4

Introduction to XHTML

To read between the lines was easier than to follow the text.

—Henry James

High thoughts must have high language.

—Aristophanes

Yea, from the table of my memory
I'll wipe away all trivial fond records.

—William Shakespeare

He had a wonderful talent for packing thought close, and rendering it portable.

—Thomas Babington Macaulay

OBJECTIVES

In this chapter you will learn:

- To understand important components of XHTML documents.
- To use XHTML to create web pages.
- To add images to web pages.
- To create and use hyperlinks to navigate web pages.
- To mark up lists of information.
- To create tables with rows and columns of data and control table formatting.
- To create and use forms to get user input.
- To make web pages accessible to search engines using <meta> tags.

4.1 Introduction

Welcome to the world of opportunity created by the World Wide Web. The Internet is almost four decades old, but it wasn't until the web's growth in popularity in the 1990s and the recent start of the Web 2.0 era that the explosion of opportunity we are experiencing began. Exciting new developments occur almost daily—the pace of innovation is unprecedented. In this chapter, you'll develop your own web pages. As the book proceeds, you'll create increasingly appealing and powerful web pages. Later in the book, you'll learn how to create complete web-based applications with databases and user interfaces.

This chapter begins unlocking the power of web-based application development with **XHTML**—the **Extensible HyperText Markup Language**. Later in the chapter, we introduce more sophisticated XHTML techniques such as **internal linking** for easier page navigation, **forms** for collecting information from a web-page visitor and **tables**, which are particularly useful for structuring information from **databases** (i.e., software that stores structured sets of data). In the next chapter, we discuss a technology called **Cascading Style Sheets™** (**CSS**), a technology that makes web pages more visually appealing.

Unlike procedural programming languages such as C, C++, or Java, XHTML is a **markup language** that specifies the format of the text that is displayed in a web browser such as Microsoft's Internet Explorer or Mozilla Firefox.

One key issue when using XHTML is the separation of the **presentation** of a document (i.e., the document's appearance when rendered by a browser) from the **structure** of the document's information. XHTML is based on HTML (HyperText Markup Language)—a legacy technology of the World Wide Web Consortium (W3C). In HTML, it was common to specify both the document's structure and its formatting. Formatting might specify where the browser placed an element in a web page or the fonts and colors used to display an element. The XHTML 1.0 Strict recommendation (the version of

XHTML that we use in this book) allows only a document's structure to appear in a valid XHTML document, and not its formatting. Normally, such formatting is specified with Cascading Style Sheets (Chapter 5). All our examples in this chapter are based upon the XHTML 1.0 Strict Recommendation.

4.2 Editing XHTML

In this chapter, we write XHTML in its source-code form. We create XHTML documents by typing them in a text editor (e.g., Notepad, TextEdit, vi, emacs) and saving them with either an `.html` or an `.htm` filename extension.

Good Programming Practice 4.1

Assign filenames to documents that describe their functionality. This practice can help you identify documents faster. It also helps people who want to link to a page, by giving them an easy-to-remember name. For example, if you are writing an XHTML document that contains product information, you might want to call it `products.html`.

Computers called web servers running specialized software store XHTML documents. Clients (e.g., web browsers) request specific resources such as the XHTML documents from web servers. For example, typing `www.deitel.com/books/downloads.html` into a web browser's address field requests `downloads.html` from the `books` directory on the web server running at `www.deitel.com`. We discuss web servers in detail in Chapter 21. For now, we simply place the XHTML documents on our computer and render them by opening them locally with a web browser such as Internet Explorer or Firefox.

4.3 First XHTML Example

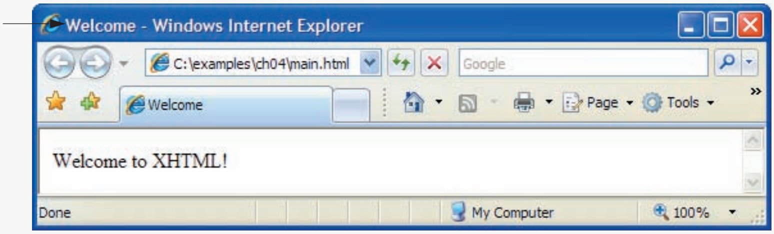
This chapter presents XHTML markup and provides screen captures that show how a browser renders (i.e., displays) the XHTML. You can download the examples from `www.deitel.com/books/iw3htp4`. Every XHTML document we show has line numbers for the reader's convenience—these line numbers are not part of the XHTML documents. As you read this book, open each XHTML document in your web browser so you can view and interact with it as it was originally intended.

Figure 4.1 is an XHTML document named `main.html`. This first example displays the message “Welcome to XHTML!” in the browser. The key line in the program is line 13, which tells the browser to display “Welcome to XHTML!” Now let us consider each line of the program.

Lines 1–3 are required in XHTML documents to conform with proper XHTML syntax. For now, copy and paste these lines into each XHTML document you create. The meaning of these lines is discussed in detail in Chapter 14.

Lines 5–6 are XHTML comments. XHTML document creators insert comments to improve markup readability and describe the content of a document. Comments also help other people read and understand an XHTML document's markup and content. Comments do not cause the browser to perform any action when the user loads the XHTML document into the web browser to view it. XHTML comments always start with `<!--` and end with `-->`. Each of our XHTML examples includes comments that specify the figure number and filename and provide a brief description of the example's purpose. Subsequent examples include comments in the markup, especially to highlight new features.

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 4.1: main.html -->
 6  <!-- First XHTML example. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Welcome</title>
10     </head>
11
12     <body>
13        <p>Welcome to XHTML!</p>
14     </body>
15  </html>
```


Welcome - Windows Internet Explorer
C:\examples\ch04\main.html
Google
Welcome
Page
Tools
Done
My Computer
100%

Welcome to XHTML!

Title bar shows contents of `title` element

Fig. 4.1 | First XHTML example.

Good Programming Practice 4.2

Place comments throughout your markup. Comments help other programmers understand the markup, assist in debugging and list useful information that you do not want the browser to render. Comments also help you understand your own markup when you revisit a document to modify or update it in the future.

XHTML markup contains text that represents the content of a document and **elements** that specify a document's structure. Some important elements of an XHTML document are the **`html`** element, the **`head`** element and the **`body`** element. The `html` element encloses the **head section** (represented by the `head` element) and the **body section** (represented by the `body` element). The head section contains information about the XHTML document, such as its **title**. The head section also can contain special document formatting instructions called **style sheets** and client-side programs called **scripts** for creating dynamic web pages. (We introduce style sheets in Chapter 5 and scripting with JavaScript in Chapter 6.) The body section contains the page's content that the browser displays when the user visits the web page.

XHTML documents delimit an element with **start** and **end** tags. A start tag consists of the element name in angle brackets (e.g., `<html>`). An end tag consists of the element name preceded by a forward slash (`/`) in angle brackets (e.g., `</html>`). In this example, lines 7 and 15 define the start and end of the `html` element. Note that the end tag in line 15 has the same name as the start tag, but is preceded by a `/` inside the angle brackets. Many start tags have **attributes** that provide additional information about an element. Browsers can use this additional information to determine how to process the element.

Each attribute has a **name** and a **value** separated by an equals sign (=). Line 7 specifies a required attribute (`xmlns`) and value (`http://www.w3.org/1999/xhtml`) for the `html` element in an XHTML document. For now, simply copy and paste the `html` element start tag in line 7 into your XHTML documents. We discuss the details of the `xmlns` attribute in Chapter 14.

Common Programming Error 4.1

Not enclosing attribute values in either single or double quotes is a syntax error. However, some web browsers may still render the element correctly.

Common Programming Error 4.2

Using uppercase letters in an XHTML element or attribute name is a syntax error. However, some web browsers may still render the element correctly.

An XHTML document divides the `html` element into two sections—head and body. Lines 8–10 define the web page's head section with a `head` element. Line 9 specifies a `title` element. This is called a **nested element** because it is enclosed in the `head` element's start and end tags. The `head` element is also a nested element because it is enclosed in the `html` element's start and end tags. The `title` element describes the web page. Titles usually appear in the **title bar** at the top of the browser window, in the browser tab that the page is displayed on, and also as the text identifying a page when users add the page to their list of **Favorites** or **Bookmarks** that enables them to return to their favorite sites. Search engines (i.e., sites that allow users to search the web) also use the `title` for indexing purposes.

Good Programming Practice 4.3

Indenting nested elements emphasizes a document's structure and promotes readability.

Common Programming Error 4.3

XHTML does not permit tags to overlap—a nested element's end tag must appear in the document before the enclosing element's end tag. For example, the nested XHTML tags `<head><title>hello</head></title>` cause a syntax error, because the enclosing `head` element's ending `</head>` tag appears before the nested `title` element's ending `</title>` tag.

Good Programming Practice 4.4

Use a consistent `title`-naming convention for all pages on a site. For example, if a site is named "Bailey's Website," then the `title` of the contact page might be "Bailey's Website - Contact." This practice can help users better understand the website's structure.

Line 12 begins the document's `body` element. The body section of an XHTML document specifies the document's content, which may include text and elements.

Some elements, such as the **paragraph element** (denoted with `<p>` and `</p>`) in line 13, mark up text for display in a browser. All the text placed between the `<p>` and `</p>` tags forms one paragraph. When the browser renders a paragraph, a blank line usually precedes and follows paragraph text.

This document ends with two end tags (lines 14–15). These tags close the `body` and `html` elements, respectively. The `</html>` tag in an XHTML document informs the browser that the XHTML markup is complete.

To open an XHTML example from this chapter, open the folder where you saved the book's examples, browse to the ch04 folder and double click the file to open it in your default web browser. At this point your browser window should appear similar to the sample screen capture shown in Fig. 4.1. (Note that we resized the browser window to save space in the book.)

4.4 W3C XHTML Validation Service

Programming web-based applications can be complex, and XHTML documents must be written correctly to ensure that browsers process them properly. To promote correctly written documents, the World Wide Web Consortium (W3C) provides a validation service (`validator.w3.org`) for checking a document's syntax. Documents can be validated by providing a URL that specifies the file's location, by uploading a file to `validator.w3.org/file-upload.html` or by pasting code directly into a text area. Uploading a file copies the file from the user's computer to another computer on the Internet. The W3C's web page indicates that the service name is **MarkUp Validation Service** and that the validation service is able to validate the syntax of XHTML documents. All the XHTML examples in this book have been validated successfully using `validator.w3.org`.

By clicking **Choose...**, users can select files on their own computers for upload. After selecting a file, clicking the **Check** button uploads and validates the file. If a document contains syntax errors, the validation service displays error messages describing the errors.

Error-Prevention Tip 4.1

Most current browsers attempt to render XHTML documents even if they are invalid. This often leads to unexpected and possibly undesirable results. Use a validation service, such as the W3C **MarkUp Validation Service***, to confirm that an XHTML document is syntactically correct.*

4.5 Headings

Some text in an XHTML document may be more important than other text. For example, the text in this section is considered more important than a footnote. XHTML provides six headings, called heading elements, for specifying the relative importance of information. Figure 4.2 demonstrates these elements (`h1` through `h6`). Heading element `h1` (line 13) is considered the most significant heading and is typically rendered in a larger font than the other five headings (lines 14–18). Each successive heading element (i.e., `h2`, `h3`, etc.) is typically rendered in a progressively smaller font.

Portability Tip 4.1

The text size used to display each heading element can vary significantly between browsers. In Chapter 5, we discuss how to control the text size and other text properties.

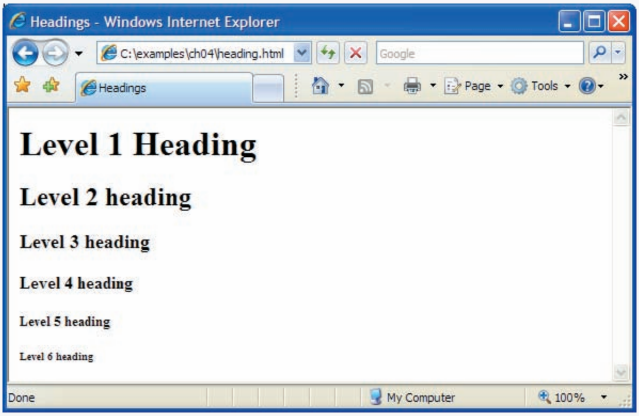
Look-and-Feel Observation 4.1

Placing a heading at the top of every XHTML page helps viewers understand the purpose of each page.

Look-and-Feel Observation 4.2

Use larger headings to emphasize more important sections of a web page.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 4.2: heading.html -->
<!-- Heading elements h1 through h6. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Headings</title>
   </head>

   <body>
      <h1>Level 1 Heading</h1>
      <h2>Level 2 heading</h2>
      <h3>Level 3 heading</h3>
      <h4>Level 4 heading</h4>
      <h5>Level 5 heading</h5>
      <h6>Level 6 heading</h6>
   </body>
</html>
```


Headings - Windows Internet Explorer
C:\examples\ch04\heading.html
Google
Headings
Page
Tools
Done
My Computer
100%

Level 1 Heading

Level 2 heading

Level 3 heading

Level 4 heading

Level 5 heading

Level 6 heading

Fig. 4.2 | Heading elements `h1` through `h6`.

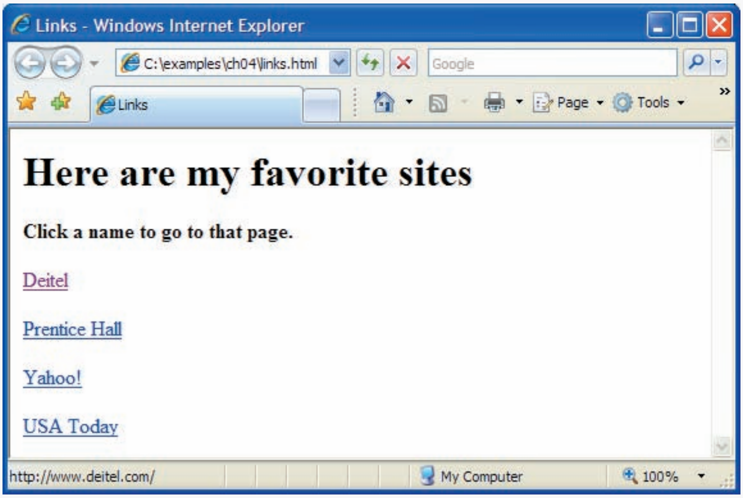
4.6 Linking

One of the most important XHTML features is the hyperlink, which references (or links to) other resources, such as XHTML documents and images. When a user clicks a hyperlink, the browser tries to execute an action associated with it (e.g., navigate to a URL, open an e-mail client, etc.). In XHTML, both text and images can act as hyperlinks. Web browsers typically underline text hyperlinks and color their text blue by default, so that users can distinguish hyperlinks from plain text. In Fig. 4.3, we create text hyperlinks to four different websites.

Line 14 introduces the **`strong`** element, which indicates that its contents has high importance. Browsers typically display such text in a bold font.

Links are created using the **`a`** (anchor) element. Line 17 defines a hyperlink to the URL assigned to attribute **`href`**, which specifies the location of a linked resource, such as

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 4.3: links.html -->
<!-- Linking to other web pages. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Links</title>
   </head>

   <body>
      <h1>Here are my favorite sites</h1>
      <p><strong>Click a name to go to that page.</strong></p>

      <!-- Create four text hyperlinks -->
      <p><a href = "http://www.deitel.com">Deitel</a></p>
      <p><a href = "http://www.prenhall.com">Prentice Hall</a></p>
      <p><a href = "http://www.yahoo.com">Yahoo!</a></p>
      <p><a href = "http://www.usatoday.com">USA Today</a></p>
   </body>
</html>
```


Links - Windows Internet Explorer
C:\examples\ch04\links.html
Google
Links
Page
Tools
http://www.deitel.com/
My Computer
100%

Here are my favorite sites

Click a name to go to that page.

Deitel

Prentice Hall

Yahoo!

USA Today

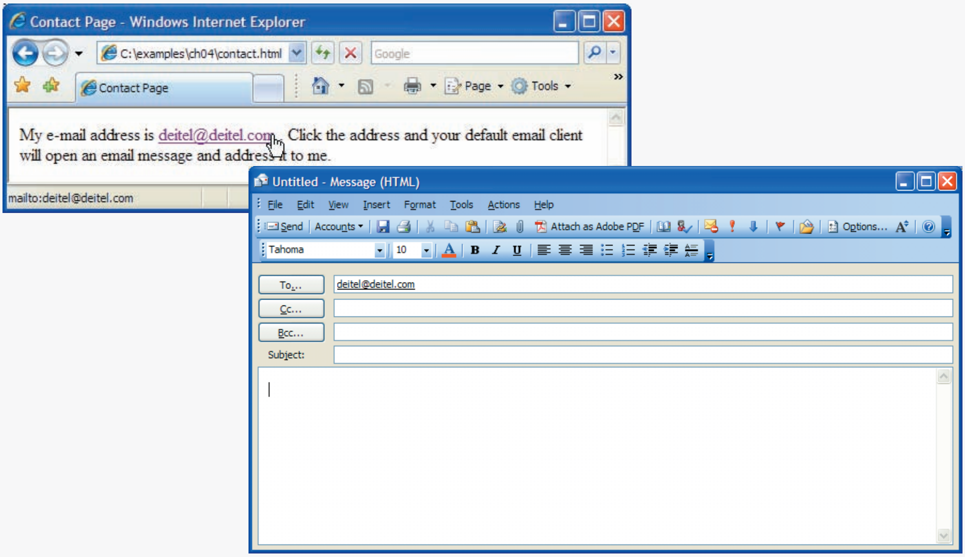
Fig. 4.3 | Linking to other web pages.

a web page, a file or an e-mail address. This particular anchor element links the text `Deitel` to a web page located at `http://www.deitel.com`. When a URL does not indicate a specific document on the website, the web server returns a default web page. This page is often called `index.html`; however, most web servers can be configured to use any file as the default web page for the site. If the web server cannot locate a requested document, it returns an error indication to the web browser, and the browser displays a web page containing an error message to the user.

Hyperlinking to an E-Mail Address

Anchors can link to e-mail addresses using a **`mailto:`** URL. When someone clicks this type of anchored link, most browsers launch the default e-mail program (e.g., Microsoft Outlook or Mozilla Thunderbird) to enable the user to write an e-mail message to the linked address. Figure 4.4 demonstrates this type of anchor. Lines 15–17 contain an e-mail link. The form of an e-mail anchor is `<a href = "mailto:`*emailAddress*`">...</a>`. In this case, we link to the e-mail address `deitel@deitel.com`.

