i is greater than 0, the loop should continue. If i becomes 0 or less, the loop exits.
- ✓ Step 3. Subtract 1 from i on each pass. The -- operator works much like ++, but it subtracts 1 from the variable.

Counting by fives

Counting by fives (or any other value) is pretty trivial after you know how for loops work. Here's the byFive() code called by the Count by Five button:

```
function byFive(){
output.innerHTML = "";
for (i = 5; i <= 25; i += 5){
output.innerHTML += i + "<br />";
} // end for loop
} // end byFive
```

It is remarkably similar to the other looping code you've seen.

- ✓ Step 1. Initialize i to 5. The first value I want is 5, so that is the initial value for i.
- ✓ Step 2. Continue as long as i is less than or equal to 25. Because I want the value 25 to appear, I set the condition to be less than or equal to 25.

✓ Step 3. Add 5 to i on each pass. Each time through the loop, I add 5 to i using the += operator.

Here is code of the for loop:

Example 4-21 all code of for loop

```
<!DOCTYPE HTML>
<html>

   <head>
      <title>loop</title>
      <meta charset="UTF-8" />
      <script type="text/javascript">
         var output;

         function init() {
            output = document.getElementById("output");
         } // end init

         function count() {
            output.innerHTML="";
            for(i=0;i<10;i++)
            {
               output.innerHTML += i+"<br/>";
            }

         } // end count

         function back() {
            output.innerHTML = "";
            for(i = 10; i > 0; i--) {
               output.innerHTML += i + "<br />";
            } // end for loop
         } // end back

         function byFive() {
            output.innerHTML = "";
            for(i = 5; i <= 25; i += 5) {
               output.innerHTML += i + "<br />";
            } // end for loop
         } // end byFive
      </script>
   </head>

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

```
        <h1>For loops</h1>
        <form action="">
            <fieldset>
                <button type="button" onclick="count()">
                        count to ten</button>
                <button type="button" onclick="back()">
                        count backwards</button>
                <button type="button" onclick="byFive()">
                        count by fives</button>
            </fieldset>
        </form>
       <div id="output">Click a button to see some counting...</div>
    </body>

</html>
```

4.3.6 Building while Loops

For loops are useful when you know how often a loop will continue, but sometimes you need a more flexible type of loop. The while loop is based on a simple idea. It contains a condition. When the condition is true, the loop continues; if the condition is evaluated as false, the loop exits.

Making a basic while loop

Figure 4-25 shows a dialog box asking for a password. The program keeps asking for a password until the user enters the correct password.

Figure 4-25 while loop

Here's the HTML code used for two different while examples:

Example 4-22 HTML code of while loop

```
<body>
        <h1>While Loop Demo</h1>
```

```
      <p>The password is 'HTML5'</p>
      <form>
         <fieldset>
            <button type="button" onclick="getPassword()">
               guess the password</button>
            <button type="button" onclick="threeTries()">
               guess the password in three tries</button>
         </fieldset>
      </form>
   </body>
```

The version shown in Figure 4-25 keeps popping up a dialog box until the user gets the answer correct.

Example 4-23 getPassword of while loop

```
function getPassword(){
//from while.html
var correct = "HTML5";
var guess = "";
while (guess != correct){
guess = prompt("Password?");
} // end while
alert("You may proceed");
} // end getPassword
```

A while loop for passwords is not hard to build:

- ✓ **Step 1. Store the correct password in a variable.** Variable names are important because they can make your code easier to follow. I use the names correct and guess to differentiate the two types of password. Beginners often call one of these variables password, but that can be confusing because there are actually two passwords (the correct password and the guessed password) in play here. The best way to design variable names is to anticipate the conditions they will be used in. This function is based on the condition guess == correct. This is a really nice condition because it's really easy to determine what we're trying to figure out (whether the guess is correct). It takes some practice to anticipate variable names well, but it's a habit well worth forming.

- ✓ **Step 2. Initialize the guess to an empty value.** The key variable for this loop is guess. It starts as an empty string. It's critical to initialize the key variable before the loop begins.

- ✓ **Step 3. Set up the while statement.** The while statement has extremely simple syntax: the keyword while followed by a condition, followed by a block of code.

- ✓ **Step 4. Build the condition.** The condition is the heart of a while loop. The condition must be constructed so the loop happens at least once (ensure this by comparing the condition to the variable initialization). When the condition is true, the loop continues. When the condition is evaluated to false, the loop will exit. This condition compares guess to correct. If guess is not equal to correct, the code will continue.

- ✓ **Step 5. Write the code block.** Use braces and indentation to indicate the block of code that will be repeated in the loop. The only code in this particular loop asks the user for a password.
- ✓ **Step 6. Add code to change the key variable inside the loop.** Somewhere inside the loop, you need code that changes the value of the key variable. In this example, the prompt statement changes the password. As long as the user eventually gets the right password, the loop ends.

Getting your loops to behave

While loops can be dangerous. It's quite easy to write a while loop that works incorrectly, and these can be an exceptionally difficult kind of bug to find and fix. If a while loop is incorrectly designed, it can refuse to ever run or run forever. These endless loops are especially troubling in JavaScript because they can crash the entire browser. If a JavaScript program gets into an endless loop, often the only solution is to use the operating system task manager (Ctrl+Alt+Delete on Windows) to shut down the entire browser.

The easy way to make sure your loop works is to remember that while loops need all the same features as for loops. (These ideas are built into the structure of a for loop. You're responsible for them yourself in a while loop.) If your loop doesn't work, check that you've followed these steps:

- ✓ Identify a key variable: A while loop is normally based on a condition, which is usually a comparison (although it might also be a variable or function that returns a Boolean value). In a for loop, the key variable is almost always an integer. While loops can be based on any type of variable.
- ✓ Initialize the variable before the loop: Before the loop begins, set up the initial value of the key variable to ensure the loop happens at least once. (How does the variable start?)
- ✓ Identify the condition for the loop: A while loop is based on a condition. Define the condition so the loop continues while the condition is true, and exits when the condition is evaluated to false. (How does the variable end?)
- ✓ Change the condition inside the loop: Somewhere inside the loop code, you need to have statements that will eventually make the condition false. If you forget this part, your loop will never end. (How does the variable change?)

This example is a good example of a while loop, but a terrible way to handle security. The password is shown in the clear, and anybody could view the source code to see the correct password. There are far better ways to handle security, but this is the cleanest example of a while loop I could think of.

Managing more complex loops

It won't take long before you find situations where the standard for or while loops do not seem adequate. For example, consider the password example again. This time, you want to ask for a password until the user gets the password correct or guesses incorrectly three times. Think about how you would build that code. There are a number of ways to do it, but here's the cleanest approach:

Example 4-24 complex while loop

```
function threeTries(){
//continues until user is correct or has three
//incorrect guesses
//from while.html
var correct = "HTML5";
var guess = "";
var keepGoing = true;
var tries = 0;
while (keepGoing){
guess = prompt("Password?");
if (guess == correct){
alert("You may proceed");
keepGoing = false;
} else {
tries++;
if (tries >= 3){
alert("Too many tries. Launching missiles...");
keepGoing = false;
} // end if
} // end if
} // end while
} // end threetries
```

This code is a little more complex, but it uses a nice technique to greatly simplify loops:

- ✓ Step 1. Initialize correct and guess. As in the previous example, initialize the correct and guess passwords.
- ✓ Step 2. Build a counter to indicate the number of tries. The tries variable will count how many attempts have been made.
- ✓ Step 3. Build a Boolean sentry variable. The keepGoing variable is special. Its entire job is to indicate whether the loop should continue or not. It is a Boolean variable, meaning it will only contain the values true or false.
- ✓ Step 4. Use keepGoing as the condition. A condition doesn't have to be a comparison. It just has to be true or false. Use the Boolean variable as the condition! As long as keepGoing has the value true, the loop will continue. Any time you want to exit the loop, set keepGoing to false.
- ✓ Step 5. Ask for the password. You still need the password, so get this information from the user.
- ✓ Step 6. Check to see if the password is correct. Use an if statement to see if the password is correct.
- ✓ Step 7. If the password is correct, provide feedback to the user and set keepGoing to false. The next time the while statement is executed, the loop ends. (Remember, you want the loop to end when the password is correct.)

- ✓ Step 8. If the password is incorrect, if the (guess == correct)condition is false, that means the user did not get the password right. In this case, add one to the number of tries.
- ✓ Step 9. Check the number of tries. Build another if statement to check the number of tries.
- ✓ Step 10. If it's been three turns, provide feedback (threatening global annihilation is always fun) and set keepGoing to false.

The basic idea of this strategy is quite straightforward: Create a special Boolean variable with the singular job of indicating whether the loop continues. Any time you want the loop to exit, change the value of that variable. If you change most of your while loops to this format (using a Boolean variable as the condition), you'll generally eliminate most while loop issues. When your code gets complicated, it gets tempting to use and (&&) and or (||) operators to make more complex conditions.

4.3.7 Managing Errors with a Debugger

By the time you're writing loops and conditions, things can go pretty badly in your code. Sometimes it's very hard to tell what exactly is going on. Fortunately, modern browsers have some nice tools that help you look at your code more carefully.

A debugger is a special tool that allows you to run a program in "slow motion," moving one line at a time so you can see exactly what is happening. Google Chrome has a built-in debugger, so I begin with that one. To see how a debugger works, follow these steps.

- ✓ Step 1. Load a page into Chrome. You can add a debugger to most browsers, but Chrome has one built in, so start with that one. I'm loading the forLoops.html page because loops are a common source of bugs.
- ✓ Step 2. Open the Developer Tools window. If you right-click anywhere on the page and choose Inspect Element (or press the F12 key), you'll get a wonderful debugging tool that looks like Figure 4-26 .
- ✓ Step 3. Inspect the page with the Elements tab. The default tab shows you the page in an outline view, letting you see the structure of your page. If you click any element in the outline, you can see what styles are associated with that element. The actual element is also highlighted on the main page so you can see exactly where everything is. This can be very useful for checking your HTML and CSS.
- ✓ Step 4. Move to the Sources tab. The Developer Tools window has a separate tab for working with JavaScript code. Select the Sources tab to see your entire code at once. There's a small menu button that lets you select from all the source pages your program uses. If your page pulls in external JavaScript files, you'll be able to select them here as well. (Note some older versions of Chrome called this the Scripts tab.)
- ✓ Step 5. Set a breakpoint. Typically you let the program begin at normal speed and slow down right before you get to a trouble spot. In this case, I'm interested in the count() function, so click on the first line (16) of that function in the code window. (It's more reliable to click on the first line of the function than the line that declares it, so use line 16 instead of line 15.) Click

the line number of the line you want to pause, and the line number will highlight, indicating it is now a break point.

- ✓ Step 6. Refresh the page. In the main browser, use the refresh button or F5 key to refresh the page. The page may initially be blank. That's fine — it means the program has paused when it encountered the function.
- ✓ Step 7. Your page is now running. If you look back over the main page, you should see it is now up and running. Nothing is happening yet because you haven't activated any of the buttons.
- ✓ Step 8. Click the Count button. The Count button should activate the code in the count function. Click this button to see if that happens.
- ✓ Step 9. Code should now be paused on line 17. Back in the code window, line 17 is now highlighted. That means the browser is paused, and when you activate the step button, the highlighted code will happen.
- ✓ Step 10. Step into the next line. In the Developer Tool window is a series of buttons on top of the right column. Step into the next line looks like a down arrow with a dot under it. You can also use the F11 key to activate the command.
- ✓ Step 11. Step a few times. Use the F11 key or the step button to step forward a few times. Watch how the highlight moves around so you can actually see the loop happening. This is very useful when your code is not behaving properly because it allows you to see exactly how the processor is moving through your code.
- ✓ Step 12. Hover over the variable i in your code. When you are in debug mode, you can hover the mouse over any variable in the code window and you'll see what the current value of that variable is. Often when your code is performing badly, it's because a variable isn't doing what you think it is.
- ✓ Step 13. Add a watch expression to simplify looking at variables. If you think the loop is not behaving, you can add a watch expression to make debugging easier. Right under the step buttons you'll see a tab called Watch Expressions. Click the plus sign to add a new expression. Type in i and enter.
- ✓ Step 14. Continue stepping through the code. Now you can continue to step through the code and see what is happening to the variable. This is incredibly useful when your code is not performing like you want it to.

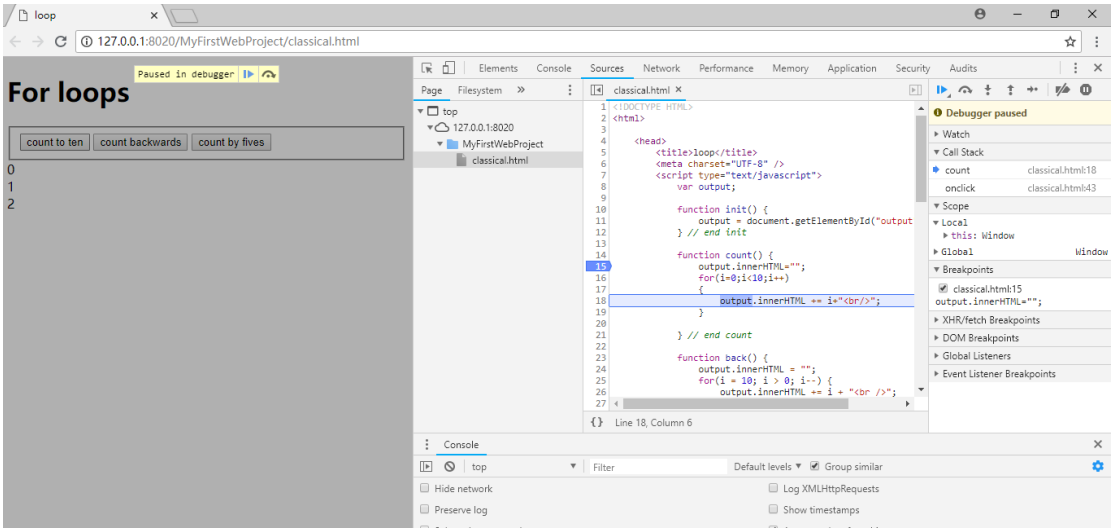


Figure 4-26 debug interface of Chrome

Debugging with the interactive console

The Developer Tools window has another really wonderful tool called the console. But there's much more you can do with this wonderful tool. Try this exercise to see some of the great ways you can use the console:

- ✓ Step 1. Begin with the for loop page. You can debug any page, but forLoops.html is especially helpful for debugging.
- ✓ Step 2. Place a breakpoint. For this demonstration, put a breakpoint in the count() function (line 16 if you're using my version of the code).
- ✓ Step 3. Step through a few lines. Use the step button or F11 key to step through a few lines of code.
- ✓ Step 4. Switch to the console tab. The Console tab switches to console mode. This is particularly interesting when the program is paused, as you can investigate and change the nature of the page in real time.
- ✓ Step 5. Change a style. Try typing document.body.style.backgroundColor = light Green in the console. This modifies the background color of the page in real time. This is fun but not seriously useful.
- ✓ Step 6. Examine the document.body. Type document.body in the console and press Enter. You'll see plenty of information about the body. Document.body is actually a JavaScript variable containing the current document body. It's very powerful and allows you to understand a lot about what's going on.
- ✓ Step 7. Examine the body's innerHTML. Like any HTML element, document.body has an innerHTML property. You can examine this in the console: document.body.innerHTML.
- ✓ Step 8. Look at the variable i. You can examine the current value of any variable as long as that

variable currently has a meaning. Type i (then press enter) to see the current value of the variable i. If the count() function isn't currently running, you may get a strange value here.

- ✓ Step 9. Check the type of i. All variables have a specific type defined by JavaScript, and sometimes that data type is not what you expected. You can ask the browser what type of data any variable contains: typeof(i) returns "number." You may also see "string" or "object."
- ✓ Step 10. See if your output variable is defined correctly. Like many interactive programs, this page has a div called div that contains the output. If this is not defined correctly, it won't work. Try output in the console to see if the output variable is correctly defined and in scope. You can view the contents of output with output. innerHTML, or you can even change the value of output like this: output.innerHTML = "Hi Mom!".
- ✓ Step 11. Check your functions. You can check to see if the functions are what you think they are in the console. Try typing count (with no parentheses) to see the contents of count

It's a fact of life — when you write code, you will have bugs. Every programmer needs to know how to diagnose and fix code when it goes wrong. The first thing to understand is that crashes and bugs are not the same. A crash is a problem with your code that prevents the program from running at all. These sound bad, but they're actually easier to resolve than bugs, which are caused by technically correct code doing the wrong thing.

Resolving syntax errors

The most common type of error is a crash or syntax error, usually meaning you misspelled a command or used a function incorrectly. From the user's point of view, browsers don't usually tell you directly when a syntax error occurs, but simply sit there and pout. The best way to discover what's going wrong is to call up the debugging console. As soon as you discover a page not acting correctly, go to the debugging console and look at the Console tab. You'll see error messages there, and you can often click on an error message to see the problem. As an example, take a look at the following code.

```
function getPassword(){
var correct "HTML5";
var guess = "";
while (guess != correct){
guess = prompt("Password?");
} // end while
alert("You may proceed");
} // end getPassword
```

This code might look just like the getPassword() function from while. but I introduced a subtle error that's difficult to find with the naked eye. Run the program in your browser, click the Guess the Password button, and the browser will seem to do nothing but glare at you insolently.

However, if you activate the Debugging console, you'll realize it's telling you what it thinks is wrong. Figure 4-27 illustrates the Debugging console trying to help.

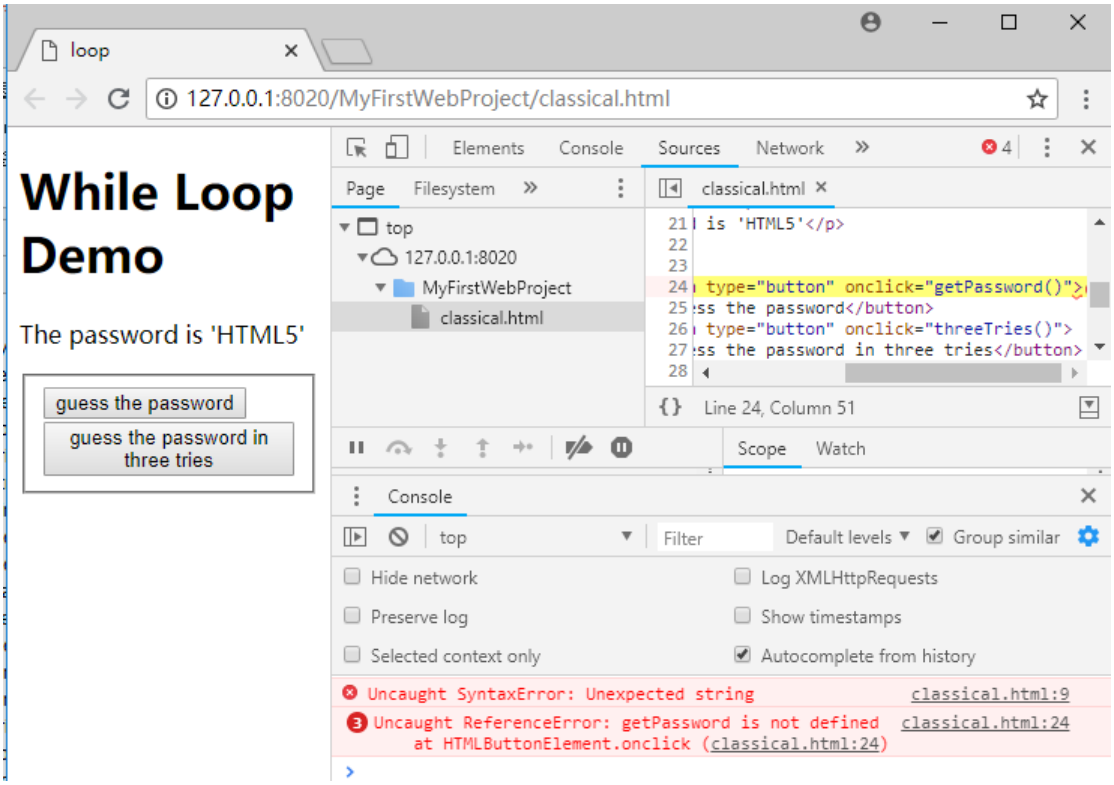


Figure 4-27 syntax error

It would be great if the debugger told you exactly what is wrong, but normally there's a bit of detective work involved in deciphering error messages. It appears in this case that there are two errors, but they're really the same thing. Click the link to the right of the first error and you'll be taken to the Sources view with the offending line highlighted, as you see in Figure 4-28

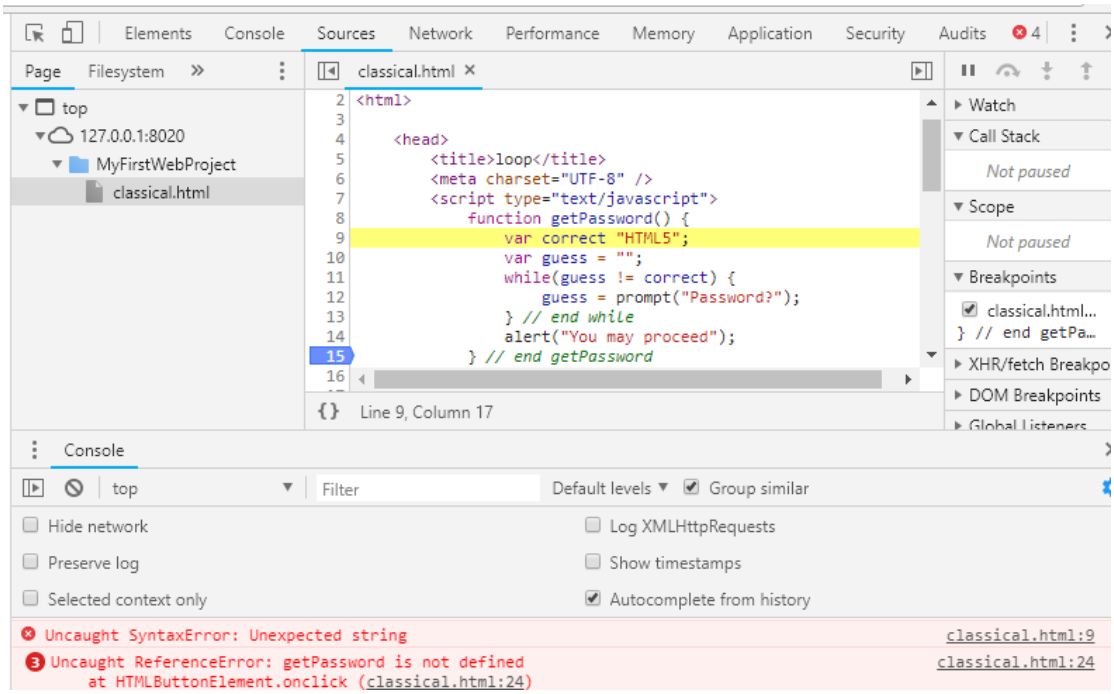


Figure 4-28 highlight the error line

The error messages aren't always as clear as they could be, but they are usually helpful in their own way. The error message here is "unexpected string." That means the browser encountered a string value when it expected something else. That's somewhat helpful, but the real strategy is to know that something is probably wrong with this line, and you need to look it over carefully. At some point, you'll probably realize that line 10 should have a single equals sign. Rather than var correct "HTML5", it should read var correct = "HTML5". This was (as are most syntax errors) a problem caused by sloppy typing. Like most syntax errors, it's kind of difficult to find (but much easier with the debugger). After you find the error, it's usually pretty easy to fix. Change the code in your editor and reload in the browser (with the F5 key) to see if your change fixes things.

Squashing logic bugs

Syntax errors seem bad because they cause the whole program to crash, but they're actually pretty easy to resolve. There's another type of problem called logic errors that are much more troublesome. In fact, they're nearly impossible to resolve without some sort of debugging tool. However, like a syntax error, when you can find a logic error, it's usually quite easy to repair. Take a look at logicError.html to see a typical logic problem in the getPassword() function:

```
function getPassword(){
var correct = "HTML5";
var guess = "";
while (guess == correct){
guess = prompt("Password?");
} // end while
alert("You may proceed");
} // end getPassword
```

Just looking at the code, it's very difficult to see the problem. Worse, when you run the program in your browser, it won't report an error. It won't work correctly, but the code is all technically correct. Rather than telling it to do something illegal (which would result in a syntax error), I have told the program to do something that's completely legal but not logical. Logic errors are called bugs, and they're much more interesting (but subtle) to resolve than syntax errors (normally called crashes).

To resolve a logic error, there's a few steps:

- ✓ Step Step 1. Understand what you're trying to accomplish. Whenever you write a program, be sure you review what you're trying to accomplish before you run the program. If you don't know what you expect, you won't know if your program got there. It's oftengood to write down what you expect so you'll know if you got there. (Professional programmers are usually required to list expectations before they write a single line of code.) For this example, when the user clicks the Guess the Password button, the user should get a prompt allowing them to guess the password.
- ✓ Step 2. Understand what your code did. Run the logicError.html page yourself to see what actually happens. With a logic error, the behavior is unpredictable. A loop may never happen, it may never end, or it might sometimes work right and sometimes not. The key to finding logic errors is to predict why the code is doing what it's doing and why it's not doing what you want. In this example, when I press the Guess the Password button, the You May Proceed dialog box

immediately appears, never giving me the chance to guess a password.

- ✓ Step 3. Form a hypothesis or two before looking at code. Think about what is wrong before you look over the code. Try to describe in plain English (not technical jargon) what is going wrong. In this case, I think something is preventing the prompt from appearing. Maybe the statement causing the prompt is written incorrectly, or maybe the code is never getting there. Those are the two most likely possibilities, so they're what I'll look for. Decide this before you look at code because the moment you see code, you'll start worrying about details rather than thinking about the big picture. Logic errors are almost always about logic, and no amount of staring at code will show you logic errors, nor will a debugger spot them for you.

- ✓ Step 4. Resolve syntax errors. Go to the console and see if there are any syntax errors. If so, resolve them. Logic errors will not appear until you've resolved all syntax errors. If your code shows no syntax errors but still doesn't work correctly, you've got a logic error.

- ✓ Step 5. Start the debugger. Interactive debugging is incredibly helpful with logic errors. Begin with your English definitions of what you think should happen and what you know is happening. Find the function you think is the problem and set a breakpoint at that function.

- ✓ Step 6. Identify key variables or conditions. Most logic errors are centered around a condition that's not working right, and conditions are usually based on variables. Begin by taking a careful look at the conditions that control the behavior you're worried about. In this case, I've got a loop that doesn't seem to be happening — ever. This means I should take a careful look at that loop statement and any variables used in that statement.

- ✓ Step 7. Step to your suspicious code. If you're worried about a condition (which is very common), use the debugger tools to step to that condition, but don't run it yet. (In most debuggers, a highlighted line is about to be run.)

- ✓ Step 8. Look at the relevant variables. Before running the condition line, think about what you think any variables used in that condition should contain. Use the Watch tools or hover over the variable names to ensure you know the current values and they're what you think they should be. In this example, I'm concerned about line 12 (while guess == correct), so I want to see what those variables contain.

- ✓ Step 9. Predict what the suspicious line should do. If you're worried about a condition, you're generally expecting it to do something it isn't doing. In this case, the condition should trigger the prompt command on line 13 when the function is called, but it appears that we're never getting to line 13 (or we are getting there and line 13 isn't doing what we think it's doing). The goal of debugging is to identify which possible problems could be happening and isolate which of these problems are actually occurring. Make sure you know what you're looking for before you start looking for it.

- ✓ Step 10. Compare your expectations with reality. As you step through the getPassword() function in the debugger (with the step into button or F11 key), you might see the problem. The while loop begun in line 12 never executes, meaning line 13 never happens, but it should always happen on the first pass. Now you know exactly what the program is doing, but you don't know why yet.

- ✓ Step 11. Think about your logic. Logic errors aren't about getting the commands right (those are syntax errors). Logic errors are about telling the computer to do the wrong thing. Think hard about the logic you've applied here. In this case, it appears my condition is backwards. You told the computer to continue looping as long as the guess is correct. You probably meant to continue as long as the guess is incorrect. The guess starts out incorrect because of the way you (appropriately) initialized both variables. Thus the condition is automatically skipped and the prompt never happens.
- ✓ Step 12. Fix it. Fixing code is easy when you know what's wrong. In this case, my condition was legal but illogical. Replace guess == correct with guess != correct and your code will work correctly.

Don't worry if you find debugging difficult. Programming is both an art and a science, and debugging logic errors falls much more along the art side of the equation. It does get much easier with practice and experience.

4.4 Functions, Arrays, and Objects

It doesn't take long for your code to become complex. Soon enough, you find yourself wanting to write more sophisticated programs. When things get larger, you need new kinds of organizational structures to handle the added complexity.

You can bundle several lines of code into one container and give this new chunk of code a name: a function. You can also take a whole bunch of variables, put them into a container, and give it a name. That's called an array. If you combine functions and data, you get another interesting structure called an object.

You may have encountered variables and functions in their simplest forms elsewhere in this book (variables were first introduced in section 1 of this chapter, and functions made their appearance in section 2). This section is about how to work with more code and more data without going crazy.

4.4.1 Breaking Code into Functions

Functions come in handy when you're making complex code easier to handle — a useful tool for controlling complexity. You can take a large, complicated program and break it into several smaller pieces. Each piece stands alone and solves a specific part of the overall problem.

You can think of each function as a miniature program. You can define variables in functions, put loops and branches in there, and do anything else you can do with a program. A program using functions is basically a program full of subprograms.

After you define your functions, they're just like new JavaScript commands. In a sense, when you

add functions, you're adding to JavaScript.

To explain functions better, think back to an old campfire song, "The Ants Go Marching." Figure 4-29 recreates this classic song for you in JavaScript format.

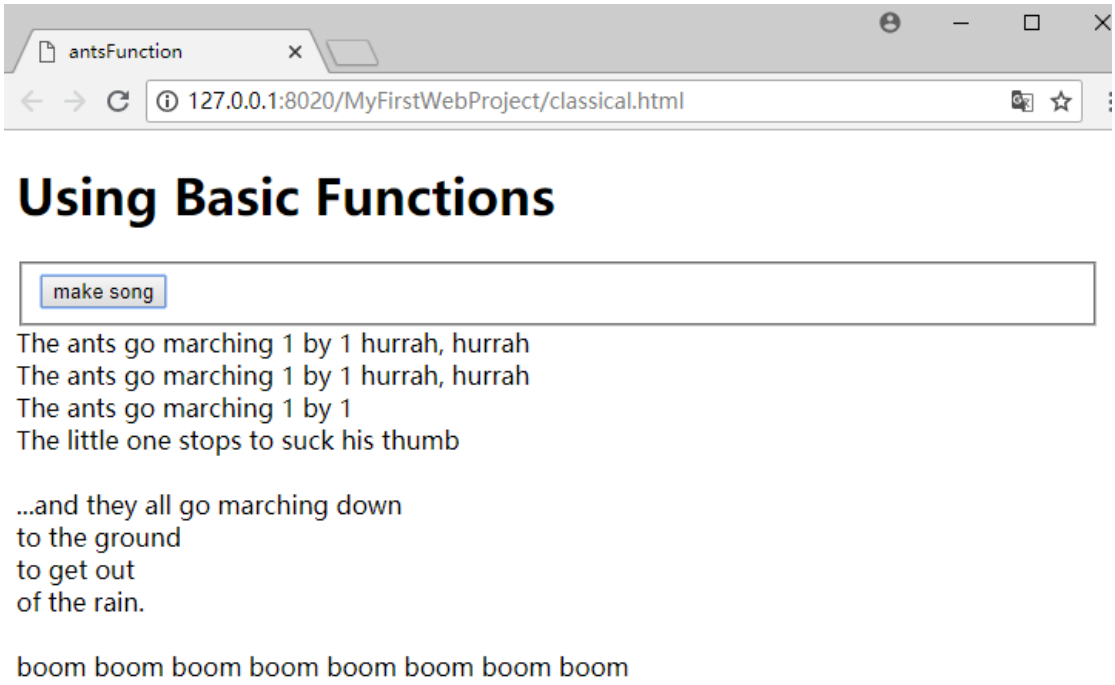


Figure 4-29 ant function

If you're unfamiliar with this song, it simply recounts the story of a bunch of ants. The littlest one apparently has some sort of attention issues. During each verse, the little one gets distracted by something that rhymes with the verse number. The song typically has ten verses, but I'm just doing two for the demo.

Thinking about structure

Before you look at the code, think about the structure of the song, "The Ants Go Marching." Like many songs, it has two parts. The chorus is a phrase repeated many times throughout the song. The song has several verses, which are similar to each other, but not quite identical.

Think about the song sheet passed around the campfire. The chorus is usually listed only one time, and each verse is listed. Sometimes, you have a section somewhere on the song sheet that looks like the following:

```
Verse 1
Chorus
Verse 2
Chorus
```

Musicians call this a road map, and that's a great name for it. A road map is a high-level view of how you progress through the song. In the road map, you don't worry about the details of the particular verse or chorus. The road map shows the big picture, and you can look at each verse or chorus for the details.

Example 4-25 multiple functions

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset = "UTF-8">
    <title>antsFunction</title>
    <script type = "text/javascript">
      var output;

      function chorus() {
        var text = "...and they all go marching down <br />";
        text += "to the ground <br />";
        text += "to get out <br />";
        text += "of the rain. <br />";
        text += " <br />";
        text += "boom boom boom boom boom boom boom boom <br /><br
/>";
        output.innerHTML += text;
      } // end chorus

      function verse1(){
        var text = "The ants go marching 1 by 1 hurrah, hurrah <br
/>";
        text += "The ants go marching 1 by 1 hurrah, hurrah <br />";
        text += "The ants go marching 1 by 1 <br />";
        text += " The little one stops to suck his thumb <br /><br
/>";
        output.innerHTML += text;
      } // end verse1

      function verse2(){
        var text = "The ants go marching 2 by 2 hurrah, hurrah <br
/>";
        text += "The ants go marching 2 by 2 hurrah, hurrah <br />";
        text += "The ants go marching 2 by 2 <br />";
        text += " The little one stops to tie his shoe <br /><br />";
        output.innerHTML += text;
      } // end verse2

      function makeSong(){
        output = document.getElementById("output");
        output.innerHTML = "";
        verse1();
```

```
      chorus();
      verse2();
      chorus();
    } // end makeSong

  </script>
 </head>
 <body>
  <h1>Using Basic Functions</h1>
  <form > <fieldset>
     <button type = "button" onclick = "makeSong()">
      make song
     </button>
   </fieldset> </form>
  <div id = "output">
   The song will appear here...
  </div>
 </body>
</html>
```

The program code breaks the parts of the song into the same pieces a song sheet does. Here are some interesting features of ants function:

- ✓ I created a function called chorus(). Functions are simply collections of code lines with a name.
- ✓ All the code for the chorus goes into this function. Anything I want as part of printing the chorus goes into the chorus() function. Later, when I want to print the chorus, I can just call the chorus() function and it will perform the code I stored there.
- ✓ Each verse has a function, too. I broke the code for each verse into its own function.
- ✓ The makeSong function is a road map. When all the details are delegated to the functions, the main part of the code just controls the order in which the functions are called. In this case, the makeSong() function is called by the button press, which runs all the other functions.
- ✓ Details are hidden in the functions. The makeSong code handles the big picture. The details (how to print the chorus or verses) are hidden inside the functions.
- ✓ I'm using standard form-based output. Each of the functions creates its own part of the song and adds it to the output as needed.

4.4.2 Passing Data to and from Functions

Functions are logically separated from each other. This separation is a good thing because it prevents certain kinds of errors. However, sometimes you want to send information to a function. You may also want a function to return some type of value. In Example 4-26, the page rewrites the "The Ants Go Marching" song in a way that takes advantage of function input and output.

Example 4-26 function parameter

```
<!DOCTYPE HTML>
<html>
<head>
  <title>parameter </title>
  <meta charset = "UTF-8" />
  <style type = "text/css">
  </style>
  <script type = "text/javascript">

  function makeSong(){
    //create output variable
    var output = document.getElementById("output");
    output.innerHTML = "";
    output.innerHTML += verse(1);
    output.innerHTML += chorus();
    output.innerHTML += verse(2);
    output.innerHTML += chorus();
  } // end makeSong

  function chorus(){
    var result = "-and they all came marching down, <br />";
    result += "to the ground, to get out, of the rain. <br />";
    result += "boom boom boom boom <br />";
    result += "boom boom boom boom <br />";
    result += "<br />";
    return result;
  } // end chorus

  function verse(verseNumber){
    var distraction = "";
    if (verseNumber == 1){
      distraction = "suck his thumb";
    } else if (verseNumber == 2){
      distraction = "tie his shoe";
    } else {
      distraction = "there's a problem here...";
    } // end if

    var result = "The ants go marching ";
    result += verseNumber + " by " + verseNumber + ", ";
    result += "hurrah, hurrah <br />";
    result += "The ants go marching ";
```

```
    result += verseNumber + " by " + verseNumber + ", ";
    result += "hurrah, hurrah <br />";
    result += "The ants go marching ";
    result += verseNumber + " by " + verseNumber + "<br />";
    result += "The little one stops to ";
    result += distraction + "<br /> <br />";

    return result;
  } // end verse
  </script>
</head>
<body>
  <h1>Using Functions, parameters, and return values</h1>
  <form>  <fieldset>
      <button type = "button" onclick = "makeSong()">
        make song
      </button>
  </fieldset> </form>
  <div id = "output">
    The song will appear here...
  </div>
</body>
</html>
```

The result of this program looks just like Figure 4-29 to the user. One advantage of functions is that I can improve the underlying behavior of a program without imposing a change in the user's experience.

This code incorporates a couple of important new ideas.

- ✓ These functions return a value. The functions no longer do their own alerts. Instead, they create a value and return it to the main program.
- ✓ Only one verse function exists. Because the verses are all pretty similar, using only one verse function makes sense. This improved function needs to know what verse it's working on to handle the differences.

Examining the makeSong code

The makeSong code has been changed in one significant way. In the last program, the makeSong code called the functions, which did all the work. This time, the functions don't actually output anything themselves. Instead, they collect information and pass it back to the main program. Inside the makeSong code, each function is treated like a variable. You've seen this behavior before. The prompt() method returns a value.

Now the chorus() and verse() methods return values. You can do anything you want to this value, including storing it to a variable, printing it, or comparing it to some other value.

If you have one function that controls all the action, often that function is called main(). Some languages require you to have a function called main(), but JavaScript isn't that picky. For this example, I went with makeSong() because that name is more descriptive than main(). Still, the makeSong() function is a main function because it controls the rest of the program.

Separating the creation of data from its use as I've done here is a good idea. That way, you have more flexibility. After a function creates some information, you can print it to the screen, store it on a web page, put it in a database, or whatever.

Looking at the chorus

The chorus of "The Ants Go Marching" song program has been changed to return a value. Take another look at the chorus() function to see what I mean.

```
function chorus(){
var result = "-and they all came marching down, <br />";
result += "to the ground, to get out, of the rain. <br />";
result += "boom boom boom boom <br />";
result += "boom boom boom boom <br />";
result += "<br />";
return result;
} // end chorus
```

Here's what changed:

- ✓ The purpose of the function has changed. The function is no longer designed to output some value to the screen. Instead, it now provides text to the main program, which can do whatever it wants with the results.
- ✓ There's a variable called text. This variable contains all the text to be sent to the main program. (It contained all the text in the last program, but it's even more important now.)
- ✓ The text variable is concatenated over several lines. I used string concatenation to build a complex value. Note the use of break tags (<br />) to force carriage returns in the HTML output.
- ✓ The return statement sends text to the main program. When you want a function to return some value, simply use return followed by a value or variable. Note that return should be the last line of the function.

Handling the verses

The verse() function is quite interesting:

- ✓ It can print more than one verse.
- ✓ It takes input to determine which verse to print.
- ✓ It modifies the verse based on the input.
- ✓ It returns a value, just like chorus().

To make the verse so versatile (get it? verse-atile!), it must take input from the primary program and

return output.

Passing data to the verse() function

The verse() function is always called with a value inside the parentheses. For example, the main program sets verse(1) to call the first verse, and verse(2) to invoke the second. The value inside the parentheses is called an argument.

The verse function must be designed to accept an argument (because I call it using values inside the parentheses). Look at the first line to see how.

function verse(verseNumber){

In the function definition, I include a variable name. Inside the function, this variable is known as a parameter. (Don't get hung up on the terminology. People often use the terms parameter and argument interchangeably.) The important idea is that whenever the verse() function is called, it automatically has a variable called verseNumber. Whatever argument you send to the verse() function from the main program will become the value of the variable verseNumber inside the function.

You can define a function with as many parameters as you want. Each parameter gives you the opportunity to send a piece of information to the function.

Determining the distraction

If you know the verse number, you can determine what distracts "the little one" in the song. You can determine the distraction in a couple ways, but a simple if-elseif structure is sufficient for this example.

```
var distraction = "";
if (verseNumber == 1){
distraction = "suck his thumb.";
} else if (verseNumber == 2){
distraction = "tie his shoe.";
} else {
distraction = "I have no idea.";
}
```

I initialized the variable distraction to be empty. If verseNum is 1, set distraction to "suck his thumb". If verseNumber is 2, distraction should be "tie his shoe". Any other value for verseNumber is treated as an error by the else clause.

If you're an experienced coder, you may be yelling at this code. It still isn't optimal. Fortunately, in the section "Building a Basic Array" later in this chapter, I show an even better solution for handling this particular situation with arrays. By the time this code segment is complete, verseNumber and distraction both contain a legitimate value.

Creating the text

When you know these variables, it's pretty easy to construct the output text:

```
var result = "The ants go marching ";
result += verseNumber + " by " + verseNumber + ", ";
result += "hurrah, hurrah <br />";
result += "The ants go marching ";
result += verseNumber + " by " + verseNumber + ", ";
result += "hurrah, hurrah <br />";
result += "The ants go marching ";
result += verseNumber + " by " + verseNumber + "<br />";
result += "The little one stops to ";
result += distraction + "<br /> <br />";
return result;
} // end verse
```

4.4.3 Managing Scope

A function is much like an independent mini-program. Any variable you create inside a function has meaning only inside that function. When the function is finished executing, its variables disappear! This setup is actually a really good thing. A major program will have hundreds of variables, and they can be difficult to keep track of. You can reuse a variable name without knowing it or have a value changed inadvertently. When you break your code into functions, each function has its own independent set of variables. You don't have to worry about whether the variables will cause problems elsewhere.