```
1  <?xml version = "1.0" encoding = "utf-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 4.4: contact.html -->
6  <!-- Linking to an e-mail address. -->
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8     <head>
9        <title>Contact Page</title>
10    </head>
11
12    <body>
13       <p>
14          My email address is
15          <a href = "mailto:deitel@deitel.com">
16             deitel@deitel.com
17          </a>
18          . Click the address and your default email client
19           will open an e-mail message and address it to me.
20       </p>
21    </body>
22 </html>
```


Contact Page - Windows Internet Explorer
C:\examples\ch04\contact.html
Google
Contact Page
Page
Tools
Untitled - Message (HTML)
File Edit View Insert Format Tools Actions Help
Send Accounts
Attach as Adobe PDF
Options...
Tahoma
10
To...
Cc...
Bcc...
Subject:

My e-mail address is deitel@deitel.com. Click the address and your default email client will open an email message and address it to me.

mailto:deitel@deitel.com

To... deitel@deitel.com

Fig. 4.4 | Linking to an e-mail address.

4.7 Images

The examples discussed so far demonstrate how to mark up documents that contain only text. However, most web pages contain both text and images. In fact, images are an equally important, if not essential, part of web-page design. The three most popular image formats used by web developers are Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG) and Portable Network Graphics (PNG) images. Users can create images using specialized software, such as Adobe Photoshop Elements (`www.adobe.com`), G.I.M.P. (`http://www.gimp.org`) and Inkscape (`http://www.inkscape.org`). Images may also be acquired from various websites. Figure 4.5 demonstrates how to incorporate images into web pages.

Lines 14–15 use an **`img` element** to insert an image in the document. The image file's location is specified with the `img` element's **`src` attribute**. This image is located in the same directory as the XHTML document, so only the image's filename is required. Optional attributes **`width`** and **`height`** specify the image's width and height, respectively. You can scale an image by increasing or decreasing the values of the image `width` and `height` attributes. If these attributes are omitted, the browser uses the image's actual width and height. Images are measured in **pixels** ("picture elements"), which represent dots of color on the screen. Any image-editing program will have a feature that displays the dimensions, in pixels, of an image. The image in Fig. 4.5 is `92` pixels wide and `120` pixels high.

Good Programming Practice 4.5

Always include the `width` and the `height` of an image inside the `<img>` tag. When the browser loads the XHTML file, it will know immediately from these attributes how much screen space to provide for the image and will lay out the page properly, even before it downloads the image.

Performance Tip 4.1

Including the `width` and `height` attributes in an `<img>` tag can result in the browser's loading and rendering pages faster.

Common Programming Error 4.4

Entering new dimensions for an image that change its inherent width-to-height ratio distorts the appearance of the image. For example, if your image is 200 pixels wide and 100 pixels high, you should ensure that any new dimensions have a 2:1 width-to-height ratio.

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 4.5: picture.html -->
 6  <!-- Images in XHTML files. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Images</title>
10     </head>
11
```

Fig. 4.5 | Images in XHTML files. (Part 1 of 2.)

```
12       <body>
13          <p>
14             <img src = "cpphtp6.jpg" width = "92" height = "120"
15                alt = "C++ How to Program book cover" />
16             <img src = "jhtp.jpg" width = "92" height = "120"
17                alt = "Java How to Program book cover" />
18          </p>
19       </body>
20    </html>
```

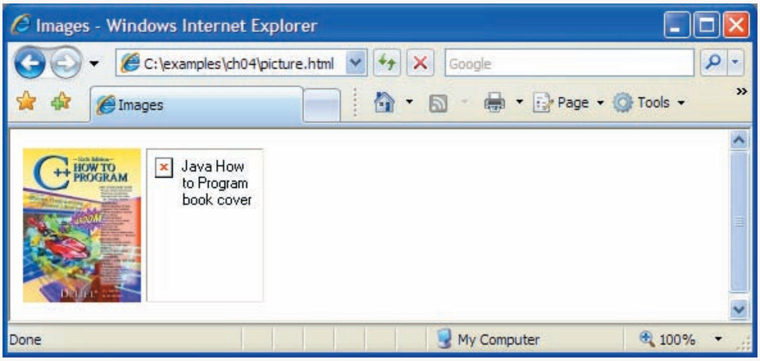

Images - Windows Internet Explorer
C:\examples\ch04\picture.html
Google
Images
Page
Tools
Java How to Program book cover
Done
My Computer
100%

Fig. 4.5 | Images in XHTML files. (Part 2 of 2.)

Every `img` element in an XHTML document must have an **`alt` attribute**. If a browser cannot render an image, the browser displays the `alt` attribute's value. A browser may not be able to render an image for several reasons. It may not support images—as is the case with a **text-based browser** (i.e., a browser that can display only text)—or the client may have disabled image viewing to reduce download time. Figure 4.5 shows Internet Explorer 7 rendering a red x symbol and displaying the `alt` attribute's value, signifying that the image (`jhtp.jpg`) cannot be found.

The `alt` attribute helps you create **accessible** web pages for users with disabilities, especially those with vision impairments who use text-based browsers. Specialized software called a **speech synthesizer** can "speak" the `alt` attribute's value so that a user with a visual impairment knows what the browser is displaying.

Some XHTML elements (called **empty elements**) contain only attributes and do not mark up text (i.e., text is not placed between the start and end tags). Empty elements (e.g., `img`) must be terminated, either by using the **forward slash character** (/) inside the closing right angle bracket (>) of the start tag or by explicitly including the end tag. When using the forward slash character, we add a space before it to improve readability (as shown at the ends of lines 15 and 17). Rather than using the forward slash character, lines 16–17 could be written with a closing `</img>` tag as follows:

```
<img src = "jhtp.jpg" width = "92" height = "120"
   alt = "Java How to Program book cover"></img>
```

Using Images as Hyperlinks

By using images as hyperlinks, web developers can create graphical web pages that link to other resources. In Fig. 4.6, we create six different image hyperlinks.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 4.6: nav.html -->
<!-- Images as link anchors. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Navigation Bar</title>
   </head>

   <body>
      <p>
         <a href = "links.html">
            <img src = "buttons/links.jpg" width = "65"
               height = "50" alt = "Links Page" />
         </a>

         <a href = "list.html">
            <img src = "buttons/list.jpg" width = "65"
               height = "50" alt = "List Example Page" />
         </a>

         <a href = "contact.html">
            <img src = "buttons/contact.jpg" width = "65"
               height = "50" alt = "Contact Page" />
         </a>

         <a href = "table1.html">
            <img src = "buttons/table.jpg" width = "65"
               height = "50" alt = "Table Page" />
         </a>

         <a href = "form.html">
            <img src = "buttons/form.jpg" width = "65"
               height = "50" alt = "Feedback Form" />
         </a>
      </p>
   </body>
</html>
```

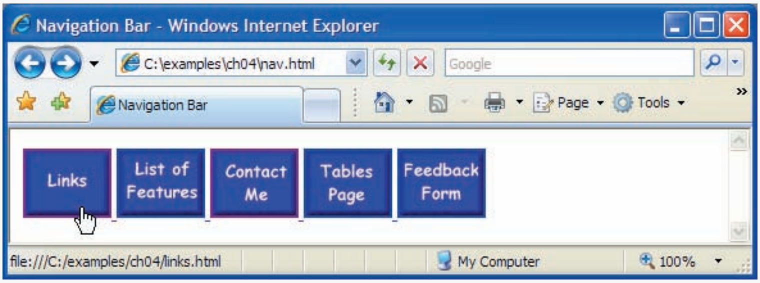
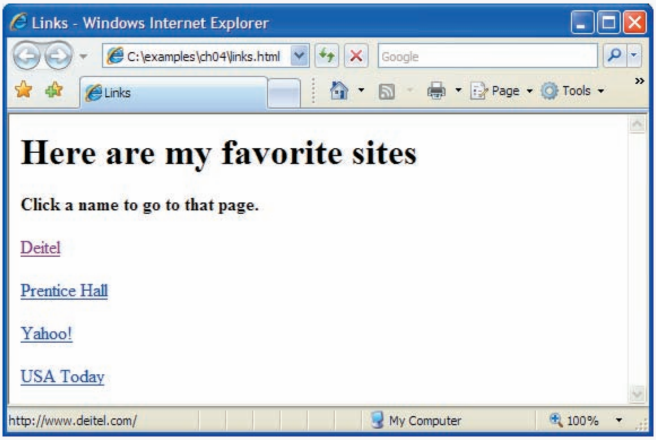

Navigation Bar - Windows Internet Explorer
C:\examples\ch04\nav.html
Google
Navigation Bar
Page
Tools
Links
List of Features
Contact Me
Tables Page
Feedback Form
file:///C:/examples/ch04/links.html
My Computer
100%

Fig. 4.6 | Images as link anchors. (Part 1 of 2.)



Links - Windows Internet Explorer
C:\examples\ch04\links.html
Google
Links
Page
Tools
http://www.deitel.com/
My Computer
100%

Here are my favorite sites

Click a name to go to that page.

Deitel

Prentice Hall

Yahoo!

USA Today

Fig. 4.6 | Images as link anchors. (Part 2 of 2.)

Lines 14–17 create an **image hyperlink** by nesting an `img` element in an anchor (`a`) element. The value of the `img` element's `src` attribute value specifies that this image (`links.jpg`) resides in a directory named `buttons`. The `buttons` directory and the XHTML document are in the same directory. Images from other web documents also can be referenced by setting the `src` attribute to the name and location of the image. Note that if you're hosting a publicly available web page that uses an image from another site, you should get permission to use the image and host a copy of image on your own website. If you refer to an image on another website, the browser has to request the image resource from the other site's server. Clicking an image hyperlink takes a user to the web page specified by the surrounding anchor element's `href` attribute. Notice that when the mouse hovers over a link of any kind, the URL that the link points to is displayed in the status bar at the bottom of the browser window.

4.8 Special Characters and Horizontal Rules

When marking up text, certain characters or symbols (e.g., <) may be difficult to embed directly into an XHTML document. Some keyboards do not provide these symbols, or the presence of these symbols may cause syntax errors. For example, the markup

```
<p>if x < 10 then increment x by 1</p>
```

results in a syntax error because it uses the less-than character (<), which is reserved for start tags and end tags such as <p> and </p>. XHTML provides **character entity references** (in the form &*code*;) for representing special characters. We could correct the previous line by writing

```
<p>if x &lt; 10 then increment x by 1</p>
```

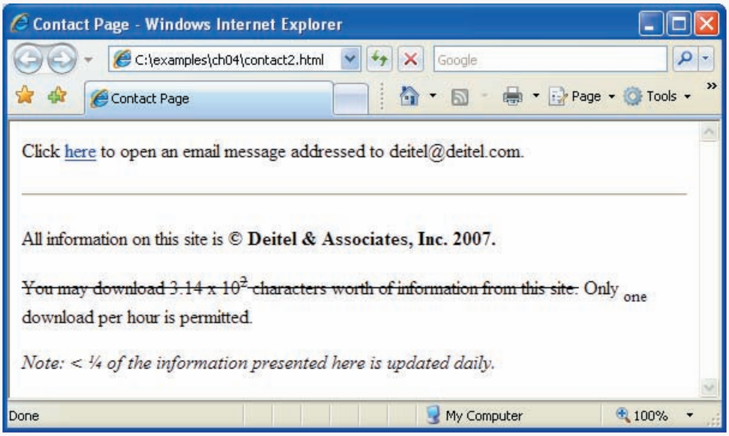
which uses the character entity reference **`&lt;`** for the less-than symbol (<).

Figure 4.7 demonstrates how to use special characters in an XHTML document. For a list of special characters, see Appendix A, XHTML Special Characters.

Lines 24–25 contain other special characters, which can be expressed as either character entity references (coded using word abbreviations such as &amp; for ampersand and &copy; for copyright) or **numeric character references**—decimal or **hexadecimal** (**hex**) values representing special characters. For example, the & character is represented in decimal and hexadecimal notation as &#38; and &#x26;, respectively. Hexadecimal numbers are base 16 numbers—digits in a hexadecimal number have values from 0 to 15 (a total of 16 different values). The letters A–F represent the hexadecimal digits corresponding to decimal values 10–15. Thus in hexadecimal notation we can have numbers like 876 consisting solely of decimal-like digits, numbers like DA19F consisting of digits and letters, and numbers like DCB consisting solely of letters. We discuss hexadecimal numbers in detail in Appendix E, Number Systems.

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 4.7: contact2.html -->
 6  <!-- Inserting special characters. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Contact Page</title>
10     </head>
11
12     <body>
13        <p>
14           Click
15           <a href = "mailto:deitel@deitel.com">here</a>
16           to open an email message addressed to
17           deitel@deitel.com.
18        </p>
19
20        <hr /> <!-- inserts a horizontal rule -->
21
22        <!-- special characters are entered -->
23        <!-- using the form &code; -->
24        <p>All information on this site is <strong>&copy;
25           Deitel &amp; Associates, Inc. 2007.</strong></p>
26
27        <!-- to strike through text use <del> tags -->
28        <!-- to subscript text use <sub> tags -->
29        <!-- to superscript text use <sup> tags -->
30        <!-- these tags are nested inside other tags -->
31        <p><del>You may download 3.14 x 10<sup>2</sup>
32           characters worth of information from this site.</del>
33           Only <sub>one</sub> download per hour is permitted.</p>
34        <p><em>Note: &lt; &frac14; of the information
35           presented here is updated daily.</em></p>
36     </body>
37  </html>
```

Fig. 4.7 | Inserting special characters. (Part 1 of 2.)


Contact Page - Windows Internet Explorer
C:\examples\ch04\contact2.html
Google
Contact Page
Page
Tools
Done
My Computer
100%

Click here to open an email message addressed to deitel@deitel.com.

All information on this site is © **Deitel & Associates, Inc. 2007**.

~~You may download 3.14 x 10^2 characters worth of information from this site.~~ Only $_{\text{one}}$ download per hour is permitted.

Note: < ¼ of the information presented here is updated daily.

Fig. 4.7 | Inserting special characters. (Part 2 of 2.)

In lines 31–33, we introduce four new elements. Most browsers render the `del` element as strike-through text. With this format users can easily indicate document revisions. To superscript text (i.e., raise text above the baseline and decreased font size) or subscript text (i.e., lower text below the baseline and decreased font size), use the `sup` or `sub` element, respectively. The paragraph in lines 34–35 contains an `em` element, which indicates that its contents should be emphasized. Browsers usually render `em` elements in an italic font. We also use character entity reference `&lt;` for a less-than sign and `&frac14;` for the fraction 1/4 (line 34).

In addition to special characters, this document introduces a horizontal rule, indicated by the `<hr />` tag in line 22. Most browsers render a horizontal rule as a horizontal line with a blank line above and below it.

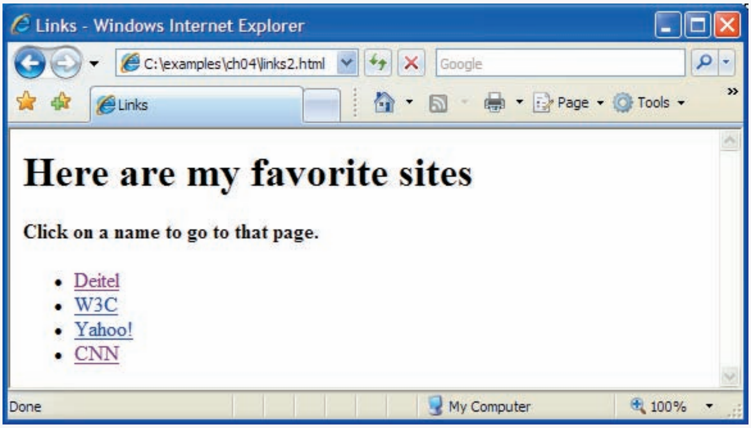
4.9 Lists

Up to this point, we have presented basic XHTML elements and attributes for linking to resources, creating headings, using special characters and incorporating images. In this section, we discuss how to organize information on a web page using lists. In the next section, we introduce another feature for organizing information, called a table. Figure 4.8 displays text in an unordered list (i.e., a list that does not order its items by letter or number). The unordered list element `ul` creates a list in which each item begins with a bullet symbol (called a disc). Each entry in an unordered list (element `ul` in line 17) is an `li` (list item) element (lines 19–22). Most web browsers render each `li` element on a new line with a bullet symbol indented from the beginning of the line.

```
1  <?xml version = "1.0" encoding = "utf-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
```

Fig. 4.8 | Unordered list containing hyperlinks. (Part 1 of 2.)

```
<!-- Fig. 4.8: links2.html -->
<!-- Unordered list containing hyperlinks. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Links</title>
   </head>

   <body>
      <h1>Here are my favorite sites</h1>
      <p><strong>Click on a name to go to that page.</strong></p>

      <!-- create an unordered list -->
      <ul>
         <!-- add four list items -->
         <li><a href = "http://www.deitel.com">Deitel</a></li>
         <li><a href = "http://www.w3.org">W3C</a></li>
         <li><a href = "http://www.yahoo.com">Yahoo!</a></li>
         <li><a href = "http://www.cnn.com">CNN</a></li>
       </ul>
   </body>
</html>
```



Links - Windows Internet Explorer
C:\examples\ch04\links2.html
Google
Links
Page
Tools
Done
My Computer
100%

Here are my favorite sites

Click on a name to go to that page.

- Deitel
- W3C
- Yahoo!
- CNN

Fig. 4.8 | Unordered list containing hyperlinks. (Part 2 of 2.)

Nested Lists

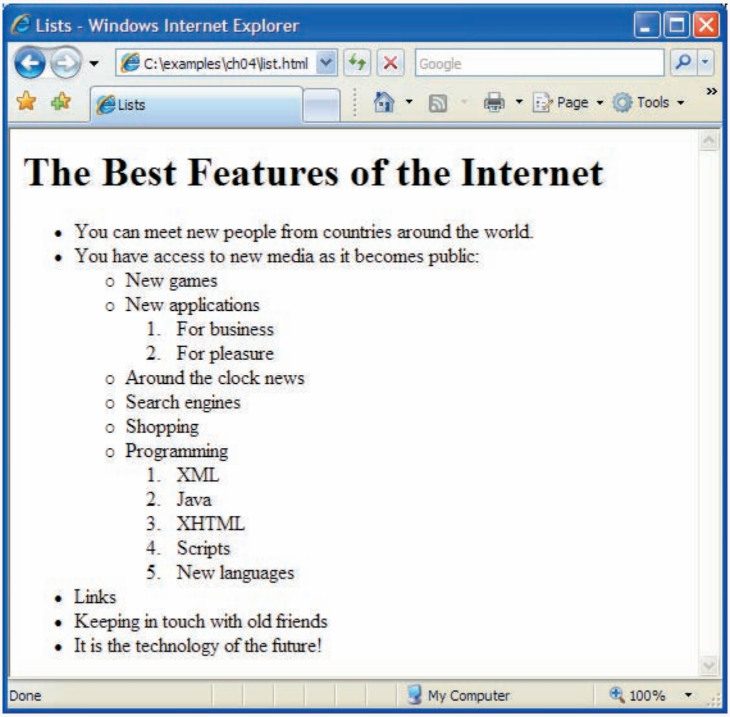
Lists may be nested to represent hierarchical relationships, as in an outline format. Figure 4.9 demonstrates nested lists and **ordered lists**. The ordered list element `ol` creates a list in which each item begins with a number.

A web browser indents each nested list to indicate a hierarchical relationship. The first ordered list begins at line 30. Items in an ordered list are enumerated one, two, three and so on. Nested ordered lists are enumerated in the same manner. The items in the outermost unordered list (line 16) are preceded by discs. List items nested inside the unordered list of line 16 are preceded by circular bullets. Although not demonstrated in this example, subsequent nested list items are preceded by square bullets.