Introducing local and global variables

You can also define variables at the main (script) level. These variables are global variables. A global variable is available at the main level and inside each function. A local variable (one defined inside a function) has meaning only inside the function. The concept of local versus global functions is sometimes referred to as scope.

Local variables are kind of like local police. Local police have a limited geographical jurisdiction, but they're very useful within that space. They know the neighborhood. Sometimes, you encounter situations that cross local jurisdictions. This situation is the kind that requires a state trooper or the FBI. Local variables are local cops, and global variables are the FBI.

Generally, try to make as many of your variables local as possible. The only time you really need a global variable is when you want some information to be used in multiple functions.

4.4.4 Building a Basic Array

If functions are groups of code lines with a name, arrays are groups of variables with a name. Arrays are similar to functions because they're used to manage complexity. An array is a special kind of variable. Use an array whenever you want to work with a list of similar data types.

The following code shows a basic demonstration of arrays:

Example 4-27 a simple array

```
<script type = "text/javascript">
//creating an empty array
var genre = new Array(5);
//storing data in the array
genre[0] = "flight simulation";
genre[1] = "first-person shooters";
genre[2] = "driving";
genre[3] = "action";
genre[4] = "strategy";
//returning data from the array
alert ("I like " + genre[4] + " games.");
</script>
```

The variable genre is a special variable because it contains many values. Essentially, it's a list of genres. The new Array(5) construct creates space in memory for five variables, all named genre.

Accessing array data

After you specify an array, you can work with the individual elements using square-bracket syntax. An integer identifies each element of the array. The index usually begins with. genre[0] = "flight simulation";

The preceding code assigns the text value "flight simulation" to the genre array variable at position 0. Most languages require all array elements to be the same type. JavaScript is very forgiving. You can combine all kinds of stuff in a JavaScript array. This flexibility can sometimes be useful, but be aware that this trick doesn't work in all languages. Generally, I try to keep all the members of an array the same type.

After you store the data in the array, you can use the same square-bracket syntax to read the information. The line

alert ("I like " + genre[4] + " games.");

finds element 4 of the genre array and includes it in an output message. And the result is shown as Figure 4-30

Figure 4-30 An array

Using arrays with for loops

The main reason to use arrays is convenience. When you have a lot of information in an array, you can write code to work with the data quickly. Whenever you have an array of data, you commonly want to do something with each element in the array. Take a look at Example 4-28 to see how you can do so:

Example 4-28 using arrays with for loops

```
<script type = "text/javascript">
//pre-loading an array
var gameList = new Array("Flight Gear", "Sauerbraten", "Future
Pinball",
"Racer", "TORCS", "Orbiter", "Step Mania", "NetHack",
"Marathon", "Crimson Fields");
var text = "";
for (i = 0; i < gameList.length; i++){
text += "I love " + gameList[i] + "\n";
} // end for loop
alert(text);
</script>
```

The result is shown as the Figure 4-31.

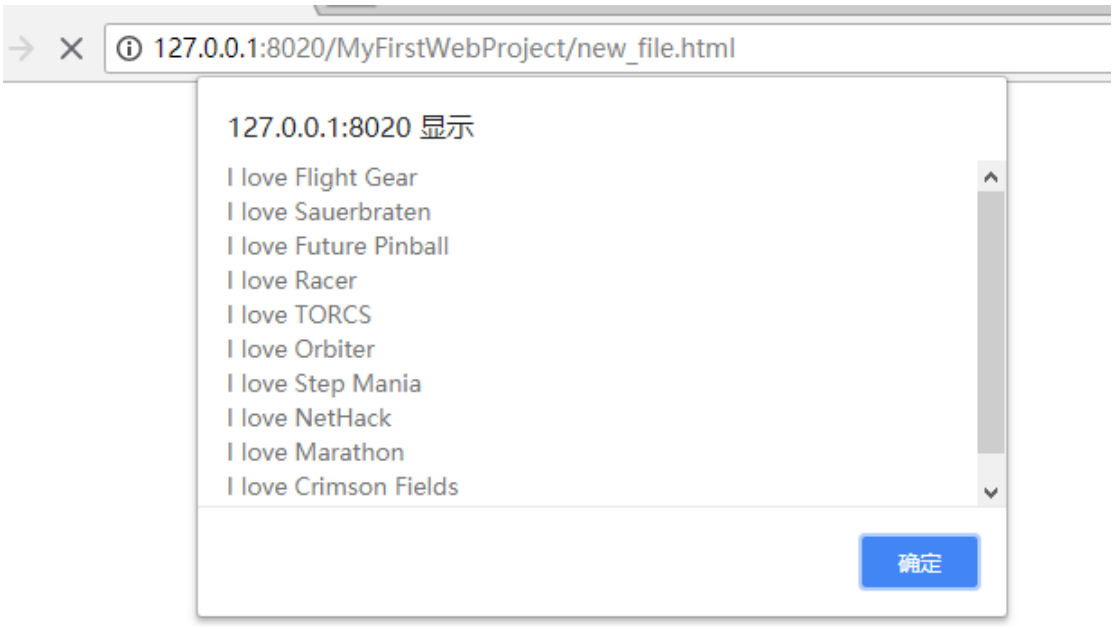


Figure 4-31 show information of an array

Notice several things in this code:

- ✓ The array called gameList. This array contains the names of some of many freeware games.
- ✓ The array is preloaded with values. If you provide a list of values when creating an array, JavaScript simply preloads the array with the values you indicate. You don't need to specify the size of the array if you preload it.

- ✓ A for loop steps through the array. Arrays and for loops are natural companions. The for loop steps through each element of the array.
- ✓ The array's length is used in the for loop condition. Rather than specifying the value 10, I used the array's length property in my for loop. This practice is good because the loop automatically adjusts to the size of the array when I add or remove elements.
- ✓ Do something with each element. Because i goes from 0 to 9 (the array indices), I can easily print each value of the array. In this example, I simply add to an output string.
- ✓ Note the newline characters. The \n combination is a special character that tells JavaScript to add a carriage return, such as you get by pressing the Enter key.

Revisiting the ants song

If you read the earlier sections, you probably just got that marching ant song out of your head. Sorry. Take a look at the following variation, which uses arrays and loops to simplify the code even more.

Example 4-29 use array to rewrite ant song

```
<script type = "text/javascript">
//This old man using functions and arrays
var distractionList = Array("", "suck his thumb", "tie his shoe",
"climb a tree", "shut the door");
function makeSong(){
//create output variable
var output = document.getElementById("output");
output.innerHTML = "";
for (verseNumber = 1; verseNumber < distractionList.length;
verseNumber++){
output.innerHTML += verse(verseNumber);
output.innerHTML += chorus();
} // end for loop
} // end makeSong
function chorus(){
var result = "-and they all came marching down, <br />";
result += "to the ground, to get out, of the rain. <br />";
result += "boom boom boom boom <br />";
result += "boom boom boom boom <br />";
result += "<br />";
return result;
} // end chorus
function verse(verseNumber){
var distraction = distractionList[verseNumber];
var result = "The ants go marching ";
result += verseNumber + " by " + verseNumber + ", ";
result += "hurrah, hurrah <br />";
```

```
result += "The ants go marching ";
result += verseNumber + " by " + verseNumber + ", ";
result += "hurrah, hurrah <br />";
result += "The ants go marching ";
result += verseNumber + " by " + verseNumber + "<br />";
result += "The little one stops to ";
result += distraction + "<br /> <br />";
return result;
} // end verse
</script>
```

This code is just a little different from the antsParam program shown in the former of this section called “Passing Data to and from Functions.”

- ✓ It has an array called distractionList. This array is (despite the misleading name) a list of distractions. I made the first one (element zero) blank so that the verse numbers would line up properly. (Remember, computers normally count beginning with zero.)
- ✓ The verse()function looks up a distraction. Because distractions are now in an array, you can use the verseNumber as an index to loop up a particular distraction. Compare this function to the verse() function in antsParam. This program can be found in the section “Passing data to and from Functions.” Although arrays require a little more planning than code structures, they can highly improve the readability of your code.
- ✓ The makeSong() function is a loop. I step through each element of the distractionList array, printing the appropriate verse and chorus.
- ✓ The chorus()function remains unchanged. You don’t need to change chorus().

4.4.5 Creating Your Own Objects

So far you’ve used a lot of wonderful objects in JavaScript, like the document object and the array object. However, that’s just the beginning. It turns out you can build your own objects too, and these objects can be very powerful and flexible. Objects typically have two important components: properties and methods. A property is like a variable associated with an object. The properties taken together describe the object. A method is like a function associated with an object. The methods describe things the object can do. If functions allow you to put code segments together and arrays allow you to put variables together, objects allow you to put both code segments and variables (and functions and arrays) in the same large construct.

Building a basic object

JavaScript makes it trivially easy to build an object. Because a variable can contain any value, you can simply start treating a variable like an object and it becomes one. Example 4-30 shows the first object property.

Example 4-30 the first object

```
<!DOCTYPE html>
<script type="text/javascript">
    //create the critter
    var critter = new Object();
    //add some properties
    critter.name = "Tom";
    critter.age = 5;
    //view property values
    alert("the critter's name is " + critter.name);
</script>
<html>
    <head>
        <meta charset="UTF-8">
        <title></title>
    </head>
    <body>
    </body>
</html>
```

Figure 4-32 shows a critter that has a property.

Figure 4-32 first object

The way it works is not difficult to follow:

- ✓ Create a new Object. JavaScript has a built-in object called Object. Make a variable with the new Object() syntax, and you'll build yourself a shiny, new standard object.
- ✓ Add properties to the object. A property is a subvariable. It's nothing more than a variable attached to a specific object. When you assign a value to critter.name, for example, you're specifying that critter has a property called name and you're also giving it a starting value.
- ✓ An object can have any number of properties. Just keep adding properties. This allows you to group a number of variables into one larger object.
- ✓ Each property can contain any type of data. Unlike arrays where it's common for all the elements to contain exactly the same type of data, each property can have a different type.

- ✓ Use the dot syntax to view or change a property. If the critter object has a name property, you can use critter.name as a variable. Like other variables, you can change the value by assigning a new value to critter.name or you can read the content of the property.

If you're used to a stricter object-oriented language, such as Java, you'll find JavaScript's easy-going attitude quite strange and maybe a bit sloppy. Other languages do have a lot more rules about how objects are made and used, but JavaScript's approach has its charms. Don't get too tied up in the differences. The way JavaScript handles objects is powerful and refreshing.

Adding methods to an object

Objects have other characteristics besides properties. They can also have methods. A method is simply a function attached to an object. To see what I'm talking about, take a look at this example:

Example 4-31 methods of object

```
<script type="text/javascript">
    //create the critter
    var critter = new Object();
    //add some properties
    critter.name = "Tom";
    critter.age = 5;
    //create a method
    critter.talk = function() {
          msg = "Hi! My name is " + this.name;
          msg += " and I'm " + this.age;
          alert(msg);
       } // end method
       // call the talk method
    critter.talk();
</script>
```

This example extends the critter object. In addition to properties, the new critter has a talk() method. If a property describes a characteristic of an object, a method describes something the object can do. Figure 4-33 illustrates the critter showing off its talk() method:

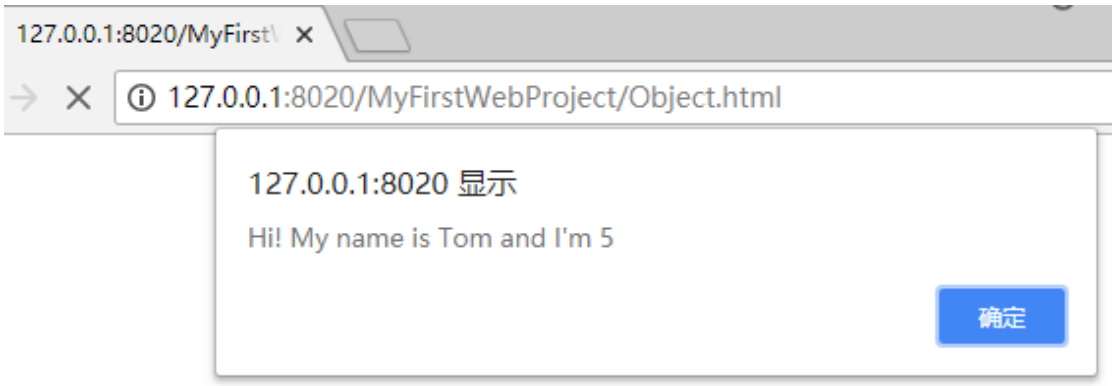


Figure 4-33 methods of object

Here's how it works:

- ✓ Build an object with whatever properties you need. Begin by building an object and giving it some properties.
- ✓ Define a method much like a property. In fact, methods are properties in JavaScript, but don't worry too much about that; it'll make your head explode.
- ✓ You can assign a prebuilt function to a method. If you created a function that you want to use as a method, you can simply assign it.
- ✓ You can also create an anonymous function. More often, you'll want to create your method right there as you define the object. You can create a function immediately with the function() { syntax.
- ✓ The this keyword refers to the current object. Inside the function, you may want to access the properties of the object. this.name refers to the name property of the current object.
- ✓ You can then refer to the method directly.After you define an object with a method, you can invoke it. For example, if the critter object has a talk method, use critter.talk() to invoke this method.

Building a reusable object

These objects are nice, but what if you want to build several objects with the same definition? JavaScript supports an idea called a constructor, which allows you to define an object pattern and reuse it.

Here's an example:

```
//create the critter
//building a constructor
function Critter(lName, lAge){
this.name = lName;
this.age = lAge;
this.talk = function(){
msg = "Hi! My name is " + this.name;
msg += " and I'm " + this.age;
alert(msg);
} // end talk method
} // end Critter class def
function main(){
//build two critters
critterA = new Critter("Alpha", 1);
critterB = new Critter("Beta", 2);
critterB.name = "Charlie";
critterB.age = 3;
//have 'em talk
critterA.talk();
```

```
critterB.talk();
} // end main
main();
```

This example involves creating a class (a pattern for generating objects) and reusing that definition to build two different critters. First, look over how the class definition works:

- ✓ Build an ordinary function: JavaScript classes are defined as extensions of a function. The function name will also be the class name. Note that the name of a class function normally begins with an uppercase letter. When a function is used in this way to describe an object, the function is called the object's constructor. The constructor can take parameters if you wish, but it normally does not return any values. In my particular example, I add parameters for name and age.
- ✓ Use this to define properties: Add any properties you want to include, including default values. Note that you can change the values of theselater if you wish. Each property should begin with this and a period. If you want your object to have a color property, you'd say something like this.color = "blue". My example uses the local parameters to define the properties. This is a very common practice because it's an easy way to preload important properties.
- ✓ Use this to define any methods you want: If you want your object to have methods, define them using the this operator followed by the function(){ keyword. You can add as many functions as you wish.

The way JavaScript defines and uses objects is easy but a little nonstandard. Most other languages that support object-oriented programming (OOP) do it in a different way than the technique described here. Some would argue that JavaScript is not a true OOP language, as it doesn't support a feature called inheritance, but instead uses a feature called prototyping. The difference isn't all that critical because most uses of OOP in JavaScript are very simple objects like the ones described here. Just appreciate that this introduction to object-oriented programming is very cursory, but enough to get you started.

Using your shiny new objects

After you define a class, you can reuse it. Look again at the main function to see how I use my newly minted Critter class:

```
function main(){
//build two critters
critterA = new Critter("Alpha", 1);
critterB = new Critter("Beta", 2);
critterB.name = "Charlie";
critterB.age = 3;
//have 'em talk
critterA.talk();
critterB.talk();
} // end main
main();
```

After you define a class, you can use it as a new data type. This is a very powerful capability. Here's how it works:

- ✓ Be sure you have access to the class: A class isn't useful unless JavaScript knows about it. In this example, the class is defined within the code.
- ✓ Create an instance of the class with the new keyword: The new keyword means you want to make a particular critter based on the definition. Normally, you assign your new object to a variable. My constructor expects the name and age to be supplied, so it automatically creates a critter with the given name and age.
- ✓ Modify the class properties as you wish: You can change the values of any of the class properties. In my example, I change the name and age of the second critter just to show how it's done.
- ✓ Call class methods: Because the critter class has a talk() method, you can use it whenever you want the critter to talk.

4.4.6 Introducing JSON

JavaScript objects and arrays are incredibly flexible. In fact, they are so well known for their power and ease of use that a special data format called JavaScript Object Notation (JSON) has been adopted by many other languages. JSON is mainly used as a way to store complex data (especially multidimensional arrays) and pass the data from program to program. JSON is essentially another way of describing complex data in a JavaScript object format.

When you describe data in JSON, you generally do not need a constructor because the data is used to determine the structure of the class. JSON data is becoming a very important part of web programming because it allows an easy mechanism for transporting data between programs and programming languages.

Storing data in JSON format

To see how JSON works, look at this simple code fragment:

```
var critter = {
"name": "George",
"age": 10
};
```

This code describes a critter. The critter has two properties, a name and an age. The critter looks much like an array, but rather than using a numeric index like most arrays, the critter has string values to serve as indices. It is in fact an object.

You can refer to the individual elements with a variation of array syntax, like this:

```
alert(critter["name"]);
```

You can also use what's called dot notation (as used in objects) like this:

```
alert(critter.age);
```

Both notations work the same way. Most of the built-in JavaScript objects use dot notation, but either is acceptable.

To store data in JSON notation:

- ✓ Create the variable. You can use the var statement like you do any variable.
- ✓ Contain the content within braces ({}). This is the same mechanism you use to create a preloaded array (as described earlier in this chapter).
- ✓ Designate a key. For the critter, I want the properties to be named “name” and “age” rather than numeric indices. For each property, I begin with the property name. The key can be a string or an integer.
- ✓ Follow the key with a colon (:).
- ✓ Create the value associated with that key. You can then associate any type of value you want with the key. In this case, I associate the value George with the key name.
- ✓ Separate each name/value pair with a comma (,). You can add as many name/value pairs as you wish.

4.5 Getting Valid Input

Getting input from the user is always nice, but sometimes users make mistakes. Whenever you can, you want to make the user’s job easier and prevent certain kinds of mistakes.

Fortunately, you can take advantage of several tools designed exactly for that purpose. These tools can help ensure that the data the user enters is useful and valid

4.5.1 Getting Input from a Drop-Down List

The most obvious way to ensure that the user enters something valid is to supply him with valid choices. The drop-down list is an obvious and easy way to do this, as you can see from Figure 4-34.

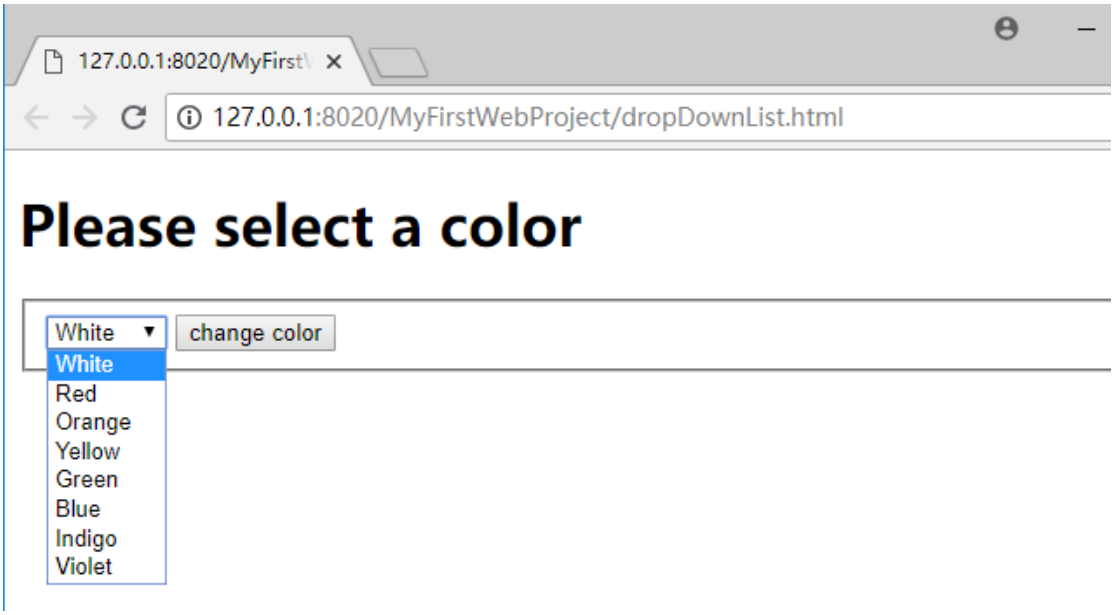


Figure 4-34 drop down list

The list-box approach has a lot of advantages over text field input:

- ✓ The user can input with the mouse, which is faster and easier than typing.
- ✓ You shouldn't have any spelling errors because the user didn't type the response.
- ✓ The user knows all the answers available because they're listed.
- ✓ You can be sure the user gives you a valid answer because you supplied the possible responses.
- ✓ User responses can be mapped to more complex values — for example, you can show the user Red and have the list box return the hex value #FF0000.