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 4.9: list.html -->
 6  <!-- Nested and ordered lists. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Lists</title>
10     </head>
11
12     <body>
13        <h1>The Best Features of the Internet</h1>
14
15        <!-- create an unordered list -->
16        <ul>
17           <li>You can meet new people from countries around
18              the world.</li>
19           <li>
20              You have access to new media as it becomes public:
21
22              <!-- this starts a nested list, which uses a -->
23              <!-- modified bullet. The list ends when you -->
24              <!-- close the <ul> tag. -->
25              <ul>
26                 <li>New games</li>
27                 <li>New applications
28
29                    <!-- nested ordered list -->
30                    <ol>
31                       <li>For business</li>
32                       <li>For pleasure</li>
33                    </ol>
34                 </li> <!-- ends line 27 new applications li -->
35
36                 <li>Around the clock news</li>
37                 <li>Search engines</li>
38                 <li>Shopping</li>
39                 <li>Programming
40
41                    <!-- another nested ordered list -->
42                    <ol>
43                       <li>XML</li>
44                       <li>Java</li>
45                       <li>XHTML</li>
46                       <li>Scripts</li>
47                       <li>New languages</li>
48                    </ol>
49                 </li> <!-- ends programming li of line 39 -->
50              </ul> <!-- ends the nested list of line 25 -->
51           </li>
52
```

Fig. 4.9 | Nested and ordered lists. (Part 1 of 2.)

```
         <li>Links</li>
         <li>Keeping in touch with old friends</li>
         <li>It is the technology of the future!</li>
      </ul> <!-- ends the unordered list of line 16 -->
   </body>
</html>
```



The Best Features of the Internet

- You can meet new people from countries around the world.
- You have access to new media as it becomes public:
  - New games
  - New applications
    1. For business
    2. For pleasure
  - Around the clock news
  - Search engines
  - Shopping
  - Programming
    1. XML
    2. Java
    3. XHTML
    4. Scripts
    5. New languages
- Links
- Keeping in touch with old friends
- It is the technology of the future!

Fig. 4.9 | Nested and ordered lists. (Part 2 of 2.)

4.10 Tables

Tables are frequently used to organize data into rows and columns. Our first example (Fig. 4.10) creates a table with six rows and two columns to display price information for fruit.

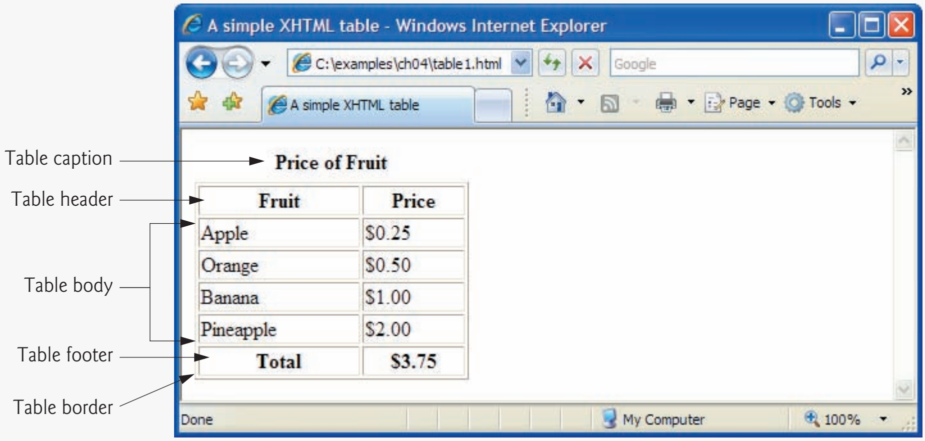
Tables are defined with the **`table`** element (lines 15–62). Lines 15–17 specify the start tag for a `table` element that has several attributes. The **`border`** attribute specifies the table's border width in pixels. To create a table without a border, set `border` to `"0"`. This example assigns attribute **`width`** the value `"40%"` to set the table's width to 40 percent of the browser's width. A developer can also set attribute `width` to a specified number of pixels. Try resizing the browser window to see how the width of the window affects the width of the table.

As its name implies, attribute **`summary`** (lines 16–17) describes the table's contents. Speech devices use this attribute to make the table more accessible to users with visual impairments. The **`caption`** element (line 21) describes the table's content and helps text-

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 4.10: table1.html -->
<!-- Creating a basic table. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>A simple XHTML table</title>
   </head>

   <body>

      <!-- the <table> tag opens a table -->
      <table border = "1" width = "40%"
         summary = "This table provides information about
            the price of fruit">

         <!-- the <caption> tag summarizes the table's -->
         <!-- contents (this helps the visually impaired) -->
         <caption><strong>Price of Fruit</strong></caption>

         <!-- the <thead> section appears first in the table -->
         <!-- it formats the table header area -->
         <thead>
            <tr> <!-- <tr> inserts a table row -->
               <th>Fruit</th> <!-- insert a heading cell -->
               <th>Price</th>
            </tr>
         </thead>

         <!-- the <tfoot> section appears last in the table -->
         <!-- it formats the table footer -->
         <tfoot>
            <tr>
               <th>Total</th>
               <th>$3.75</th>
            </tr>
         </tfoot>

         <!-- all table content is enclosed -->
         <!-- within the <tbody> -->
         <tbody>
            <tr>
               <td>Apple</td> <!-- insert a data cell -->
               <td>$0.25</td>
            </tr>
            <tr>
               <td>Orange</td>
               <td>$0.50</td>
            </tr>
            <tr>
               <td>Banana</td>
```

Fig. 4.10 | Creating a basic table. (Part 1 of 2.)

```
54                <td>$1.00</td>
55             </tr>
56             <tr>
57                <td>Pineapple</td>
58                <td>$2.00</td>
59             </tr>
60          </tbody>
61       </table>
62
63    </body>
64 </html>
```



Price of Fruit

Fruit	Price
Apple	$0.25
Orange	$0.50
Banana	$1.00
Pineapple	$2.00
Total	$3.75

Fig. 4.10 | Creating a basic table. (Part 2 of 2.)

based browsers interpret the table data. Text inside the `<caption>` tag is rendered above the table by most browsers. Attribute `summary` and element `caption` are two of the many XHTML features that make web pages more accessible to users with disabilities.

A table has three distinct sections—head, body and foot. The head section (or header cell) is defined with a **`thead`** element (lines 25–30), which contains header information such as column names. Each **`tr`** element (lines 26–29) defines an individual table row. The columns in the head section are defined with **`th`** elements. Most browsers center text formatted by `th` (table header column) elements and display them in bold. Table header elements are nested inside table row elements (lines 27–28).

The body section, or table body, contains the table's primary data. The table body (lines 43–60) is defined in a **`tbody`** element. In the body, each `tr` element specifies one row. Data cells contain individual pieces of data and are defined with **`td`** (table data) elements in each row.

The foot section (lines 34–39) is defined with a **`tfoot`** (table foot) element. The text placed in the footer commonly includes calculation results and footnotes. Like other sections, the foot section can contain table rows, and each row can contain cells. As in the head section, cells in the foot section are created using `th` elements, instead of the `td` elements used in the table body. Note that the table foot section must be above the body section in the code, but the table foot displays at the bottom of the table.

Using **rowspan** *and* **colspan**

Figure 4.10 explored a basic table's structure. Figure 4.11 presents another table example and introduces two new attributes that allow you to build more complex tables.

The table begins in line 15. Table cells are sized to fit the data they contain. Document authors can create larger data cells using the attributes **rowspan** and **colspan**. The values assigned to these attributes specify the number of rows or columns occupied by a cell. The th element at lines 23–26 uses the attribute rowspan = "2" to allow the cell containing the picture of the camel to use two vertically adjacent cells (thus the cell *spans* two rows). The th element in lines 29–32 uses the attribute colspan = "4" to widen the header cell (containing Camelid comparison and Approximate as of 6/2007) to span four cells.

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 4.11: table2.html -->
 6  <!-- Complex XHTML table. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Tables</title>
10     </head>
11
12     <body>
13        <h1>Table Example Page</h1>
14
15        <table border = "1">
16           <caption>Here is a more complex sample table.</caption>
17
18           <thead>
19              <!-- rowspans and colspans merge the specified -->
20              <!-- number of cells vertically or horizontally -->
21              <tr>
22                 <!-- merge two rows -->
23                 <th rowspan = "2">
24                    <img src = "camel.gif" width = "205"
25                       height = "167" alt = "Picture of a camel" />
26                 </th>
27
28                 <!-- merge four columns -->
29                 <th colspan = "4">
30                    <h1>Camelid comparison</h1>
31                    <p>Approximate as of 6/2007</p>
32                 </th>
33              </tr>
34              <tr>
35                 <th># of Humps</th>
36                 <th>Indigenous region</th>
37                 <th>Spits?</th>
38                 <th>Produces Wool?</th>
39              </tr>
40           </thead>
```

Fig. 4.11 | Complex XHTML table. (Part 1 of 2.)

```
            <tbody>
               <tr>
                  <th>Camels (bactrian)</th>
                  <td>2</td>
                  <td>Africa/Asia</td>
                  <td>Yes</td>
                  <td>Yes</td>
               </tr>
               <tr>
                  <th>Llamas</th>
                  <td>1</td>
                  <td>Andes Mountains</td>
                  <td>Yes</td>
                  <td>Yes</td>
               </tr>
            </tbody>
         </table>
      </body>
   </html>
```

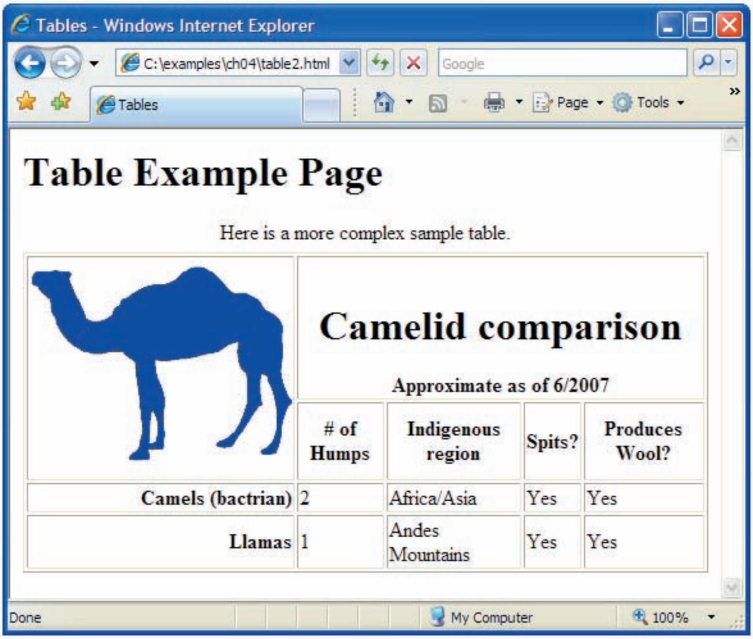

Tables - Windows Internet Explorer
C:\examples\ch04\table2.html
Google
Tables
Page
Tools
Done
My Computer
100%

Table Example Page

Here is a more complex sample table.

	Camelid comparison Approximate as of 6/2007			
	# of Humps	Indigenous region	Spits?	Produces Wool?
Camels (bactrian)	2	Africa/Asia	Yes	Yes
Llamas	1	Andes Mountains	Yes	Yes

Fig. 4.11 | Complex XHTML table. (Part 2 of 2.)

4.11 Forms

When browsing websites, users often need to provide such information as search keywords, e-mail addresses and zip codes. XHTML provides a mechanism, called a form, for collecting data from a user.

Data that users enter on a web page is normally sent to a web server that provides access to a site's resources (e.g., XHTML documents, images). These resources are located either on the same machine as the web server or on a machine that the web server can access through the network. When a browser requests a web page or file that is located on a server, the server processes the request and returns the requested resource. A request contains the name and path of the desired resource and the method of communication (called a **protocol**). XHTML documents use the Hypertext Transfer Protocol (HTTP).

Figure 4.12 is a simple form that sends data to the web server, which passes the form data to a program. The program processes the data received from the web server and typically returns information to the web server. The web server then sends the information as an XHTML document to the web browser. We discuss web servers in Chapter 21. [*Note:* This example demonstrates client-side functionality. If the form is submitted (by clicking **Submit**), nothing will happen, because we don't yet know how to process the form data. In later chapters, we present the server-side programming (e.g., in ASP.NET, JavaServer Faces, and PHP) necessary to process information entered into a form.]

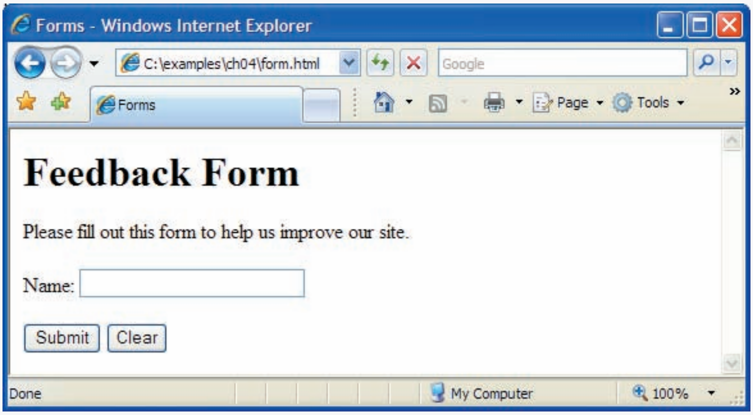
Forms can contain visual and nonvisual components. Visual components include clickable buttons and other graphical user interface components with which users interact. Nonvisual components, called **hidden inputs**, store any data that you specify, such as e-mail addresses and XHTML document filenames that act as links. The form is defined in lines 21–46 by a **form** element.

Attribute **method** (line 21) specifies how the form's data is sent to the web server. Using method = "**post**" appends form data to the browser request, which contains the protocol (HTTP) and the requested resource's URL. This method of passing data to the server is transparent—the user doesn't see the data after the form is submitted. The other possible value, method = "**get**", appends the form data directly to the end of the URL of the script, where it is visible in the browser's **Address** field. The *post* and *get* methods for sending form data are discussed in detail in Chapter 21, Web Servers (IIS and Apache).

```
1   <?xml version = "1.0" encoding = "utf-8"?>
2   <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5   <!-- Fig. 4.12: form.html -->
6   <!-- Form with hidden fields and a text box. -->
7   <html xmlns = "http://www.w3.org/1999/xhtml">
8      <head>
9         <title>Forms</title>
10     </head>
11
12     <body>
13        <h1>Feedback Form</h1>
14
15        <p>Please fill out this form to help
16           us improve our site.</p>
17
18        <!-- this tag starts the form, gives the -->
19        <!-- method of sending information and the -->
20        <!-- location of form script -->
21        <form method = "post" action = "">
22           <p>
```

Fig. 4.12 | Form with hidden fields and a text box. (Part 1 of 2.)

```
23             <!-- hidden inputs contain non-visual -->
24             <!-- information -->
25             <input type = "hidden" name = "recipient"
26                value = "deitel@deitel.com" />
27             <input type = "hidden" name = "subject"
28                value = "Feedback Form" />
29             <input type = "hidden" name = "redirect"
30                value = "main.html" />
31          </p>
32
33          <!-- <input type = "text"> inserts a text box -->
34          <p><label>Name:
35                <input name = "name" type = "text" size = "25"
36                   maxlength = "30" />
37             </label></p>
38
39          <p>
40             <!-- input types "submit" and "reset" insert -->
41             <!-- buttons for submitting and clearing the -->
42             <!-- form's contents -->
43             <input type = "submit" value = "Submit" />
44             <input type = "reset" value = "Clear" />
45          </p>
46       </form>
47    </body>
48 </html>
```



Feedback Form

Please fill out this form to help us improve our site.

Name:

Submit Clear

Fig. 4.12 | Form with hidden fields and a text box. (Part 2 of 2.)

The **action** attribute in the <form> tag in line 21 specifies the URL of a script on the web server that will be invoked to process the form's data. Since we haven't introduced server-side programming yet, we leave this attribute empty for now.

Lines 25–44 define six **input** elements that specify data to provide to the script that processes the form (also called the **form handler**). There are several types of input elements. An input's type is determined by its **type attribute**. This form uses a text input, a submit input, a reset input and three hidden inputs.

The **text** input in lines 35–36 inserts a **text box** in the form. Users can type data in text boxes. The `label` element (lines 34–37) provides users with information about the `input` element's purpose. The `input` element's **`size`** attribute specifies the number of characters visible in the text box. Optional attribute **`maxlength`** limits the number of characters input into the text box—in this case, the user is not permitted to type more than `30` characters.

Look-and-Feel Observation 4.3

Include a `label` element for each form element to help users determine the purpose of each form element.

Two `input` elements in lines 43–44 create two buttons. The **`submit`** `input` element is a button. When the `submit` button is pressed, the user is sent to the location specified in the `form`'s `action` attribute. The **`value`** attribute sets the text displayed on the button. The **`reset`** `input` element allows a user to reset all `form` elements to their default values. The `value` attribute of the `reset` `input` element sets the text displayed on the button (the default value is **Reset** if you omit the `value` attribute).

The three `input` elements in lines 25–30 have the `type` attribute `hidden`, which allows you to send form data that is not input by a user. The three hidden inputs are an e-mail address to which the data will be sent, the e-mail's subject line and a URL for the browser to open after submission of the form. Two other `input` attributes are **`name`**, which identifies the `input` element, and **`value`**, which provides the value that will be sent (or posted) to the web server.

Good Programming Practice 4.6

Place hidden `input` elements at the beginning of a form, immediately after the opening `<form>` tag. This placement allows document authors to locate hidden `input` elements quickly.

Additional Form Elements

In the previous example, you saw basic elements of XHTML forms. Now that you know the general structure of a form, we introduce elements and attributes for creating more complex forms. Figure 4.13 contains a form that solicits user feedback about a website.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 4.13: form2.html -->
<!-- Form using a variety of components. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>More Forms</title>
   </head>

   <body>
      <h1>Feedback Form</h1>
```

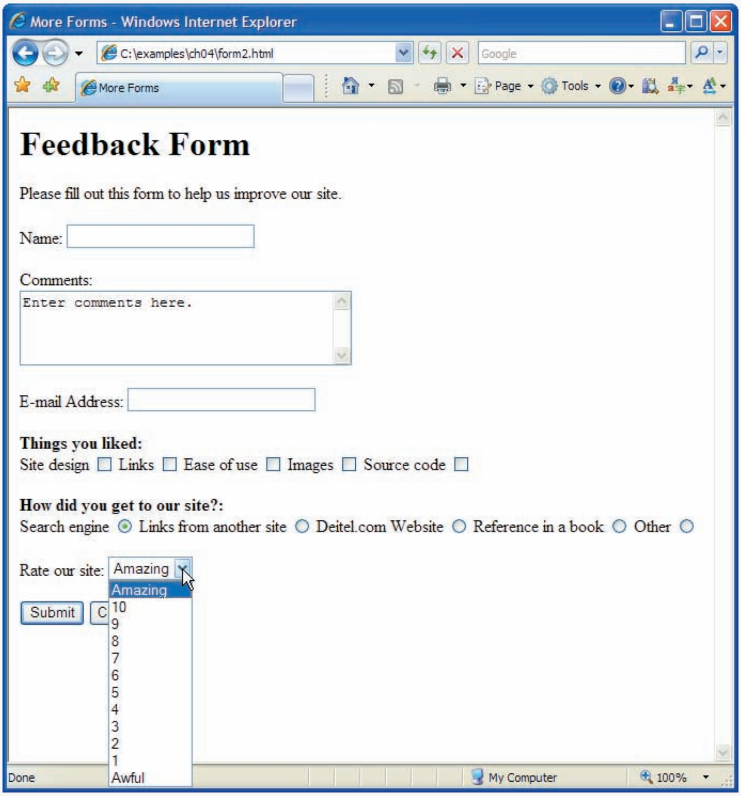
Fig. 4.13 | Form using a variety of components. (Part 1 of 4.)

```
      <p>Please fill out this form to help
         us improve our site.</p>

      <form method = "post" action = "">
         <p>
            <input type = "hidden" name = "recipient"
               value = "deitel@deitel.com" />
            <input type = "hidden" name = "subject"
               value = "Feedback Form" />
            <input type = "hidden" name = "redirect"
               value = "main.html" />
         </p>

         <p><label>Name:
               <input name = "name" type = "text" size = "25" />
            </label></p>

         <!-- <textarea> creates a multiline textbox -->
         <p><label>Comments:<br />
            <textarea name = "comments"
                rows = "4" cols = "36">Enter comments here.</textarea>
         </label></p>

         <!-- <input type = "password"> inserts a -->
         <!-- textbox whose display is masked with -->
         <!-- asterisk characters -->
         <p><label>E-mail Address:
            <input name = "email" type = "password" size = "25" />
         </label></p>

         <p>
            <strong>Things you liked:</strong><br />

            <label>Site design
               <input name = "thingsliked" type = "checkbox"
                  value = "Design" /></label>
            <label>Links
               <input name = "thingsliked" type = "checkbox"
                  value = "Links" /></label>
            <label>Ease of use
               <input name = "thingsliked" type = "checkbox"
                  value = "Ease" /></label>
            <label>Images
               <input name = "thingsliked" type = "checkbox"
                  value = "Images" /></label>
            <label>Source code
               <input name = "thingsliked" type = "checkbox"
                  value = "Code" /></label>
         </p>

         <!-- <input type = "radio" /> creates a radio -->
         <!-- button. The difference between radio buttons -->
```

Fig. 4.13 | Form using a variety of components. (Part 2 of 4.)

```
         <!-- and checkboxes is that only one radio button -->
         <!-- in a group can be selected. -->
         <p>
            <strong>How did you get to our site?:</strong><br />

            <label>Search engine
               <input name = "howtosite" type = "radio"
                  value = "search engine" checked = "checked" /></label>
            <label>Links from another site
               <input name = "howtosite" type = "radio"
                  value = "link" /></label>
            <label>Deitel.com Website
               <input name = "howtosite" type = "radio"
                  value = "deitel.com" /></label>
            <label>Reference in a book
               <input name = "howtosite" type = "radio"
                  value = "book" /></label>
            <label>Other
               <input name = "howtosite" type = "radio"
                  value = "other" /></label>
         </p>

         <p>
            <label>Rate our site:

                <!-- the <select> tag presents a drop-down -->
                <!-- list with choices indicated by the -->
                <!-- <option> tags -->
               <select name = "rating">
                  <option selected = "selected">Amazing</option>
                  <option>10</option>
                  <option>9</option>
                  <option>8</option>
                  <option>7</option>
                  <option>6</option>
                  <option>5</option>
                  <option>4</option>
                  <option>3</option>
                  <option>2</option>
                  <option>1</option>
                  <option>Awful</option>
               </select>
            </label>
         </p>

         <p>
            <input type = "submit" value = "Submit" />
            <input type = "reset" value = "Clear" />
         </p>
      </form>
   </body>
</html>
```

Fig. 4.13 | Form using a variety of components. (Part 3 of 4.)


More Forms - Windows Internet Explorer
C:\examples\ch04\form2.html
Google
More Forms
Page
Tools
Done
My Computer
100%

Feedback Form

Please fill out this form to help us improve our site.

Name:

Comments:

```
Enter comments here.
```

E-mail Address:

Things you liked:
Site design □ Links □ Ease of use □ Images □ Source code □

How did you get to our site?:
Search engine ● Links from another site ○ Deitel.com Website ○ Reference in a book ○ Other ○

Rate our site: Amazing

Amazing
10
9
8
7
6
5
4
3
2
1
Awful

Submit C

Fig. 4.13 | Form using a variety of components. (Part 4 of 4.)

In line 32, we introduce the **br** element, which most browsers render as a =line break. Any markup or text following a `br` element is rendered on the next line. Like the `img` element, `br` is an example of an empty element terminated with a forward slash. We add a space before the forward slash to enhance readability.

The **textarea** element (lines 33–34) inserts a multiline text box, called a text area, into the form. The number of rows is specified with the **rows** attribute, and the number of columns (i.e., characters per line) is specified with the **cols** attribute. In this example, the `textarea` is four rows high and 36 characters wide. To display default text in the text area, place the text between the `<textarea>` and `</textarea>` tags. Default text can be specified in other `input` types, such as text boxes, by using the `value` attribute.

The **password** input in line 41 inserts a password box with the specified `size` (maximum number of characters allowed). A password box allows users to enter sensitive information, such as credit card numbers and passwords, by “masking” the information input with asterisks (*). The actual value input is sent to the web server, not the characters that mask the input.

Lines 47–61 introduce the checkbox `form` element. Checkboxes enable users to select from a set of options. When a user selects a checkbox, a check mark appears in the checkbox. Otherwise, the checkbox remains empty. Each **"checkbox"** `input` creates a new checkbox. Checkboxes can be used individually or in groups. Checkboxes that belong to a group are assigned the same name (in this case, "thingsliked").

Common Programming Error 4.5

When your form has several checkboxes with the same name, you must make sure that they have different values, or the scripts running on the web server will not be able to distinguish them.

After the checkboxes, we present two more ways to allow the user to make choices. In this example, we introduce two new input types. The first type is the radio button (lines 71–85) specified with type **"radio"**. Radio buttons are similar to checkboxes, except that only one radio button in a group of radio buttons may be selected at any time. The radio buttons in a group all have the same name attributes and are distinguished by their different value attributes. The attribute–value pair checked = "checked" (line 73) indicates which radio button, if any, is selected initially. The checked attribute also applies to checkboxes.

Common Programming Error 4.6

Not setting the name attributes of the radio buttons in a form to the same name is a logic error because it lets the user select all of them at the same time.

The **select** element (lines 94–107) provides a drop-down list from which the user can select an item. The name attribute identifies the drop-down list. The **option** elements (lines 95–106) add items to the drop-down list. The option element's **selected** attribute specifies which item initially is displayed as the selected item in the select element. If no option element is marked as selected, the browser selects the first option by default.

4.12 Internal Linking

Earlier in the chapter, we discussed how to hyperlink one web page to another. Figure 4.14 introduces internal linking—a mechanism that enables the user to jump between locations in the same document. Internal linking is useful for long documents that contain many sections. Clicking an internal link enables users to find a section without scrolling through the entire document.

Line 14 contains a tag with the id attribute (set to "features") for an internal hyperlink. To link to a tag with this attribute inside the same web page, the href attribute of an anchor element includes the id attribute value preceded by a pound sign (as in #features). Line 56 contains a hyperlink with the id features as its target. Selecting this hyperlink in a web browser scrolls the browser window to the h1 tag in line 14. Note that you may have to resize your browser to a small window and scroll down before clicking the link to see the browser scroll to the h1 element.

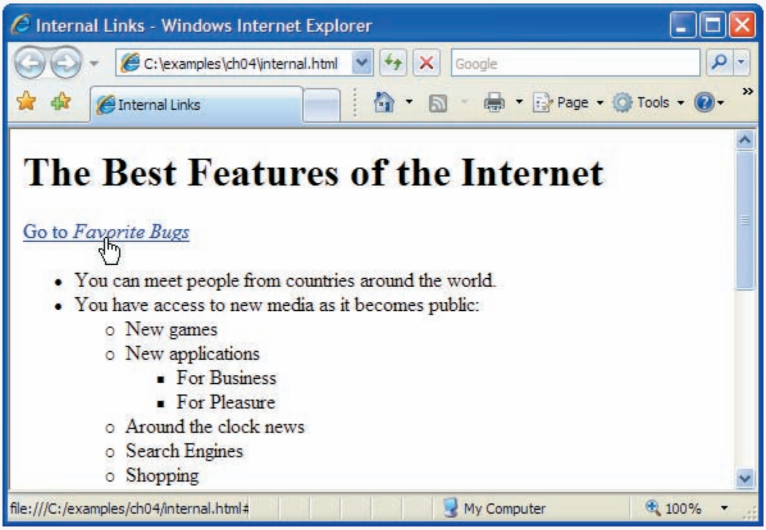
```
1   <?xml version = "1.0" encoding = "utf-8"?>
2   <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5   <!-- Fig. 4.14: internal.html -->
6   <!-- Internal hyperlinks to make pages more navigable. -->
7   <html xmlns = "http://www.w3.org/1999/xhtml">
```

Fig. 4.14 | Internal hyperlinks to make pages more navigable. (Part 1 of 3.)

```
   <head>
      <title>Internal Links</title>
   </head>

   <body>
      <!-- id attribute creates an internal hyperlink destination -->
      <h1 id = "features">The Best Features of the Internet</h1>

      <!-- an internal link's address is "#id" -->
      <p><a href = "#bugs">Go to <em>Favorite Bugs</em></a></p>

      <ul>
         <li>You can meet people from countries
            around the world.</li>
         <li>You have access to new media as it becomes public:
            <ul>
               <li>New games</li>
               <li>New applications
                  <ul>
                     <li>For Business</li>
                     <li>For Pleasure</li>
                  </ul>
               </li>

               <li>Around the clock news</li>
               <li>Search Engines</li>
               <li>Shopping</li>
               <li>Programming
                  <ul>
                     <li>XHTML</li>
                     <li>Java</li>
                     <li>Dynamic HTML</li>
                     <li>Scripts</li>
                     <li>New languages</li>
                  </ul>
               </li>
            </ul>
         </li>

         <li>Links</li>
         <li>Keeping in touch with old friends</li>
         <li>It is the technology of the future!</li>
      </ul>

      <!-- id attribute creates an internal hyperlink destination -->
      <h1 id = "bugs">My 3 Favorite Bugs</h1>
      <p>
         <!-- internal hyperlink to features -->
         <a href = "#features">Go to <em>Favorite Features</em></a>
      </p>
      <ol>
         <li>Fire Fly</li>
         <li>Gal Ant</li>
```

Fig. 4.14 | Internal hyperlinks to make pages more navigable. (Part 2 of 3.)

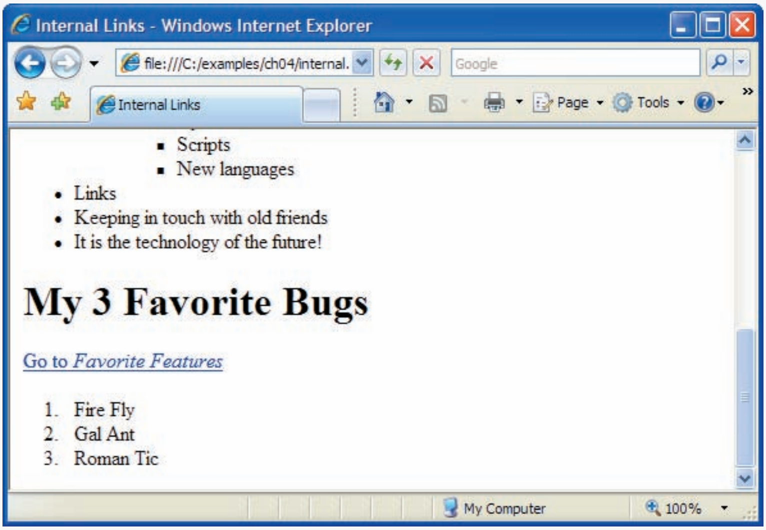
```
         <li>Roman Tic</li>
      </ol>
   </body>
</html>
```


Internal Links - Windows Internet Explorer
C:\examples\ch04\internal.html
Google
Internal Links
Page
Tools
file:///C:/examples/ch04/internal.html#
My Computer
100%

The Best Features of the Internet

Go to *Favorite Bugs*

- You can meet people from countries around the world.
- You have access to new media as it becomes public:
  - New games
  - New applications
    - For Business
    - For Pleasure
  - Around the clock news
  - Search Engines
  - Shopping


Internal Links - Windows Internet Explorer
file:///C:/examples/ch04/internal.
Google
Internal Links
Page
Tools
My Computer
100%

    - Scripts
    - New languages
- Links
- Keeping in touch with old friends
- It is the technology of the future!

My 3 Favorite Bugs

Go to *Favorite Features*

1. Fire Fly
2. Gal Ant
3. Roman Tic

Fig. 4.14 | Internal hyperlinks to make pages more navigable. (Part 3 of 3.)

Look-and-Feel Observation 4.4

Internal hyperlinks are useful in XHTML documents that contain large amounts of information. Internal links to different parts of the page make it easier for users to navigate the page—they do not have to scroll to find the section they want.

Although not demonstrated in this example, a hyperlink can specify an internal link in another document by specifying the document name followed by a pound sign and the id value, as in:

href = "*filename*.html#*id*"

For example, to link to a tag with the id attribute called booklist in books.html, href is assigned "books.html#booklist". You can also send the browser to an internal link on another website by appending the pound sign and id value of an element to any URL, as in:

href = "*URL*/filename.html#*id*"

4.13 meta Elements

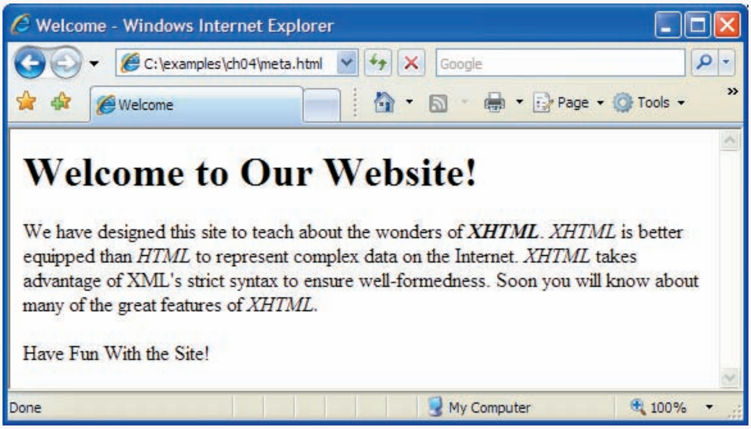
Search engines help people find websites. They usually catalog sites by following links from page to page (often known as spidering or crawling) and saving identification and classification information for each page. One way that search engines catalog pages is by reading the content in each page's **meta** elements, which specify information about a document.

Two important attributes of the meta element are **name**, which identifies the type of meta element, and **content**, which provides the information search engines use to catalog pages. Figure 4.15 introduces the meta element.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 4.15: meta.html -->
<!-- meta elements provide keywords and a description of a page. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Welcome</title>

      <!-- <meta> tags provide search engines with -->
      <!-- information used to catalog a site -->
      <meta name = "keywords" content = "web page, design,
         XHTML, tutorial, personal, help, index, form,
         contact, feedback, list, links, deitel" />
      <meta name = "description" content = "This website will
         help you learn the basics of XHTML and web page design
         through the use of interactive examples and
         instruction." />
   </head>
   <body>
      <h1>Welcome to Our Website!</h1>

      <p>We have designed this site to teach about the wonders
      of <strong><em>XHTML</em></strong>. <em>XHTML</em> is
      better equipped than <em>HTML</em> to represent complex
      data on the Internet. <em>XHTML</em> takes advantage of
      XML’s strict syntax to ensure well-formedness. Soon you
```

Fig. 4.15 | meta elements provide keywords and a description of a page. (Part 1 of 2.)