Building the form

When you're creating a predefined list of choices, create the HTML form first because it defines all the elements you'll need for the function. The code is a standard form:

Example 4-32 drop down list

```
<body>
<form>
<h1>Please select a color</h1>
<fieldset>
<select id = "selColor">
<option value = "#FFFFFF">White</option>
<option value = "#FF0000">Red</option>
<option value = "#FFCC00">Orange</option>
<option value = "#FFFF00">Yellow</option>
<option value = "#00FF00">Green</option>
<option value = "#0000FF">Blue</option>
```

```
<option value = "#663366">Indigo</option>
<option value = "#FF00FF">Violet</option>
</select>
<input type = "button" value = "change color" onclick = "changeColor()" />
</fieldset>
</form>
</body>
```

The select object's default behavior is to provide a drop-down list. The first element on the list is displayed, but when the user clicks the list, the other options appear.

A select object that the code refers to should have an id field. The other element in the form is a button. When the user clicks the button, the changeColor() function is triggered.

Because the only element in this form is the select object, you may want to change the background color immediately without requiring a button click. You can do so by adding an event handler directly onto the select object:

```
<select id = "selColor" onchange = "changeColor()">
```

The event handler causes the changeColor() function to be triggered as soon as the user changes the select object's value. Typically, you'll forego the user clicking a button only when the select is the only element in the form. If the form includes several elements, processing doesn't usually happen until the user signals she's ready by clicking a button.

Reading the list box

Fortunately, standard drop-down lists are quite easy to read. Here's the JavaScript code:

Example 4-33 drop down list script

```
<script type = "text/javascript">
function changeColor(){
var selColor = document.getElementById("selColor");
var color = selColor.value;
document.body.style.backgroundColor = color;
} // end function
</script>
```

As you can see, the process for reading the select object is much like working with a text-style field:

- ✓ Create a variable to represent the select object. The document. getElementById() trick works here just like it does for text fields.
- ✓ Extract the value property of the select object. The value property of the select object reflects the value of the currently selected option. So, if the user has chosen Yellow, the value of selColor is "#FFFF00".
- ✓ Set the document's background color. Use the DOM mechanism to set the body's background color to the chosen value.

4.5.2 Managing Multiple Selections

You can use the select object in a more powerful way than the method I describe in the preceding section. Figure 4-35 shows a page with a multiple selection list box.

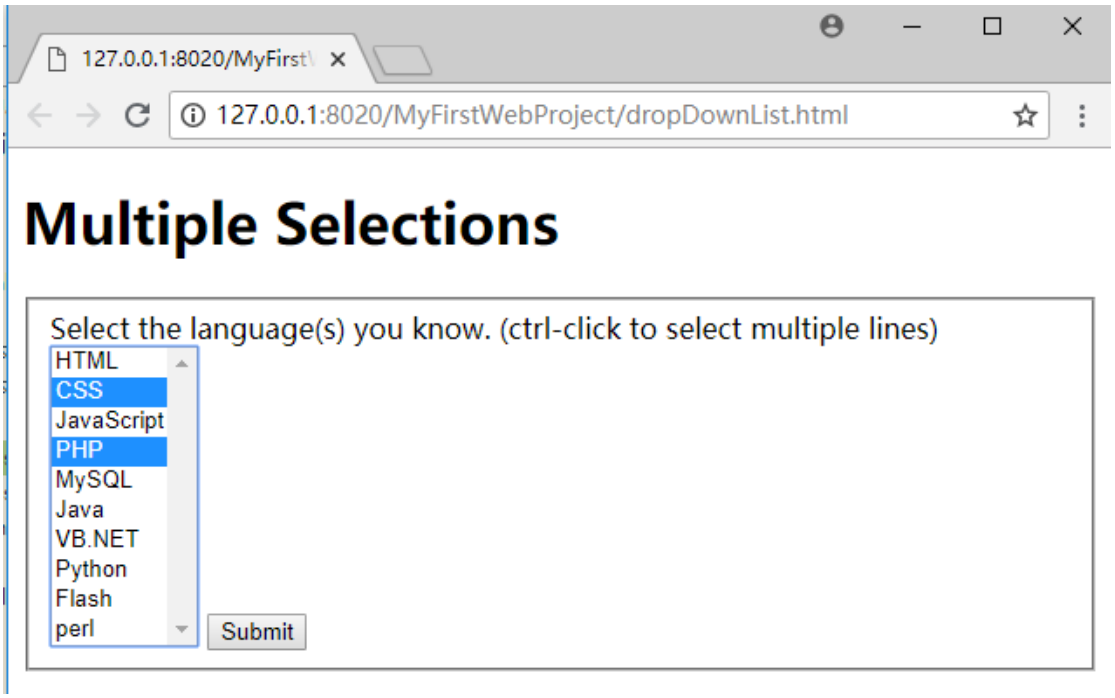


Figure 4-35 multiple selection

To make multiple selection work, you have to make a few changes to both the HTML and the JavaScript code.

Coding a multiple selection select object

You modify the select code in two ways to make multiple selections:

Indicate multiple selections are allowed. By default, select boxes have only one value. You'll need to set a switch to tell the browser to allow more than one item to be selected.

Make the mode a multiline select. The standard drop-down behavior doesn't make sense when you want multiple selections because the user needs to see all the options at once. Most browsers automatically switch into a multiline mode, but you should control the process directly.

The HTML code for multiple selection is similar to the dropdownList page, described in the preceding section, but note a couple of changes.

```
<body>
<h1>Multiple Selections</h1>
<form>
<fieldset>
<label> Select the language(s) you know. (ctrl-click to select
multiple lines) </label>
```

```
<select id = "selLanguage" multiple = "multiple" size = "10">
<option value = "HTML">HTML</option>
<option value = "CSS">CSS</option>
<option value = "JavaScript">JavaScript</option>
<option value = "PHP">PHP</option>
<option value = "MySQL">MySQL</option>
<option value = "Java">Java</option>
<option value = "VB.NET">VB.NET</option>
<option value = "Python">Python</option>
<option value = "Flash">Flash</option>
<option value = "Perl">perl</option>
</select>
<button type = "button" onclick = "showChoices()"> Submit </button>
</fieldset>
</form>
<div id = "output">
</div>
</body>
```

The code isn't shocking, but it does have some important features:

- ✓ Call the select object selLanguage. As usual, the form elements need an id attribute so that you can read it in the JavaScript.
- ✓ Add the multiple attribute to your select object. This attribute tells the browser to accept multiple inputs using Shift+click (for contiguous selections) or Ctrl+click (for more precise selection).
- ✓ Set the size to 10. The size indicates the number of lines to be displayed. I set the size to 10 because my list has ten options.
- ✓ Make a button. With multiple selection, you probably won't want to trigger the action until the user has finished making selections. A separate button is the easiest way to make sure the code is triggered when you want it to happen.
- ✓ Create an output div. This code holds the response.

Writing the JavaScript code

The JavaScript code for reading a multiple-selection list box is a bit different than the standard selection code described in the section "Reading the list box" earlier in this section. The value property usually returns one value, but a multiple-selection list box often returns more than one result.

The key is to recognize that a list of option objects inside a select object is really a kind of array, not just one value. You can look more closely at the list of objects to see which ones are selected, which is essentially what the showChoices() function does:

```
<script type = "text/javascript">
function showChoices(){
//retrieve data
var selLanguage = document.getElementById("selLanguage");
//set up output string
var result = "<h2>Your Languages<\/h2>";
result += "<ul> \n";
//step through options
for (i = 0; i < selLanguage.length; i++){
//examine current option
currentOption = selLanguage[i];
//print it if it has been selected
if (currentOption.selected == true){
result += " <li>" + currentOption.value + "<\/li> \n";
} // end if
} // end for loop
//finish off the list and print it out
result += "<\/ul> \n";
output = document.getElementById("output");
output.innerHTML = result;
} // end showChoices
</script>
```

At first, the code seems intimidating, but if you break it down, it's not too tricky.

1. Create a variable to represent the entire select object.

The standard document.getElementById() technique works fine.

```
var selLanguage = document.getElementById("selLanguage");
```

2. Create a string variable to hold the output.

When you're building complex HTML output, working with a string variable is much easier than directly writing code to the element.

```
var result = "<h2>Your Languages<\/h2>";
```

3. Build an unordered list to display the results.

An unordered list is a good way to spit out the results, so I create one in my result variable.

```
result += "<ul> \n";
```

4. Step through selLanguage as if it were an array.

Use a for loop to examine the list box line by line. Note that selLanguage has a length property like an array.

```
for (i = 0; i < selLanguage.length; i++){
```

5. Assign the current element to a temporary variable.

The currentOption variable holds a reference to each option element in the original select object as the loop progresses.

```
currentOption = selLanguage[i];
```

6. Check to see whether the current element has been selected.

The object currentOption has a selected property that tells you whether the object has been highlighted by the user. selected is a Boolean property, so it's either true or false.

```
if (currentOption.selected == true){
```

7. If the element has been selected, add an entry to the output list.

If the user has highlighted this object, create an entry in the unordered list housed in the result variable.

```
result += " <li>" + currentOption.value + "<\/li> \n";
```

8. Close up the list.

After the loop has finished cycling through all the objects, you can close up the unordered list you've been building.

```
result += "<\/ul> \n";
```

9. Print results to the output div.

The output div's innerHTML property is a perfect place to print the unordered list.

```
output = document.getElementById("output");
output.innerHTML = result;
```

Something strange is going on here. The options of a select box act like an array. An unordered list is a lot like an array. Bingo! They are arrays, just in different forms. You can think of any listed data as an array. Sometimes you organize the data like a list (for display), sometimes like an array (for storage in memory), and sometimes it's a select group (for user input). Now you're starting to think like a programmer!

4.5.3 Check, Please: Reading Check Boxes

Check boxes fulfill another useful data input function. They're useful any time you have Boolean data. If some value can be true or false, a check box is a good tool. Figure 4-36 illustrates a page that responds to check boxes. Check boxes are independent of each other. Although they're often found in groups, any check box can be checked or unchecked regardless of the status of its neighbors.