```
29           will know about many of the great features of
30           <em>XHTML.</em></p>
31
32           <p>Have Fun With the Site!</p>
33        </body>
34     </html>
```


Welcome - Windows Internet Explorer
C:\examples\ch04\meta.html
Google
Welcome
Page
Tools
Done
My Computer
100%

Welcome to Our Website!

We have designed this site to teach about the wonders of ***XHTML***. *XHTML* is better equipped than *HTML* to represent complex data on the Internet. *XHTML* takes advantage of XML's strict syntax to ensure well-formedness. Soon you will know about many of the great features of *XHTML*.

Have Fun With the Site!

Fig. 4.15 | meta elements provide keywords and a description of a page. (Part 2 of 2.)

Lines 13–15 demonstrate a "keywords" meta element. The content attribute of such a meta element provides search engines with a list of words that describe a page. These words are compared with words in search requests. Thus, including meta elements and their content information can draw more viewers to your site.

Lines 16–19 demonstrate a "description" meta element. The content attribute of such a meta element provides a three- to four-line description of a site, written in sentence form. Search engines also use this description to catalog your site and sometimes display this information as part of the search results. Note that this use of the meta element is one of many methods of search engine optimization (SEO). For more information on SEO, visit Deitel's SEO Resource Center at www.deitel.com/searchengineoptimization.

Software Engineering Observation 4.1

meta elements are not visible to users and must be placed inside the head section of your XHTML document. If meta elements are not placed in this section, they will not be read by search engines.

4.14 Wrap-Up

This chapter introduced XHTML and explored many of its features. We discussed the basics of XHTML as well as marking up information in tables, creating forms for gathering user input, linking to sections in the same document and using <meta> tags. In Chapter 5, we build on the XHTML introduced in this chapter by discussing how to make web pages more visually appealing with Cascading Style Sheets.

4.15 Web Resources

`www.deitel.com/xhtml`

Visit our online XHTML Resource Center for links to some of the best XHTML information on the web. There you'll find categorized links to introductions, tutorials, books, blogs, forums, sample chapters, and more. Also check out links about XHTML 2, the upcoming version of the XHTML standard.

Summary

Section 4.1 Introduction

- XHTML (Extensible HyperText Markup Language) is a markup language for creating web pages.
- XHTML is based on HTML (HyperText Markup Language)—a legacy technology of the World Wide Web Consortium (W3C).
- XHTML 1.0 allows only a document's content and structure to appear in a valid XHTML document, and not its formatting.
- Formatting is specified with Cascading Style Sheets

Section 4.2 Editing XHTML

- A machine that runs a specialized piece of software called a web server stores XHTML documents.

Section 4.3 First XHTML Example

- In XHTML, text is marked up with elements delimited by tags that are names contained in pairs of angle brackets. Some elements may contain attributes that provide additional information about the element.
- Every XHTML document contains a start `<html>` tag and an end `</html>` tag.
- Comments in XHTML always begin with `<!--` and end with `-->`. The browser ignores all text inside a comment.
- Every XHTML document contains a `head` element, which generally contains information, such as a title, and a `body` element, which contains the page content. Information in the `head` element generally is not rendered in the display window but may be made available to the user through other means.
- The `title` element names a web page. The title usually appears in the colored bar (called the title bar) at the top of the browser window and also appears as the text identifying a page when users add your page to their list of **Favorites** or **Bookmarks**.
- The `body` of an XHTML document is the area in which the document's content is placed. The content may include text and tags.
- All text placed between the `<p>` and `</p>` tags forms one paragraph.

Section 4.4 W3C XHTML Validation Service

- XHTML documents that are syntactically correct are guaranteed to render properly. XHTML documents that contain syntax errors may not display properly.
- Validation services (e.g., `validator.w3.org`) ensure that an XHTML document is syntactically correct.

Section 4.5 Headings

- XHTML provides six headings (h1 through h6) for specifying the relative importance of information. Heading element h1 is considered the most significant heading and is rendered in a larger font than the other five headings. Each successive heading element (i.e., h2, h3, etc.) is rendered in a progressively smaller font.

Section 4.6 Linking

- Web browsers typically underline text hyperlinks and color them blue by default.
- The strong element typically causes the browser to render text in a bold font.
- Users can insert links with the a (anchor) element. The most important attribute for the a element is href, which specifies the resource (e.g., page, file, e-mail address) being linked.
- Anchors can link to an e-mail address using a mailto: URL. When someone clicks this type of anchored link, most browsers launch the default e-mail program (e.g., Outlook Express) to initiate an e-mail message addressed to the linked address.

Section 4.7 Images

- The img element's src attribute specifies an image's location. In a valid XHTML document every img element must have an alt attribute, which contains text that is displayed if the client cannot render the image.
- The alt attribute makes web pages more accessible to users with disabilities, especially those with vision impairments.
- Some XHTML elements are empty elements that contain only attributes and do not mark up text. Empty elements (e.g., img) must be terminated, either by using the forward slash character (/) or by explicitly writing an end tag.

Section 4.8 Special Characters and Horizontal Rules

- XHTML provides special characters or entity references (in the form &*code*;) for representing characters that cannot be rendered otherwise.
- Special character codes can be either word abbreviations or numbers, decimal or hexadecimal.
- Most browsers render a horizontal rule, indicated by the <hr /> tag, as a horizontal line. The hr element also inserts a line break above and below the horizontal line.

Section 4.9 Lists

- The unordered list element ul creates a list in which each item in the list begins with a bullet symbol (called a disc). Each entry in an unordered list is an li (list item) element. Most web browsers render these elements with a line break and a bullet symbol at the beginning of the line.
- The ordered list element ol creates a list in which each item begins with a number.
- Lists may be nested to represent hierarchical data relationships.

Section 4.10 Tables

- XHTML tables are used to mark up tabular data. The table element defines an XHTML table.
- Element summary summarizes the table's contents and is used by speech devices to make the table more accessible to users with visual impairments.
- Element caption describe's the table's content. The text inside the <caption> tag is rendered above the table in most browsers.
- A table can be split into three distinct sections: head (thead), body (tbody) and foot (tfoot). The head section contains such information as table titles and column headers. The table body contains the primary table data. The table foot contains such information as footnotes.

- Element `tr`, or table row, defines individual table rows. Element `th` defines a header cell. Other data in a row is defined with `td`, or table data, elements.
- You can merge data cells with the `rowspan` and `colspan` attributes. The values assigned to these attributes specify the number of rows or columns occupied by the cell. These attributes can be placed inside any data cell or table header cell.

Section 4.11 Forms

- XHTML provides forms for collecting information from users. Forms contain visual components, such as buttons, that users interact with. Forms may also contain nonvisual components, called hidden inputs, which are used to store any data that needs to be sent to the server, but is not entered by the user.
- A form begins with the `form` element. Attribute `method` specifies how the form's data is sent to the web server.
- The `action` attribute of the `form` element specifies the script to which the `form` data will be sent.
- The `"text"` input inserts a text box into the form. Text boxes allow the user to input data.
- The `input` element's `size` attribute specifies the number of characters visible in the `input` element. Optional attribute `maxlength` limits the number of characters input into a text box.
- The `"submit"` input submits the data entered in the form to the web server for processing. Most web browsers create a button that submits the form data when clicked. The `"reset"` input allows a user to reset all `form` elements to their default values.
- The `textarea` element inserts a multiline text box, called a text area, into a form. The number of rows in the text area is specified with the `rows` attribute, and the number of columns (i.e., characters per line) is specified with the `cols` attribute.
- The `"password"` input inserts a password box into a form. A password box allows users to enter sensitive information, such as credit card numbers and passwords, by "masking" the information input with another character. Asterisks are usually the masking character used for password boxes. The actual value input is sent to the web server, not the asterisks that mask the input.
- The checkbox input allows the user to make a selection. When the checkbox is selected, a check mark appears in the checkbox. Otherwise, the checkbox is empty. Checkboxes can be used individually and in groups. Checkboxes that are part of the same group have the same `name`.
- The `br` element causes most browsers to render a line break. Any markup or text following a `br` element is rendered on the next line.
- A radio button is similar in function and use to a checkbox, except that only one radio button in a group can be selected at any time. All radio buttons in a group have the same `name` attribute but different `value` attributes.
- The `select` input provides a drop-down list of items. The `name` attribute identifies the drop-down list. The `option` element adds items to the drop-down list.

Section 4.12 Internal Linking

- The `a` tag can be used to link to another section of the same document by specifying the element's `id` as the link's `href`.
- To link internally to an element with its id attribute set, use the syntax `#`*id*.

Section 4.13 `meta` Elements

- One way that search engines catalog pages is by reading the `meta` element's contents. Two important attributes of the `meta` element are `name`, which identifies the type of `meta` element, and `content`, which provides information a search engine uses to catalog a page.

- The content attribute of a keywords meta element provides search engines with a list of words that describe a page. These words are compared with words in search requests.
- The content attribute of a description meta element provides a three- to four-line description of a site, written in sentence form. Search engines also use this description to catalog your site and sometimes display this information as part of the search results.

Terminology

<!--...--> (XHTML comment)
a element (<a>...</a>)
accessible web pages
action attribute (form)
alt attribute (img)
&amp; (& special character)
anchor
angle brackets (< >)
attribute
body element
border attribute (table)
br element (line break)
browser request
caption element
Cascading Syle Sheets (CSS)
character entity reference
checkbox
checked attribute (input)
cols attribute (textarea)
colspan attribute (th, td)
comment in XHTML
&copy; (© special character)
data cells
database
debugging
del element
element
em element
e-mail anchor
empty element
end tag
Extensible HyperText Markup Language (XHTML)
form
form element
form handler
get request type
head element
header cell
heading
h1 through h6 (heading elements)
height attribute (img)
hexadecimal code
hidden input element
hr element (horizontal rule)
href attribute (a)
.htm (XHTML filename extension)
<html> tag
.html (XHTML filename extension)
hyperlink
img element
input element
internal linking
level of nesting
li element (list item)
line break
link
linked document
list item
&lt; (< special character)
mailto: URL
markup language
maxlength attribute (input)
meta element
method attribute (form)
name attribute
nested list
nested tag
numeric character reference
ol element (ordered list)
option element
p element (paragraph)
password box
pixel
post request type
presentation of a document
protocol
radio input
reset input
resources
rows attribute (textarea)
rowspan attribute (th, tr)
script
select element
selected attribute (option)
size attribute (input)

source code
special character
speech synthesizer
`src` attribute (`img`)
start tag
`strong` element
`sub` element
`submit` input
subscript
superscript
`table` element
tag
`tbody` element
`td` element
text editor
text-based browser
textarea
`textarea` element
`tfoot` element (table foot)
`thead` element (table head)
`title` element
`tr` element (table row)
`type` attribute (`input`)
`ul` element (unordered list)
valid document
validation service
`value` attribute (`input`)
value of an attribute
web page
web server
`width` attribute (`img`)
World Wide Web (WWW)
World Wide Web Consortium (W3C)
XHTML (Extensible HyperText Markup Language)
XHTML comment
XHTML document
XHTML form
XHTML markup
XHTML tag
XML declaration
`xmlns` attribute

Self-Review Exercises

4.1 State whether each of the following is *true* or *false*. If *false*, explain why.

a) An ordered list cannot be nested inside an unordered list.
b) XHTML is an acronym for XML HTML.
c) Element `br` represents a line break.
d) Hyperlinks are denoted by `link` elements.
e) The width of all data cells in a table must be the same.
f) You are limited to a maximum of 100 internal links per page.

4.2 Fill in the blanks in each of the following:

a) The __________ element inserts a horizontal rule.
b) A superscript is marked up using element __________ and a subscript is marked up using element __________.
c) The least important heading element is __________ and the most important heading element is __________.
d) Element __________ marks up an unordered list.
e) Element __________ marks up a paragraph.
f) The __________ attribute in an `input` element inserts a button that, when clicked, clears the contents of the form.
g) The __________ element marks up a table row.
h) __________ are usually used as masking characters in a password box.

Answers to Self-Review Exercises

4.1 a) False. An ordered list can be nested inside an unordered list and vice versa. b) False. XHTML is an acronym for Extensible HyperText Markup Language. c) True. d) False. Hyperlinks are denoted by a elements. e) False. You can specify the width of any column, either in pixels or as a percentage of the table width. f) False. You can have an unlimited number of internal links.

4.2 a) `hr`. b) `sup`, `sub`. c) `h6`, `h1`. d) `ul`. e) `p`. f) `type = "reset"`. g) `tr`. h) Asterisks.

Exercises

4.3 Use XHTML to create a document that contains the following text:

```
Internet and World Wide Web How to Program: Fourth Edition
Welcome to the world of Internet programming. We have provided
topical coverage for many Internet-related topics.
```

Use `h1` for the title (the first line of text), `p` for text (the second and third lines of text) and `sub` for each word that begins with a capital letter (except the title). Insert a horizontal rule between the `h1` element and the `p` element. Open your new document in a web browser to view the marked-up document.

4.4 Why is the following markup invalid?

```
<p>Here is some text...
<hr />
<p>And some more text...</p>
```

4.5 Why is the following markup invalid?

```
<p>Here is some text...<br>
And some more text...</p>
```

4.6 An image named `deitel.gif` is 200 pixels wide and 150 pixels high. Write an XHTML statement using the `width` and `height` attributes of the `img` element to perform each of the following transformations:

a) Increase the size of the image by 100 percent.
b) Increase the size of the image by 50 percent.
c) Change the width-to-height ratio to 2:1, keeping the `width` attained in part (a).

4.7 Create a link to each of the following:

a) The file `index.html`, located in the `files` directory.
b) The file `index.html`, located in the `text` subdirectory of the `files` directory.
c) The file `index.html`, located in the `other` directory in your parent directory. [*Hint:* `..` signifies parent directory.]
d) The President's e-mail address (`president@whitehouse.gov`).
e) The file named `README` in the `pub` directory of `ftp.cdrom.com`. [*Hint:* Use `ftp://`.]

4.8 Create an XHTML document containing three ordered lists: ice cream, soft serve and frozen yogurt. Each ordered list should contain a nested, unordered list of your favorite flavors. Provide a minimum of three flavors in each unordered list.

4.9 Create an XHTML document that uses an image as an e-mail link. Use attribute `alt` to provide a description of the image and link.

4.10 Create an XHTML document that contains links to your favorite websites. Your page should contain the heading "My Favorite Web Sites."

4.11 Create an XHTML document that contains an unordered list with links to all the examples presented in this chapter. [*Hint:* Place all the chapter examples in one directory.]

4.12 Identify each of the following as either an element or an attribute:

a) `html`
b) `width`
c) `href`
d) `br`
e) `h3`
f) `a`
g) `src`

4.13 State which of the following statements are *true* and which are *false*. If *false*, explain why.

a) A valid XHTML document can contain uppercase letters in element names.
b) Tags need not be closed in a valid XHTML document.
c) XHTML documents can have the file extension `.htm`.
d) Valid XHTML documents can contain tags that overlap.
e) `&less;` is the character entity reference for the less-than (<) character.
f) In a valid XHTML document, `<li>` can be nested inside either `<ol>` or `<ul>` tags.

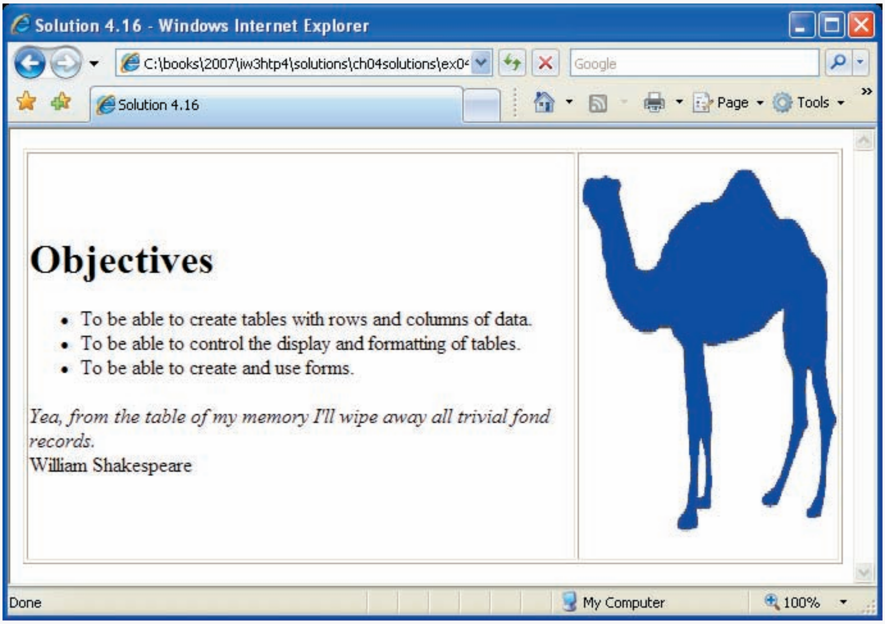
4.14 Fill in the blanks in each of the following:

a) XHTML comments begin with `<!--` and end with __________.
b) In XHTML, attribute values must be enclosed in __________.
c) __________ is the character entity reference for an ampersand.
d) Element __________ can be used to bold text.

4.15 Categorize each of the following as an element or an attribute:

a) `width`
b) `td`
c) `th`
d) `name`
e) `select`
f) `type`

4.16 Create the XHTML markup that produces the table shown in Fig. 4.16. Use `<em>` and `<strong>` tags as necessary. The image (`camel.gif`) is included in the Chapter 4 examples directory that can be downloaded from `http://www.deitel.com/books/iw3htp4/`.



Solution 4.16 - Windows Internet Explorer
C:\books\2007\iw3htp4\solutions\ch04solutions\ex0
Google
Solution 4.16
Page
Tools
Done
My Computer
100%

Objectives

- To be able to create tables with rows and columns of data.
- To be able to control the display and formatting of tables.
- To be able to create and use forms.

Yea, from the table of my memory I'll wipe away all trivial fond records.
William Shakespeare

Fig. 4.16 | XHTML table for Exercise 4.16.

4.17 Write an XHTML document that produces the table shown in Fig. 4.17.

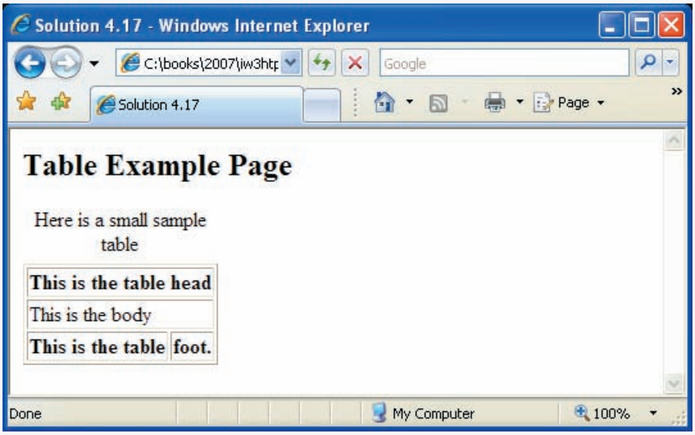


Table Example Page

Here is a small sample table

This is the table head	
This is the body	
This is the table	**foot.**

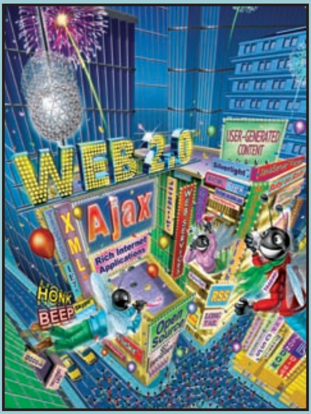
Fig. 4.17 | XHTML table for Exercise 4.17.

4.18 A local university has asked you to create an XHTML document that allows prospective students to provide feedback about their campus visit. Your XHTML document should contain a form with text boxes for a name, address and e-mail. Provide checkboxes that allow prospective students to indicate what they liked most about the campus. The checkboxes should include: students, location, campus, atmosphere, dorm rooms and sports. Also, provide radio buttons that ask the prospective students how they became interested in the university. Options should include: friends, television, Internet and other. In addition, provide a text area for additional comments, a submit button and a reset button.

4.19 Create an XHTML document titled "How to Get Good Grades." Use `<meta>` tags to include a series of keywords that describe your document.

5

Cascading Style Sheets™ (CSS)



Fashions fade, style is eternal.
—Yves Saint Laurent

A style does not go out of style as long as it adapts itself to its period. When there is an incompatibility between the style and a certain state of mind, it is never the style that triumphs.
—Coco Chanel

How liberating to work in the margins, outside a central perception.
—Don DeLillo

I've gradually risen from lower-class background to lower-class foreground.
—Marvin Cohen

OBJECTIVES

In this chapter you will learn:

- To control the appearance of a website by creating style sheets.
- To use a style sheet to give all the pages of a website the same look and feel.
- To use the `class` attribute to apply styles.
- To specify the precise font, size, color and other properties of displayed text.
- To specify element backgrounds and colors.
- To understand the box model and how to control margins, borders and padding.
- To use style sheets to separate presentation from content.

Outline

5.1 Introduction

In Chapter 4, we introduced the Extensible HyperText Markup Language (XHTML) for marking up information to be rendered in a browser. In this chapter, we shift our focus to formatting and presenting information. To do this, we use a W3C technology called **Cascading Style Sheets™** (**CSS**) that allows document authors to specify the presentation of elements on a web page (e.g., fonts, spacing, colors) separately from the structure of the document (section headers, body text, links, etc.). This **separation of structure from presentation** simplifies maintaining and modifying a web page.

XHTML was designed to specify the content and structure of a document. Though it has some attributes that control presentation, it is better not to mix presentation with content. If a website's presentation is determined entirely by a style sheet, a web designer can simply swap in a new style sheet to completely change the appearance of the site. CSS provides a way to apply style outside of XHTML, allowing the XHTML to dictate the content while the CSS dictates how it's presented.

As with XHTML, the W3C provides a CSS code validator located at `jigsaw.w3.org/css-validator/`. It is a good idea to validate all CSS code with this tool to make sure that your code is correct and works on as many browsers as possible.

CSS is a large topic. As such, we can introduce only the basic knowledge of CSS that you'll need to understand the examples and exercises in the rest of the book. For more CSS references and resources, check out our CSS Resource Center at `www.deitel.com/css21`.

The W3C's CSS specification is currently in its second major version, with a third in development. The current versions of most major browsers support much of the functionality in CSS 2. This allows programmers to make full use of its features. In this chapter, we introduce CSS, demonstrate some of the features introduced in CSS 2 and discuss some of the upcoming CSS 3 features. As you read this book, open each XHTML document in your web browser so you can view and interact with it in a web browser, as it was originally intended.

Remember that the examples in this book have been tested in Internet Explorer 7 and Firefox 2. The latest versions of many other browsers (e.g., Safari, Opera, Konqueror) should render this chapter's examples properly, but we have not tested them. Some examples in this chapter *will not work* in older browsers, such as Internet Explorer 6 and earlier. Make sure you have either Internet Explorer 7 (Windows only) or Firefox 2 (available for all major platforms) installed before running the examples in this chapter.

5.2 Inline Styles

You can declare document styles in several ways. This section presents **inline styles** that declare an individual element's format using the XHTML attribute **`style`**. Inline styles override any other styles applied using the techniques we discuss later in the chapter. Figure 5.1 applies inline styles to `p` elements to alter their font size and color.

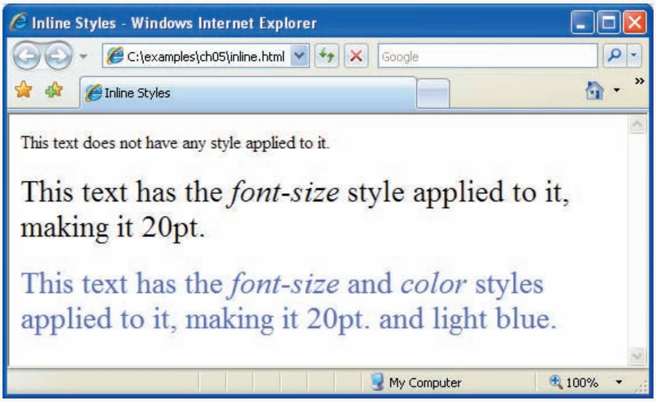
Good Programming Practice 5.1

Inline styles do not truly separate presentation from content. To apply similar styles to multiple elements, use embedded style sheets or external style sheets, introduced later in this chapter.

The first inline style declaration appears in line 17. Attribute `style` specifies an element's style. Each **CSS property** (**`font-size`** in this case) is followed by a colon and a value. In line 17, we declare this particular `p` element to use 20-point font size.

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 5.1: inline.html -->
 6  <!-- Using inline styles -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Inline Styles</title>
10     </head>
11     <body>
12        <p>This text does not have any style applied to it.</p>
13
14        <!-- The style attribute allows you to declare -->
15        <!-- inline styles. Separate multiple style properties -->
16        <!-- with a semicolon. -->
17        <p style = "font-size: 20pt">This text has the
18           <em>font-size</em> style applied to it, making it 20pt.
19        </p>
20
21        <p style = "font-size: 20pt; color: #6666ff">
22        This text has the <em>font-size</em> and
23        <em>color</em> styles applied to it, making it
24        20pt. and light blue.</p>
25     </body>
26  </html>
```

Fig. 5.1 | Using inline styles. (Part 1 of 2.)


Inline Styles - Windows Internet Explorer
C:\examples\ch05\inline.html
Google
Inline Styles
My Computer
100%

This text does not have any style applied to it.

This text has the *font-size* style applied to it, making it 20pt.

This text has the *font-size* and *color* styles applied to it, making it 20pt. and light blue.

Fig. 5.1 | Using inline styles. (Part 2 of 2.)

Line 21 specifies the two properties, `font-size` and **`color`**, separated by a semicolon. In this line, we set the given paragraph's `color` to light blue, using the hexadecimal code `#6666ff`. Color names may be used in place of hexadecimal codes. We provide a list of hexadecimal color codes and color names in Appendix B, XHTML Colors.

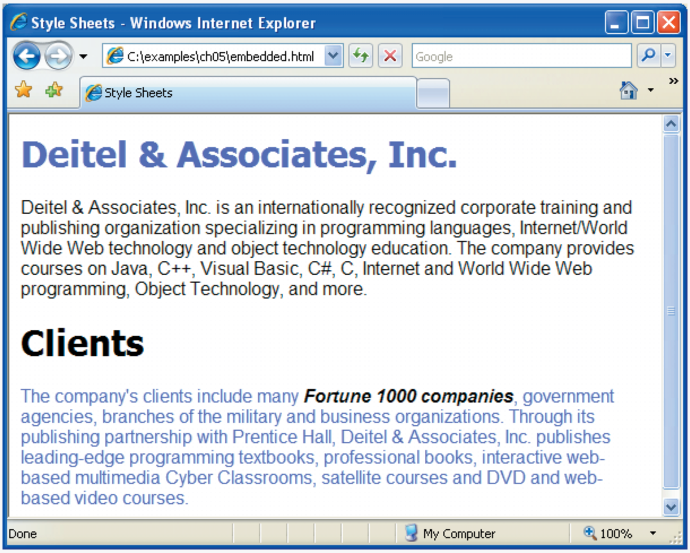
5.3 Embedded Style Sheets

A second technique for using style sheets is **embedded style sheets**. Embedded style sheets enable a you to embed an entire CSS document in an XHTML document's `head` section. To achieve this separation between the CSS code and the XHTML that it styles, we will use **CSS selectors**. Figure 5.2 creates an embedded style sheet containing four styles.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 5.2: embedded.html -->
<!-- Embedded style sheets. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Style Sheets</title>

      <!-- this begins the style sheet section -->
      <style type = "text/css">
         em       { font-weight: bold;
                    color: black }
         h1       { font-family: tahoma, helvetica, sans-serif }
         p        { font-size: 12pt;
                    font-family: arial, sans-serif }
         .special { color: #6666ff }
      </style>
   </head>
```

Fig. 5.2 | Embedded style sheets. (Part 1 of 2.)

```
21      <body>
22         <!-- this class attribute applies the .special style -->
23         <h1 class = "special">Deitel &amp; Associates, Inc.</h1>
24
25         <p>Deitel &amp; Associates, Inc. is an internationally
26         recognized corporate training and publishing organization
27         specializing in programming languages, Internet/World
28         Wide Web technology and object technology education.
29         The company provides courses on Java, C++, Visual Basic,
30         C#, C, Internet and World Wide Web programming, Object
31         Technology, and more.</p>
32
33         <h1>Clients</h1>
34         <p class = "special"> The company's clients include many
35         <em>Fortune 1000 companies</em>, government agencies,
36         branches of the military and business organizations.
37         Through its publishing partnership with Prentice Hall,
38         Deitel &amp; Associates, Inc. publishes leading-edge
39         programming textbooks, professional books, interactive
40         web-based multimedia Cyber Classrooms, satellite
41         courses and World Wide Web courses.</p>
42      </body>
43   </html>
```


Style Sheets - Windows Internet Explorer
C:\examples\ch05\embedded.html
Google
Style Sheets
Done
My Computer
100%

Deitel & Associates, Inc.

Deitel & Associates, Inc. is an internationally recognized corporate training and publishing organization specializing in programming languages, Internet/World Wide Web technology and object technology education. The company provides courses on Java, C++, Visual Basic, C#, C, Internet and World Wide Web programming, Object Technology, and more.

Clients

The company's clients include many ***Fortune 1000 companies***, government agencies, branches of the military and business organizations. Through its publishing partnership with Prentice Hall, Deitel & Associates, Inc. publishes leading-edge programming textbooks, professional books, interactive web-based multimedia Cyber Classrooms, satellite courses and DVD and web-based video courses.

Fig. 5.2 | Embedded style sheets. (Part 2 of 2.)

The style element (lines 12–19) defines the embedded style sheet. Styles placed in the head apply to matching elements wherever they appear in the entire document. The style element's type attribute specifies the **Multipurpose Internet Mail Extensions (MIME) type** that describes a file's content. CSS documents use the MIME type text/css. Other MIME types include image/gif (for GIF images), text/javascript (for the JavaScript scripting language, which we discuss in Chapters 6–11), and more.

The body of the style sheet (lines 13–18) declares the **CSS rules** for the style sheet. A CSS selector determines which elements will be styled according to a rule. Our first rule begins with the selector `em` (line 13) to select all `em` elements in the document. The **`font-weight`** property in line 13 specifies the "boldness" of text. Possible values are `bold`, `normal` (the default), `bolder` (bolder than `bold` text) and `lighter` (lighter than `normal` text). Boldness also can be specified with multiples of 100, from 100 to 900 (e.g., `100`, `200`, `…`, `900`). Text specified as `normal` is equivalent to `400`, and `bold` text is equivalent to `700`. However, many systems do not have fonts that can scale with this level of precision, so using the values from `100` to `900` might not display the desired effect.

In this example, all `em` elements will be displayed in a bold font. We also apply styles to all `h1` (line 15) and `p` (lines 16–17) elements. The body of each rule is enclosed in curly braces (`{` and `}`).

Line 18 uses a new kind of selector to declare a **style class** named `special`. Style classes define styles that can be applied to any element. In this example, we declare class `special`, which sets `color` to `blue`. We can apply this style to any element type, whereas the other rules in this style sheet apply only to specific element types defined in the style sheet (i.e., `em`, `h1` or `p`). Style-class declarations are preceded by a period. We will discuss how to apply a style class momentarily.

CSS rules in embedded style sheets use the same syntax as inline styles; the property name is followed by a colon (`:`) and the value of the property. Multiple properties are separated by semicolons (`;`). In the rule for `em` elements, the `color` property specifies the color of the text, and property `font-weight` makes the font bold.

The **`font-family`** property (line 15) specifies the name of the font to use. Not all users have the same fonts installed on their computers, so CSS allows you to specify a comma-separated list of fonts to use for a particular style. The browser attempts to use the fonts in the order they appear in the list. It's advisable to end a font list with a **generic font family name** in case the other fonts are not installed on the user's computer. In this example, if the `tahoma` font is not found on the system, the browser will look for the `helvetica` font. If neither is found, the browser will display its default `sans-serif` font. Other generic font families include `serif` (e.g., `times new roman`, `Georgia`), `cursive` (e.g., `script`), `fantasy` (e.g., `critter`) and `monospace` (e.g., `courier`, `fixedsys`).

The **`font-size`** property (line 16) specifies a 12-point font. Other possible measurements in addition to `pt` (point) are introduced later in the chapter. Relative values— `xx-small`, `x-small`, `small`, `smaller`, `medium`, `large`, `larger`, `x-large` and `xx-large`—also can be used. Generally, relative values for `font-size` are preferred over point sizes because an author does not know the specific measurements of the display for each client. Relative `font-size` values permit more flexible viewing of web pages.

For example, a user may wish to view a web page on a handheld device with a small screen. Specifying an 18-point font size in a style sheet will prevent such a user from seeing more than one or two characters at a time. However, if a relative font size is specified, such as `large` or `larger`, the actual size is determined by the browser that displays the font. Using relative sizes also makes pages more accessible to users with disabilities. Users with impaired vision, for example, may configure their browser to use a larger default font, upon which all relative sizes are based. Text that the author specifies to be `smaller` than the main text still displays in a smaller size font, yet it is clearly visible to each user. Accessibility is an important consideration—in 1998, congress passed the Section 508 Amendment to the Rehabilitation Act of 1973, mandating that websites of government agencies are required to be accessible to disabled users.

Line 23 uses the XHTML attribute **`class`** in an `h1` element to apply a style class—in this case class `special` (declared with the `.special` selector in the style sheet on line 18). When the browser renders the `h1` element, note that the text appears on screen with the properties of both an `h1` element (`arial` or `sans-serif` font defined in line 17) and the `.special` style class applied (the color `#6666ff` defined in line 18). Also notice that the browser still applies its own default style to the `h1` element—the header is still displayed in a large font size. Similarly, all `em` elements will still be italicized by the browser, but they will also be bold as a result of our style rule.

The formatting for the `p` element and the `.special` class is applied to the text in lines 34–41. In many cases, the styles applied to an element (the parent or ancestor element) also apply to the element's nested elements (child or descendant elements). The `em` element nested in the `p` element in line 35 inherits the style from the `p` element (namely, the 12-point font size in line 16) but retains its italic style. In other words, styles defined for the paragraph and not defined for the em element is applied to the em element. Because multiple values of one property can be set or inherited on the same element, they must be reduced to one style per element before being rendered. We discuss the rules for resolving these conflicts in the next section.

5.4 Conflicting Styles

Styles may be defined by a user, an author or a user agent (e.g., a web browser). A user is a person viewing your web page, you are the author—the person who writes the document—and the user agent is the program used to render and display the document. Styles "cascade," or flow together, such that the ultimate appearance of elements on a page results from combining styles defined in several ways. Styles defined by the user take precedence over styles defined by the user agent, and styles defined by authors take precedence over styles defined by the user.

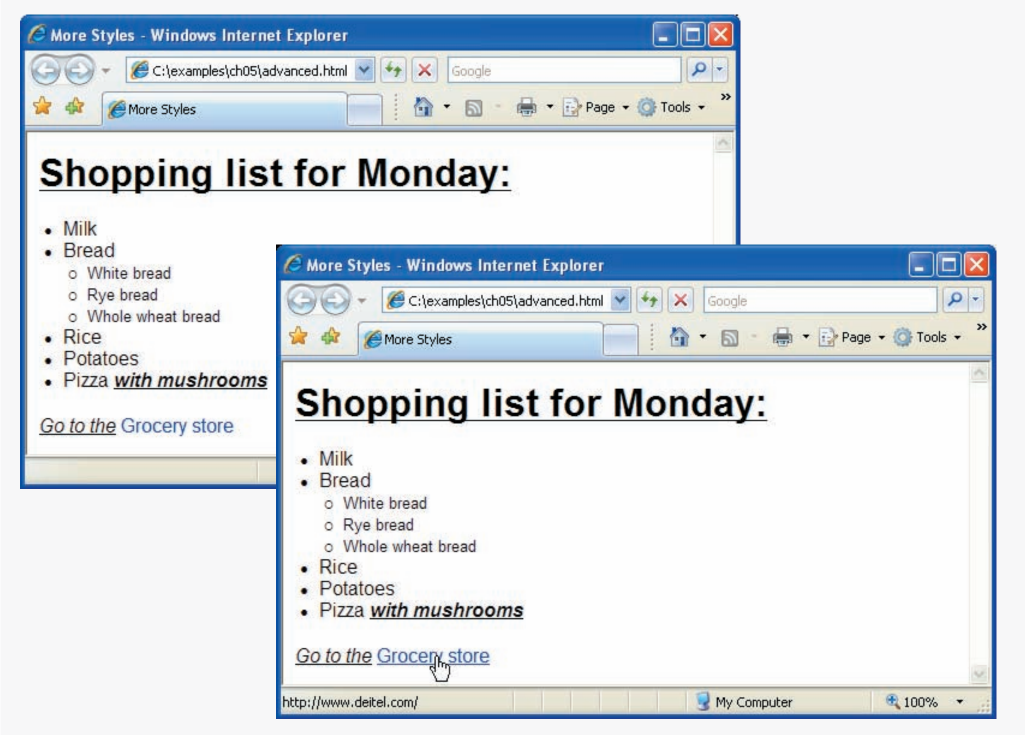
Most styles defined for parent elements are also inherited by child (nested) elements. While it makes sense to inherit most styles, such as font properties, there are certain properties that we don't want to be inherited. Consider for example the `background-image` property, which allows the programmer to set an image as the background of an element. If the `body` element is assigned a background image, we don't want the same image to be in the background of every element in the body of our page. Instead, the `background-image` property of all child elements retains its default value of `none`. In this section, we discuss the rules for resolving conflicts between styles defined for elements and styles inherited from parent and ancestor elements.

Figure 5.2 presented an example of inheritance in which a child `em` element inherited the `font-size` property from its parent `p` element. However, in Fig. 5.2, the child `em` element had a `color` property that conflicted with (i.e., had a different value than) the `color` property of its parent `p` element. Properties defined for child and descendant elements have a greater specificity than properties defined for parent and ancestor elements. Conflicts are resolved in favor of properties with a higher specificity. In other words, the styles explicitly defined for a child element are more specific than the styles defined for the child's parent element; therefore, the child's styles take precedence. Figure 5.3 illustrates examples of inheritance and specificity.

Line 12 applies property `text-decoration` to all `a` elements whose `class` attribute is set to `nodec`. The `text-decoration` property applies **decorations** to text in an element. By default, browsers underline the text of an `a` (anchor) element. Here, we set the `text-decoration` property to `none` to indicate that the browser should not underline hyperlinks. Other possible values for `text-decoration` include `overline`, `line-through`, `underline` and `blink`. [*Note:* `blink` is not supported by Internet Explorer.] The `.nodec` appended to `a` is a more specific class selector; this style will apply only to `a` (anchor) elements that specify `nodec` in their `class` attribute.

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 5.3: advanced.html -->
 6  <!-- Inheritance in style sheets. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>More Styles</title>
10        <style type = "text/css">
11           body     { font-family: arial, helvetica, sans-serif }
12           a.nodec  { text-decoration: none }
13           a:hover  { text-decoration: underline }
14           li em    { font-weight: bold }
15           h1, em   { text-decoration: underline }
16           ul       { margin-left: 20px }
17           ul ul    { font-size: .8em }
18        </style>
19     </head>
20     <body>
21        <h1>Shopping list for Monday:</h1>
22
23        <ul>
24           <li>Milk</li>
25           <li>Bread
26              <ul>
27                 <li>White bread</li>
28                 <li>Rye bread</li>
29                 <li>Whole wheat bread</li>
30              </ul>
31           </li>
32           <li>Rice</li>
33           <li>Potatoes</li>
34           <li>Pizza <em>with mushrooms</em></li>
35        </ul>
36
37        <p><em>Go to the</em>
38           <a class = "nodec" href = "http://www.deitel.com">
39              Grocery store</a>
40        </p>
41     </body>
42  </html>
```

Fig. 5.3 | Inheritance in style sheets. (Part 1 of 2.)


More Styles - Windows Internet Explorer
C:\examples\ch05\advanced.html
Google
More Styles
Page
Tools

Shopping list for Monday:

- Milk
- Bread
  - White bread
  - Rye bread
  - Whole wheat bread
- Rice
- Potatoes
- Pizza ***with mushrooms***

Go to the Grocery store

More Styles - Windows Internet Explorer
C:\examples\ch05\advanced.html
Google
More Styles
Page
Tools

Shopping list for Monday:

- Milk
- Bread
  - White bread
  - Rye bread
  - Whole wheat bread
- Rice
- Potatoes
- Pizza ***with mushrooms***

Go to the Grocery store

http://www.deitel.com/
My Computer
100%

Fig. 5.3 | Inheritance in style sheets. (Part 2 of 2.)

Portability Tip 5.1

To ensure that your style sheets work in various web browsers, test them on all the client web browsers that will render documents using your styles, as well as using the W3C CSS Validator.

Line 13 specifies a style for hover, which is a **pseudoclass**. Pseudoclasses give the author access to content not specifically declared in the document. The **hover pseudoclass** is activated dynamically when the user moves the mouse cursor over an element. Note that pseudoclasses are separated by a colon (with no surrounding spaces) from the name of the element to which they are applied.

Common Programming Error 5.1

Including a space before or after the colon separating a pseudoclass from the name of the element to which it is applied is an error that prevents the pseudoclass from being applied properly.