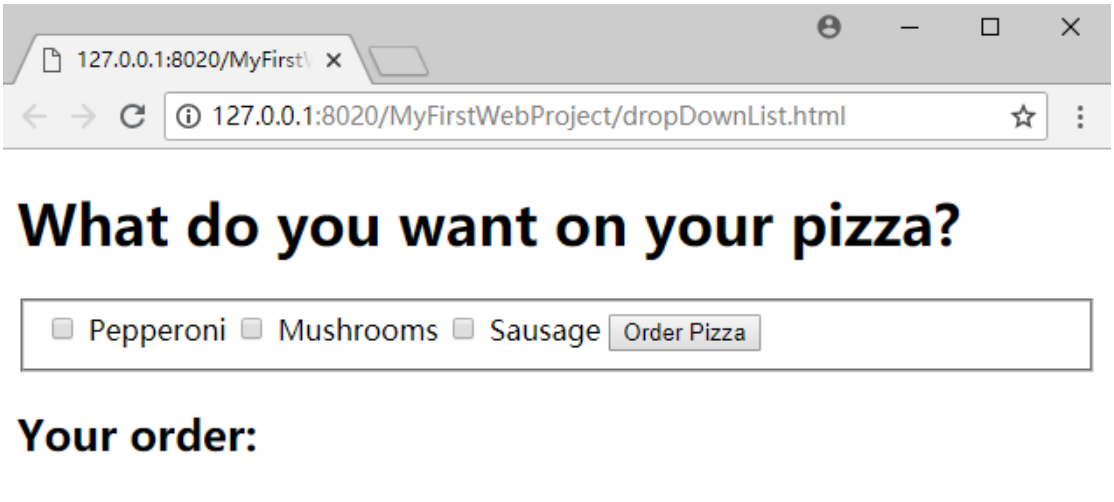


Figure 4-36 check box

Building the check box page

To build the check box page shown in Figure 4-36, start by looking at the HTML:

Example 4-34 create check box

```
<body>
<h1>What do you want on your pizza?</h1>
<form>
<fieldset>
<input type = "checkbox" id = "chkPepperoni" value = "pepperoni" />
<label for = "chkPepperoni">Pepperoni</label>
<input type = "checkbox" id = "chkMushroom" value = "mushrooms" />
<label for = "chkMushroom">Mushrooms</label>
<input type = "checkbox" id = "chkSausage" value = "sausage" />
<label for = "chkSausage">Sausage</label>
<button type = "button" onclick = "order()"> Order Pizza </button>
</fieldset>
</form>
<h2>Your order:</h2>
<div id = "output">
</div>
</body>
```

Each check box is an individual input element. Note that check box values aren't displayed. Instead, a label (or similar text) is usually placed after the check box. A button calls an order() function.

Note the labels have a for attribute which connects each label to the corresponding check box. When you connect a label to a check box in this way, the user can activate the check box by clicking on the box or the label. This provides a larger target for the user, making their life easier. Happy users make fewer mistakes, which makes your life easier.

Responding to the check boxes

Check boxes don't require a lot of care and feeding. After you extract it, the check box has two critical properties:

```
You can use the value property to store a value associated with the
check box.
The checked property is a Boolean value, indicating whether the
check box is checked or not.
```

The code for the order() function shows how it's done:

Example 4-35 check box script

```
function order(){
//get variables
var chkPepperoni = document.getElementById("chkPepperoni");
var chkMushroom = document.getElementById("chkMushroom");
var chkSausage = document.getElementById("chkSausage");
var output = document.getElementById("output");
var result = "<ul> \n"
if (chkPepperoni.checked){
result += "<li>" + chkPepperoni.value + "</li> \n";
} // end if
if (chkMushroom.checked){
result += "<li>" + chkMushroom.value + "</li> \n";
} // end if
if (chkSausage.checked){
result += "<li>" + chkSausage.value + "</li> \n";
} // end if
result += "</ul> \n"
output.innerHTML = result;
} // end function
```

For each check box,

1. Determine whether the check box is checked.

Use the checked property as a condition.

2. If so, return the value property associated with the check box. Often, in practice, the value property is left out. The important thing is whether the check box is checked. If chkMushroom is checked, the user obviously wants mushrooms, so you may not need to explicitly store that data in the check box itself.

4.5.4 Working with Radio Buttons

Radio button groups appear pretty simple, but they're more complex than they seem. Figure 4-37 shows a page using radio button selection.

Figure 4-37 radio box

The most important thing to remember about radio buttons is that, like wildebeests and power-walkers, they must be in groups. Each group of radio buttons has only one button active. The group should be set up so that one button is always active.

You specify the radio button group in the HTML code. Each element of the group can still have a unique id (which comes in handy for associating with a label). Look over the code, and you'll notice something interesting. All the radio buttons have the same name!

Example 4-36 create radio box

```
<body>
<h1>With what weapon will you fight the dragon?</h1>
<form action = "">
<fieldset>
<input type = "radio" name = "we apon" id = "radSpoon"
value = "spoon" checked = "checked" />
<label for = "radSpoon">Spoon</label>
<input type = "radio" name = "weapon" id = "radFlower" value =
"flower" />
<label for = "radFlower">Flower</label>
<input type = "radio" name = "weapon" id = "radNoodle" value = "wet
noodle" />
<label for = "radNoodle">Wet Noodle</label>
<button type = "button" onclick = "fight()"> fight the dragon
</button>
</fieldset>
</form>
<div id = "output">
</div>
</body>
```

Using a name attribute when everything else has an id seems a little odd, but you do it for a good

reason. The name attribute is used to indicate the group of radio buttons. Because all the buttons in this group have the same name, they're related, and only one of them will be selected. Each button can still have a unique ID (and in fact it does). The ID is still useful for associating a label with the button. Once again, this provides a larger click target so the user can click on either the button or the label associated with that button.

The browser recognizes this behavior and automatically unselects the other buttons in the group whenever one is selected. I added a label to describe what each radio button means. You need to preset one of the radio buttons to true with the checked = "checked" attribute. If you fail to do so, you have to add code to account for the possibility that there is no answer at all.

Interpreting Radio Buttons

Getting information from a group of radio buttons requires a slightly different technique than most of the form elements. Unlike the select object, there is no container object that can return a simple value. You also can't just go through every radio button on the page because you may have more than one group. (Imagine a page with a multiple-choice test.) This issue is where the name attribute comes in. Although ids must be unique, multiple elements on a page can have the same name. If they do, you can treat these elements as an array. Look over the code to see how it works:

Example 4-37 radio box script

```
function fight(){
var weapon = document.getElementsByName("weapon");
for (i = 0; i < weapon.length; i++){
currentWeapon = weapon[i];
if (currentWeapon.checked){
var selectedWeapon = currentWeapon.value;
} // end if
} // end for
var output = document.getElementById("output");
var response = "<h2>You defeated the dragon with a ";
response += selectedWeapon + "</h2> \n";
output.innerHTML = response;
} // end function
```

This code looks much like all the other code in this chapter, but it has a sneaky difference:

- ✓ It uses getElementsByName to retrieve an array of elements with this name. Now that you're comfortable with getElementById, I throw a monkey wrench in the works. Note that it's plural — getElementsByName — because this tool is used to extract an array of elements. It returns

an array of elements. (In this case, all the radio buttons in the weapon group.)

- ✓ It treats the result as an array. The resulting variable (weapon in this example) is an array. As usual, the most common thing to do with arrays is process them with loops. Use a for loop to step through each element in the array.
- ✓ Assign each element of the array to currentWeapon. This variable holds a reference to the current radio button.
- ✓ Check to see whether the current weapon is checked. The checked property indicates whether any radio button is checked.
- ✓ If so, retain the value of the radio button. If a radio button is checked, its value is the current value of the group, so store it in a variable for later use.
- ✓ Output the results. You can now process the results as you would with data from any other resource.

4.6 Quiz

1. How to create a comment in JavaScript

2. Describe the features of variable

3. Describe the action of prompt();

4. Describe the characteristics of an object

5. What is the DOM?

6. What is function?

7. Give an example for nested statement.

8. Give an example for switch statement.

9. You can bundle several lines of code into one container and give this new chunk of code a name: a _____. You can also take a whole bunch of variables, put them into a container, and give it a name. That's called an _____. If you combine functions and data, you get another interesting structure called an _____.

10. Create an array and assigned the value.

11. What is the object's components?

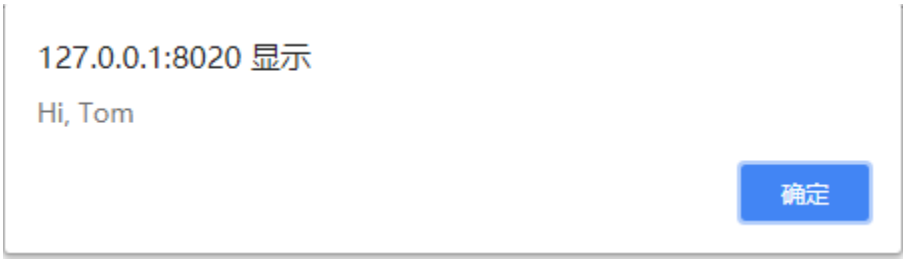
12. Create constructor of object.

13. What is the advantages to get input from drop down list?

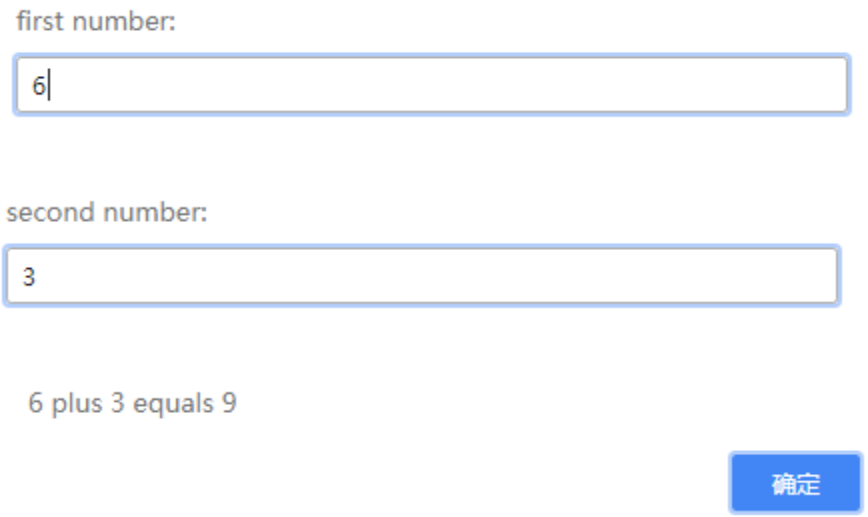
14. How group many radio button?

4.7 Practice

1. Write a JavaScript to welcome your user. The result like these.



2. Write a JavaScript to process the user's input. The result like these.



Of course, you can display the result in one alert. How to display it?

3. Practice the methods of String.

4. Create page with if statement

5. Create page with different if statement. Such as if else, if else-if else, and nested if else

6. Create page with switch statement.

7. Create page with for loops

8. Create page with while loops

9. Create ant song page with for array.

10. Create a page using array with for loop.

11. Create a page using object.

12. Create a page using drop down list.

13. Create a page using multiple selection.

14. Create a page using check box.

15. Create a page using radio box.

5 Reference

This textbook comes from many other books, paper, and electronic documents. Thanks for these document. Some of these are listed as the following.

[1] Sams Teach Yourself, HTML, CSS JavaScript Web Publishing in One Hour a Day, 7th Edition. Pearson Education,2016

[2] J.M. Gustafson, HTML5 Web Application Development By Example, Packt Publishing 2013.

[3] John Brock, Arun Gupta, Geerjan Wielenga, Java EE and HTML5 Enterprise Application Development., Oracle Press.2014.