Line 14 causes all em elements that are children of li elements to be bold. In the screen output of Fig. 5.3, note that Go to the (contained in an em element in line 37) does not appear bold, because the em element is not in an li element. However, the em element containing with mushrooms (line 34) is nested in an li element; therefore, it is formatted in bold. The syntax for applying rules to multiple elements is similar. In line 15, we separate the selectors with a comma to apply an underline style rule to all h1 and all em elements.

Line 16 assigns a left margin of 20 pixels to all ul elements. We will discuss the margin properties in detail in Section 5.9. A pixel is a **relative-length measurement**—it varies in size, based on screen resolution. Other relative lengths include **em** (the *M*-height of the font, which is usually set to the height of an uppercase *M*), **ex** (the *x*-height of the font, which is usually set to the height of a lowercase *x*) and percentages (e.g., `font-size: 50%`). To set an element to display text at 150 percent of its default text size, the author could use the syntax

```
font-size: 1.5em
```

Alternatively, you could use

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

Other units of measurement available in CSS are **absolute-length measurements**—i.e., units that do not vary in size based on the system. These units are **in** (inches), **cm** (centimeters), **mm** (millimeters), **pt** (points; 1 `pt` = 1/72 `in`) and **pc** (picas; 1 `pc` = 12 `pt`). Line 17 specifies that all nested unordered lists (`ul` elements that are descendants of `ul` elements) are to have font size `.8em`. [*Note:* When setting a style property that takes a measurement (e.g. `font-size`, `margin-left`), no units are necessary if the value is zero.]

Good Programming Practice 5.2

Whenever possible, use relative-length measurements. If you use absolute-length measurements, your document may not be readable on some client browsers (e.g., wireless phones).

5.5 Linking External Style Sheets

Style sheets are a convenient way to create a document with a uniform theme. With **external style sheets** (i.e., separate documents that contain only CSS rules), you can provide a uniform look and feel to an entire website. Different pages on a site can all use the same style sheet. When changes to the styles are required, the author needs to modify only a single CSS file to make style changes across the entire website. Note that while embedded style sheets separate content from presentation, both are still contained in a single file, preventing a web designer and a content author from working in parallel. External style sheets solve this problem by separating the content and style into separate files.

Software Engineering Observation 5.1

Always use an external style sheet when developing a website with multiple pages. External style sheets separate content from presentation, allowing for more consistent look-and-feel, more efficient development, and better performance.

Figure 5.4 presents an external style sheet. Lines 1–2 are **CSS comments**. Like XHTML comments, CSS comments describe the content of a CSS document. Comments may be placed in any type of CSS code (i.e., inline styles, embedded style sheets and external style sheets) and always start with `/*` and end with `*/`. Text between these delimiters is ignored by the browser.

```
1   /* Fig. 5.4: styles.css */
2   /* External stylesheet */
3
```

Fig. 5.4 | External style sheet. (Part 1 of 2.)

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

a.nodec  { text-decoration: none }

a:hover  { text-decoration: underline }

li em    { font-weight: bold }

h1, em   { text-decoration: underline }

ul       { margin-left: 20px }

ul ul    { font-size: .8em; }
```

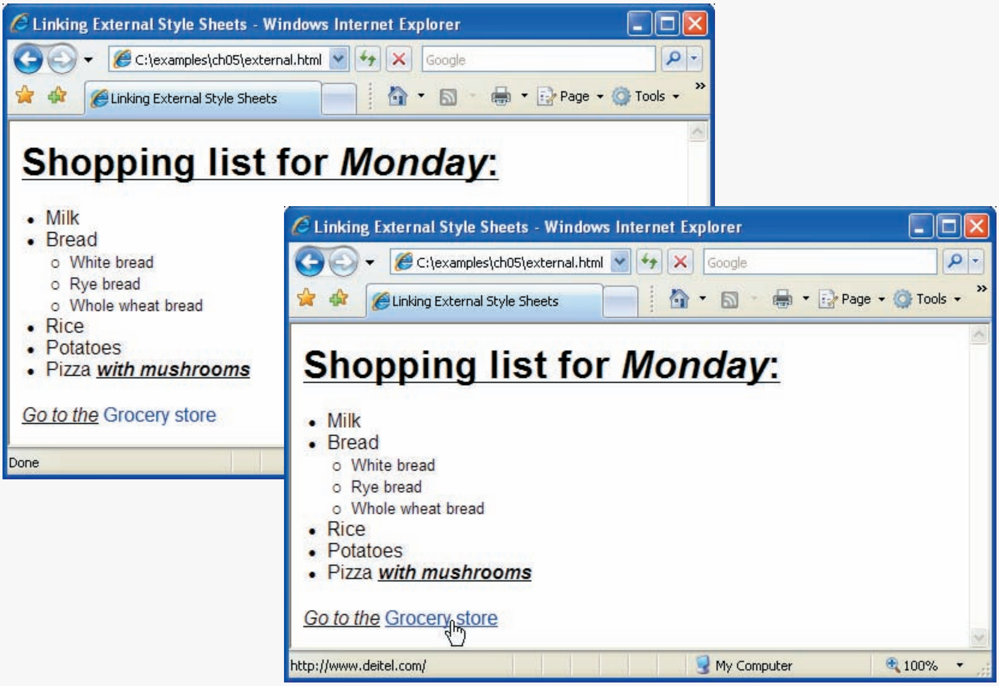
Fig. 5.4 | External style sheet. (Part 2 of 2.)

Figure 5.5 contains an XHTML document that references the external style sheet in Fig. 5.4. Lines 10–11 (Fig. 5.5) show a **`link`** element that uses the **`rel`** attribute to specify a **relationship** between the current document and another document. In this case, we declare the linked document to be a **`stylesheet`** for this document. The `type` attribute specifies the MIME type of the related document as `text/css`. The `href` attribute provides the URL for the document containing the style sheet. In this case, `styles.css` is in the same directory as `external.html`.

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 5.6: external.html -->
 6  <!-- Linking an external style sheet. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Linking External Style Sheets</title>
10        <link rel = "stylesheet" type = "text/css"
11           href = "styles.css" />
12     </head>
13     <body>
14        <h1>Shopping list for <em>Monday</em>:</h1>
15
16        <ul>
17           <li>Milk</li>
18           <li>Bread
19              <ul>
20                 <li>White bread</li>
21                 <li>Rye bread</li>
22                 <li>Whole wheat bread</li>
23              </ul>
24           </li>
25           <li>Rice</li>
26           <li>Potatoes</li>
```

Fig. 5.5 | Linking an external style sheet. (Part 1 of 2.)

```
            <li>Pizza <em>with mushrooms</em></li>
         </ul>

         <p><em>Go to the</em>
            <a class = "nodec" href = "http://www.deitel.com">
               Grocery store</a>
         </p>
      </body>
   </html>
```


Linking External Style Sheets - Windows Internet Explorer
C:\examples\ch05\external.html
Google
Linking External Style Sheets
Page
Tools
Done
http://www.deitel.com/
My Computer
100%

Shopping list for *Monday*:

- Milk
- Bread
  - White bread
  - Rye bread
  - Whole wheat bread
- Rice
- Potatoes
- Pizza *with mushrooms*

Go to the Grocery store

Shopping list for *Monday*:

- Milk
- Bread
  - White bread
  - Rye bread
  - Whole wheat bread
- Rice
- Potatoes
- Pizza *with mushrooms*

Go to the Grocery store

Fig. 5.5 | Linking an external style sheet. (Part 2 of 2.)

Software Engineering Observation 5.2

External style sheets are reusable. Creating them once and reusing them reduces programming effort.

Performance Tip 5.1

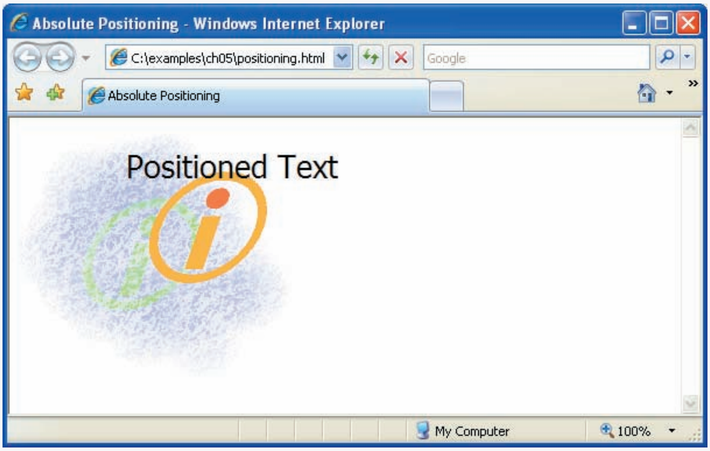
Reusing external style sheets reduces load time and bandwidth usage on a server, since the style sheet can be downloaded once, stored by the web browser, and applied to all pages on a website.

5.6 Positioning Elements

Before CSS, controlling the positioning of elements in an XHTML document was difficult—the browser determined positioning. CSS introduced the **`position`** property and a capability called absolute positioning, which gives authors greater control over how document elements are displayed. Figure 5.6 demonstrates absolute positioning.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 5.6: positioning.html -->
<!-- Absolute positioning of elements. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Absolute Positioning</title>
      <style type = "text/css">
         .bgimg   { position: absolute;
                    top: 0px;
                    left: 0px;
                    z-index: 1 }
         .fgimg   { position: absolute;
                    top: 25px;
                    left: 100px;
                    z-index: 2 }
         .text    { position: absolute;
                    top: 25px;
                    left: 100px;
                    z-index: 3;
                    font-size: 20pt;
                    font-family: tahoma, geneva, sans-serif }
      </style>
   </head>
   <body>
      <p><img src = "bgimg.gif" class = "bgimg"
         alt = "First positioned image" /></p>

      <p><img src = "fgimg.gif" class = "fgimg"
         alt = "Second positioned image" /></p>

      <p class = "text">Positioned Text</p>
   </body>
</html>
```

Fig. 5.6 | Absolute positioning of elements. (Part 1 of 2.)


Absolute Positioning - Windows Internet Explorer
C:\examples\ch05\positioning.html
Google
Absolute Positioning
My Computer
100%

Positioned Text

Fig. 5.6 | Absolute positioning of elements. (Part 2 of 2.)

Normally, elements are positioned on the page in the order that they appear in the XHTML document. Lines 11–14 define a style called `bgimg` for the first `img` element (`i.gif`) on the page. Specifying an element's `position` as `absolute` removes the element from the normal flow of elements on the page, instead positioning it according to the distance from the `top`, `left`, `right` or `bottom` margins of its containing block-level element (i.e., an element such as `body` or `p`). Here, we position the element to be `0` pixels away from both the `top` and `left` margins of its containing element. In line 28, this style is applied to the image, which is contained in a `p` element.

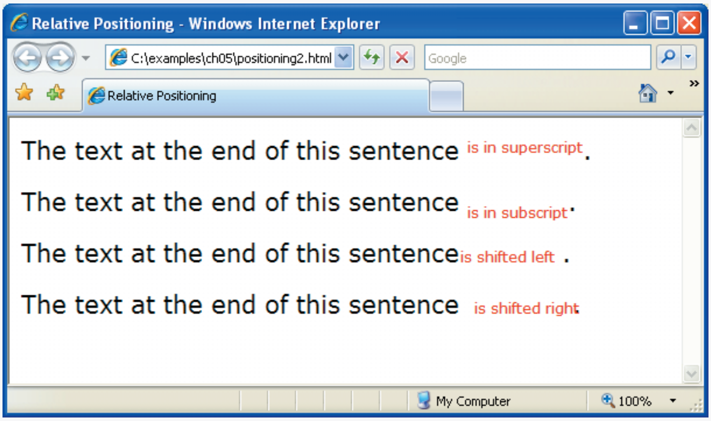
The **`z-index`** property allows you to layer overlapping elements properly. Elements that have higher `z-index` values are displayed in front of elements with lower `z-index` values. In this example, `i.gif` has the lowest `z-index` (`1`), so it displays in the background. The `.fgimg` CSS rule in lines 15–18 gives the circle image (`circle.gif`, in lines 31–32) a `z-index` of `2`, so it displays in front of `i.gif`. The `p` element in line 34 (`Positioned Text`) is given a `z-index` of `3` in line 22, so it displays in front of the other two. If you do not specify a `z-index` or if elements have the same `z-index` value, the elements are placed from background to foreground in the order they are encountered in the document.

Absolute positioning is not the only way to specify page layout. Figure 5.7 demonstrates relative positioning, in which elements are positioned relative to other elements.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 5.7: positioning2.html -->
<!-- Relative positioning of elements. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Relative Positioning</title>
```

Fig. 5.7 | Relative positioning of elements. (Part 1 of 2.)

```
10          <style type = "text/css">
11             p            { font-size: 1.3em;
12                            font-family: verdana, arial, sans-serif }
13             span         { color: red;
14                            font-size: .6em;
15                            height: 1em }
16             .super       { position: relative;
17                            top: -1ex }
18             .sub         { position: relative;
19                            bottom: -1ex }
20             .shiftleft   { position: relative;
21                            left: -1ex }
22             .shiftright { position: relative;
23                            right: -1ex }
24          </style>
25       </head>
26       <body>
27          <p>The text at the end of this sentence
28          <span class = "super">is in superscript</span>.</p>
29
30          <p>The text at the end of this sentence
31          <span class = "sub">is in subscript</span>.</p>
32
33          <p>The text at the end of this sentence
34          <span class = "shiftleft">is shifted left</span>.</p>
35
36          <p>The text at the end of this sentence
37          <span class = "shiftright">is shifted right</span>.</p>
38       </body>
39    </html>
```


Relative Positioning - Windows Internet Explorer
C:\examples\ch05\positioning2.html
Google
Relative Positioning
My Computer
100%

The text at the end of this sentence is in superscript.

The text at the end of this sentence is in subscript.

The text at the end of this sentence is shifted left .

The text at the end of this sentence is shifted right.

Fig. 5.7 | Relative positioning of elements. (Part 2 of 2.)

Setting the `position` property to `relative`, as in class `super` (lines 16–17), lays out the element on the page and offsets it by the specified `top`, `bottom`, `left` or `right` value. Unlike absolute positioning, relative positioning keeps elements in the general flow of elements on the page, so positioning is relative to other elements in the flow. Recall that `ex` (line 17) is the *x*-height of a font, a relative-length measurement typically equal to the height of a lowercase *x*.

Common Programming Error 5.2

Because relative positioning keeps elements in the flow of text in your documents, be careful to avoid unintentionally overlapping text.

Inline and Block-Level Elements

We introduce the **span** element in line 28. Lines 13–15 define the CSS rule for all span elements. The height of the span determines how much vertical space the span will occupy. The font-size determines the size of the text inside the span.

Element span is a **grouping element**—it does not apply any inherent formatting to its contents. Its primary purpose is to apply CSS rules or id attributes to a section of text. Element span is an **inline-level element**—it applies formatting to text without changing the flow of the document. Examples of inline elements include span, img, a, em and strong. The **div** element is also a grouping element, but it is a **block-level element**. This means it is displayed on its own line and has a virtual box around it. Examples of block-level elements include div, p and heading elements (h1 through h6). We'll discuss inline and block-level elements in more detail in Section 5.9.

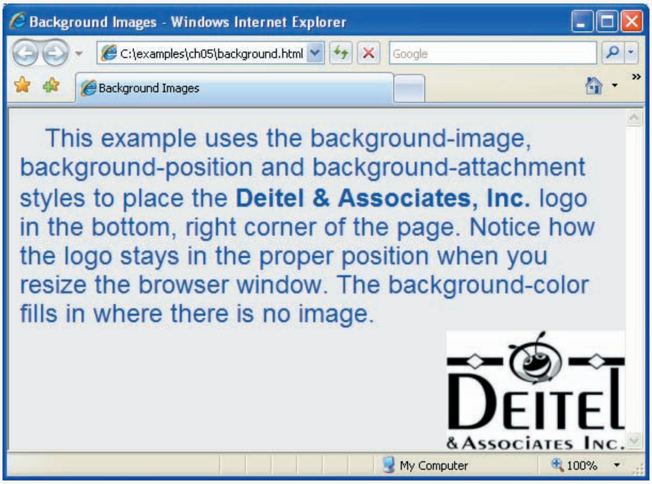
5.7 Backgrounds

CSS provides control over the background of block-level elements. CSS can set a background color or add background images to XHTML elements. Figure 5.8 adds a corporate logo to the bottom-right corner of the document. This logo stays fixed in the corner even when the user scrolls up or down the screen.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 5.8: background.html -->
<!-- Adding background images and indentation. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Background Images</title>
      <style type = "text/css">
         body  { background-image: url(logo.gif);
                 background-position: bottom right;
                 background-repeat: no-repeat;
                 background-attachment: fixed;
                 background-color: #eeeeee }
         p     { font-size: 18pt;
                 color: #1144AA;
                 text-indent: 1em;
                 font-family: arial, sans-serif; }
         .dark { font-weight: bold }
      </style>
   </head>
```

Fig. 5.8 | Adding background images and indentation. (Part 1 of 2.)

```
23      <body>
24         <p>
25         This example uses the background-image,
26         background-position and background-attachment
27         styles to place the <span class = "dark">Deitel
28         &amp; Associates, Inc.</span> logo in the bottom,
29         right corner of the page. Notice how the logo
30         stays in the proper position when you resize the
31         browser window. The background-color fills in where
32         there is no image.
33         </p>
34      </body>
35   </html>
```


Background Images - Windows Internet Explorer
C:\examples\ch05\background.html
Google
Background Images
My Computer
100%

This example uses the background-image, background-position and background-attachment styles to place the **Deitel & Associates, Inc.** logo in the bottom, right corner of the page. Notice how the logo stays in the proper position when you resize the browser window. The background-color fills in where there is no image.

Fig. 5.8 | Adding background images and indentation. (Part 2 of 2.)

The **background-image** property (line 11) specifies the image URL for the image `logo.gif` in the format `url(`*fileLocation*`)`. You can also set the **background-color** property in case the image is not found (and to fill in where the image does not cover).

The **background-position** property (line 12) places the image on the page. The keywords `top`, `bottom`, `center`, `left` and `right` are used individually or in combination for vertical and horizontal positioning. An image can be positioned using lengths by specifying the horizontal length followed by the vertical length. For example, to position the image as horizontally centered (positioned at 50 percent of the distance across the screen) and 30 pixels from the top, use

```
background-position: 50% 30px;
```

The **background-repeat** property (line 13) controls background image **tiling**, which places multiple copies of the image next to each other to fill the background. Here, we set the tiling to `no-repeat` to display only one copy of the background image. Other values include `repeat` (the default) to tile the image vertically and horizontally, `repeat-x` to tile the image only horizontally or `repeat-y` to tile the image only vertically.

The final property setting, **`background-attachment: fixed`** (line 14), fixes the image in the position specified by **`background-position`**. Scrolling the browser window will not move the image from its position. The default value, `scroll`, moves the image as the user scrolls through the document.

Line 18 uses the **`text-indent`** property to indent the first line of text in the element by a specified amount, in this case `1em`. An author might use this property to create a web page that reads more like a novel, in which the first line of every paragraph is indented.

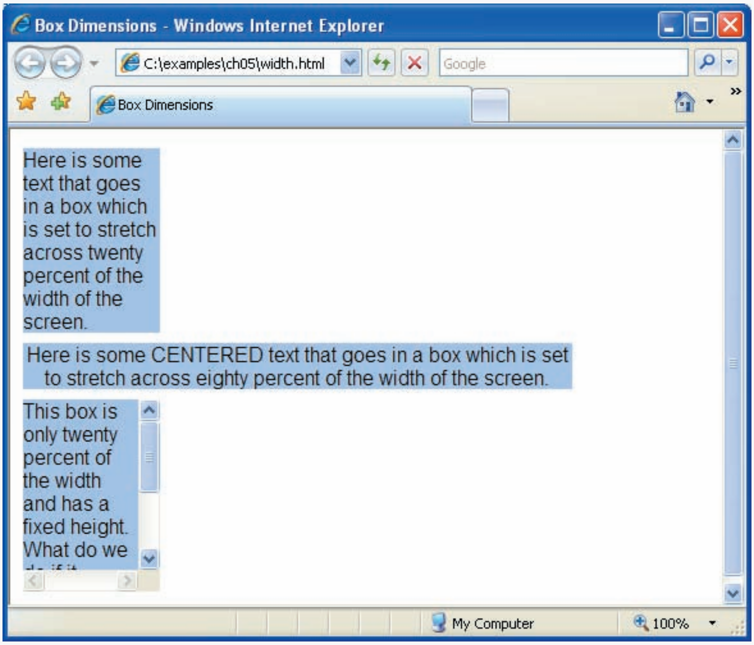
Another CSS property that formats text is the **`font-style`** property, which allows the developer to set text to `none`, `italic` or `oblique` (`oblique` is simply more slanted than italic—the browser will default to `italic` if the system or font does not support oblique text).

5.8 Element Dimensions

In addition to positioning elements, CSS rules can specify the actual dimensions of each page element. Figure 5.9 demonstrates how to set the dimensions of elements.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 5.9: width.html -->
<!-- Element dimensions and text alignment. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Box Dimensions</title>
      <style type = "text/css">
         div { background-color: #aaccff;
               margin-bottom: .5em;
               font-family: arial, helvetica, sans-serif }
      </style>
   </head>
   <body>
      <div style = "width: 20%">Here is some
      text that goes in a box which is
      set to stretch across twenty percent
      of the width of the screen.</div>

      <div style = "width: 80%; text-align: center">
      Here is some CENTERED text that goes in a box
      which is set to stretch across eighty percent of
      the width of the screen.</div>

      <div style = "width: 20%; height: 150px; overflow: scroll">
      This box is only twenty percent of
      the width and has a fixed height.
      What do we do if it overflows? Set the
      overflow property to scroll!</div>
   </body>
</html>
```

Fig. 5.9 | Element dimensions and text alignment. (Part 1 of 2.)


Box Dimensions - Windows Internet Explorer
C:\examples\ch05\width.html
Google
Box Dimensions
My Computer
100%

Here is some text that goes in a box which is set to stretch across twenty percent of the width of the screen.

Here is some CENTERED text that goes in a box which is set to stretch across eighty percent of the width of the screen.

This box is only twenty percent of the width and has a fixed height. What do we do if it

Fig. 5.9 | Element dimensions and text alignment. (Part 2 of 2.)

The inline style in line 17 illustrates how to set the **`width`** of an element on screen; here, we indicate that the `div` element should occupy 20 percent of the screen width. The height of an element can be set similarly, using the **`height`** property. The `height` and `width` values also can be specified as relative or absolute lengths. For example,

```
width: 10em
```

sets the element's width to 10 times the font size. Most elements are left aligned by default; however, this alignment can be altered to position the element elsewhere. Line 22 sets text in the element to be `center` aligned; other values for the **`text-align`** property include `left` and `right`.

In the third `div`, we specify a percentage height and a pixel width. One problem with setting both dimensions of an element is that the content inside the element can exceed the set boundaries, in which case the element is simply made large enough for all the content to fit. However, in line 27, we set the **`overflow`** property to `scroll`, a setting that adds scroll bars if the text overflows the boundaries.

5.9 Box Model and Text Flow

All block-level XHTML elements have a virtual box drawn around them based on what is known as the box model. When the browser renders elements using the box model, the content of each element is surrounded by padding, a border and a margin (Fig. 5.10).

CSS controls the border using three properties: **`border-width`**, **`border-color`** and **`border-style`**. We illustrate these three properties in Fig. 5.11.

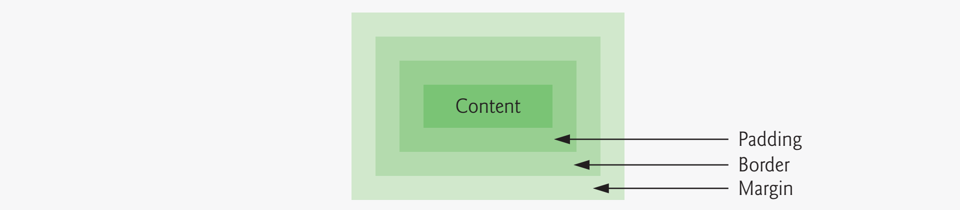


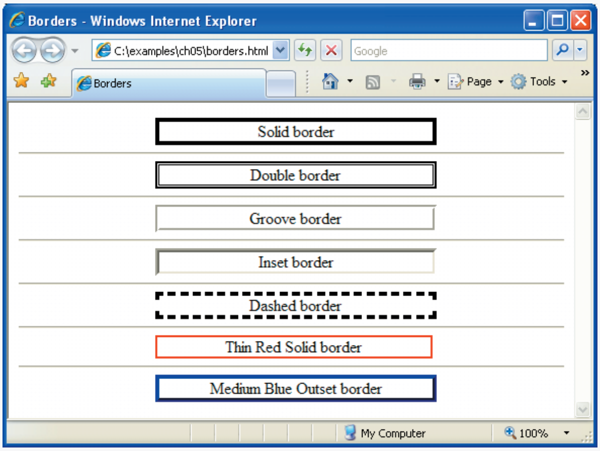
Fig. 5.10 | Box model for block-level elements.

Property `border-width` may be set to any valid CSS length (e.g., `em`, `ex`, `px`, etc.) or to the predefined value of `thin`, `medium` or `thick`. The **`border-color` property** sets the color. [*Note:* This property has different meanings for different style borders.] The `border-style` options are `none`, `hidden`, `dotted`, `dashed`, `solid`, `double`, `groove`, `ridge`, `inset` and `outset`. Borders `groove` and `ridge` have opposite effects, as do `inset` and `outset`. When border-style is set to none, no border is rendered.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 5.11: borders.html -->
<!-- Borders of block-level elements. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Borders</title>
      <style type = "text/css">
         div     { text-align: center;
                   width: 50%;
                   position: relative;
                   left: 25%;
                   border-width: 4px }
         .medium { border-width: medium }
         .thin   { border-width: thin }
         .solid  { border-style: solid }
         .double { border-style: double }
         .groove { border-style: groove }
         .inset  { border-style: inset }
         .outset { border-style: outset }
         .dashed { border-style: dashed }
         .red    { border-color: red }
         .blue   { border-color: blue }
      </style>
   </head>
   <body>
      <div class = "solid">Solid border</div><hr />
      <div class = "double">Double border</div><hr />
      <div class = "groove">Groove border</div><hr />
      <div class = "inset">Inset border</div><hr />
      <div class = "dashed">Dashed border</div><hr />
```

Fig. 5.11 | Borders of block-level elements. (Part 1 of 2.)

```
34        <div class = "thin red solid">Thin Red Solid border</div><hr />
35        <div class = "medium blue outset">Medium Blue Outset border</div
36     </body>
37  </html>
```


Borders - Windows Internet Explorer
C:\examples\ch05\borders.html
Google
Borders
Page
Tools
My Computer
100%

Solid border

Double border

Groove border

Inset border

Dashed border

Thin Red Solid border

Medium Blue Outset border

Fig. 5.11 | Borders of block-level elements. (Part 2 of 2.)

Each border property may be set for an individual side of the box (e.g., `border-top-style` or `border-left-color`). Note that we assign more than one class to an XHTML element by separating multiple class names with spaces, as shown in lines 36–37.

As we have seen with absolute positioning, it is possible to remove elements from the normal flow of text. Floating allows you to move an element to one side of the screen; other content in the document then flows around the floated element. Figure 5.12 demonstrates how floats and the box model can be used to control the layout of an entire page.

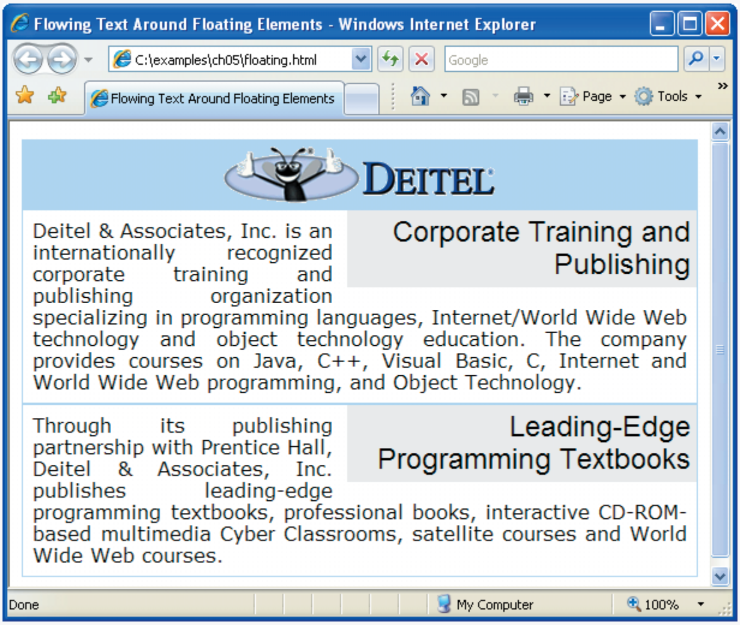
Looking at the XHTML code, we can see that the general structure of this document consists of a header and two main sections. Each section contains a subheading and a paragraph of text.

Block-level elements (such as `div`s) render with a line break before and after their content, so the header and two sections will render vertically one on top of another. In the absence of our styles, the subheading `div`s would also stack vertically on top of the text in the `p` tags. However, in line 24 we set the `float` property to `right` in the class `floated`, which is applied to the subheadings. This causes each subheading `div` to float to the right edge of its containing element, while the paragraph of text will flow around it.

Line 17 assigns a margin of `.5em` to all paragraph tags. The **`margin`** property sets the space between the outside of the border and all other content on the page. In line 21, we assign `.2em` of padding to the floated `div`s. The **`padding`** property determines the distance between the content inside an element and the inside of the element's border. Margins for individual sides of an element can be specified (lines 22–23) by using the properties **`margin-top`**, **`margin-right`**, **`margin-left`** and **`margin-bottom`**. Padding can be specified in the same way, using **`padding-top`**, **`padding-right`**, **`padding-left`** and **`padding-bottom`**. To see the effects of margins and padding, try putting the margin and padding properties inside comments and observing the difference.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 5.12: floating.html -->
<!-- Floating elements. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Flowing Text Around Floating Elements</title>
      <style type = "text/css">
         div.heading { background-color: #bbddff;
                       text-align: center;
                       font-family: arial, helvetica, sans-serif;
                       padding: .2em }
         p           { text-align: justify;
                       font-family: verdana, geneva, sans-serif;
                       margin: .5em }
         div.floated { background-color: #eeeeee;
                       font-size: 1.5em;
                       font-family: arial, helvetica, sans-serif;
                       padding: .2em;
                       margin-left: .5em;
                       margin-bottom: .5em;
                       float: right;
                       text-align: right;
                       width: 50% }
         div.section { border: 1px solid #bbddff }
      </style>
   </head>
   <body>
      <div class = "heading"><img src = "deitel.png" alt = "Deitel" />
         </div>
      <div class = "section">
         <div class = "floated">Corporate Training and Publishing</div>
         <p>Deitel &amp; Associates, Inc. is an internationally
         recognized corporate training and publishing organization
         specializing in programming languages, Internet/World
         Wide Web technology and object technology education.
         The company provides courses on Java, C++, Visual Basic, C#,
         C, Internet and web programming, Object
         Technology, and more.</p>
      </div>
      <div class = "section">
         <div class = "floated">Leading-Edge Programming Textbooks</div>
         <p>Through its publishing
         partnership with Prentice Hall, Deitel &amp; Associates,
         Inc. publishes leading-edge programming textbooks,
         professional books, interactive CD-ROM-based multimedia
         Cyber Classrooms, satellite courses and DVD and web-based
         video courses.</p>
      </div>
   </body>
</html>
```

Fig. 5.12 | Floating elements. (Part 1 of 2.)


Flowing Text Around Floating Elements - Windows Internet Explorer
C:\examples\ch05\floating.html
Google
Flowing Text Around Floating Elements
Page
Tools
DEITEL
Done
My Computer
100%

Corporate Training and Publishing

Deitel & Associates, Inc. is an internationally recognized corporate training and publishing organization specializing in programming languages, Internet/World Wide Web technology and object technology education. The company provides courses on Java, C++, Visual Basic, C, Internet and World Wide Web programming, and Object Technology.

Leading-Edge Programming Textbooks

Through its publishing partnership with Prentice Hall, Deitel & Associates, Inc. publishes leading-edge programming textbooks, professional books, interactive CD-ROM-based multimedia Cyber Classrooms, satellite courses and World Wide Web courses.

Fig. 5.12 | Floating elements. (Part 2 of 2.)

In line 27, we assign a border to the `section` boxes using a shorthand declaration of the border properties. CSS allows shorthand assignments of borders to allow you to define all three border properties in one line. The syntax for this shorthand is `border: <width> <style> <color>`. Our border is one pixel thick, `solid`, and the same color as the `background-color` property of the heading `div` (line 11). This allows the border to blend with the header and makes the page appear as one box with a line dividing its sections.

5.10 Media Types

CSS **media types** allow a programmer to decide what a page should look like depending on the kind of media being used to display the page. The most common media type for a web page is the **`screen` media type**, which is a standard computer screen. Other media types in CSS 2 include **`handheld`**, **`braille`**, **`aural`** and **`print`**. The `handheld` medium is designed for mobile Internet devices, while `braille` is for machines that can read or print web pages in braille. `aural` styles allow the programmer to give a speech-synthesizing web browser more information about the content of the web page. This allows the browser to present a web page in a sensible manner to a visually impaired person. The `print` media type affects a web page's appearance when it is printed. For a complete list of CSS media types, see `http://www.w3.org/TR/REC-CSS2/media.html#media-types`.

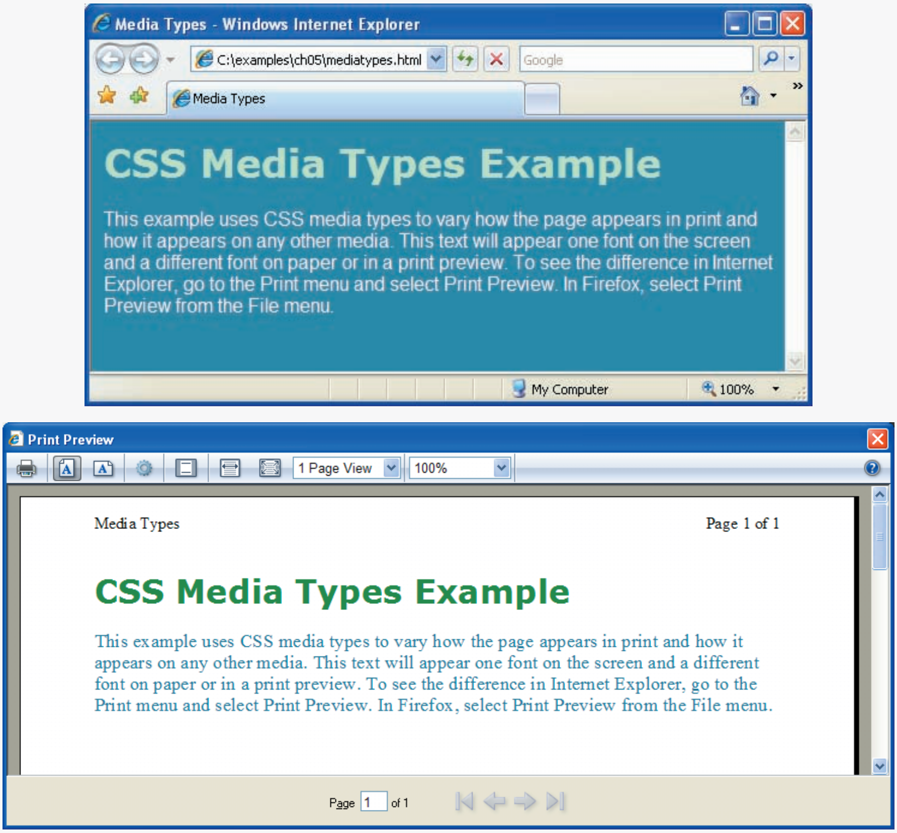
Media types allow a programmer to decide how a page should be presented on any one of these media without affecting the others. Figure 5.13 gives a simple example that applies one set of styles when the document is viewed on the screen, and another when the document is printed. To see the difference, look at the screen captures below the paragraph or use the **Print Preview** feature in Internet Explorer or Firefox.

In line 11, we begin a block of styles that applies to all media types, declared by @media all and enclosed in curly braces ({ and }). In lines 13–18, we define some styles for all media types. Lines 20–27 set styles to be applied only when the page is printed, beginning with the declaration @media print and enclosed in curly braces.

The styles we applied for all media types look nice on a screen but would not look good on a printed page. A colored background would use a lot of ink, and a black-and-white printer may print a page that's hard to read because there isn't enough contrast

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 5.13: mediatypes.html -->
 6  <!-- CSS media types. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Media Types</title>
10        <style type = "text/css">
11           @media all
12           {
13              body  { background-color: #4488aa }
14              h1    { font-family: verdana, helvetica, sans-serif;
15                      color: #aaffcc }
16              p     { font-size: 12pt;
17                      color: white;
18                      font-family: arial, sans-serif }
19           } /* end @media all declaration. */
20           @media print
21           {
22              body  { background-color: white }
23              h1    { color: #008844}
24              p     { font-size: 14pt;
25                      color: #4488aa;
26                      font-family: "times new roman", times, serif }
27           } /* end @media print declaration. */
28        </style>
29     </head>
30     <body>
31        <h1>CSS Media Types Example</h1>
32
33        <p>
34        This example uses CSS media types to vary how the page
35        appears in print and how it appears on any other media.
36        This text will appear one font on the screen and a
37        different font on paper or in a print preview. To see
38        the difference in Internet Explorer, go to the Print
39        menu and select Print Preview. In Firefox, select Print
40        Preview from the File menu.
41        </p>
42     </body>
43  </html>
```

Fig. 5.13 | CSS media types. (Part 1 of 2.)



CSS Media Types Example

This example uses CSS media types to vary how the page appears in print and how it appears on any other media. This text will appear one font on the screen and a different font on paper or in a print preview. To see the difference in Internet Explorer, go to the Print menu and select Print Preview. In Firefox, select Print Preview from the File menu.

CSS Media Types Example

This example uses CSS media types to vary how the page appears in print and how it appears on any other media. This text will appear one font on the screen and a different font on paper or in a print preview. To see the difference in Internet Explorer, go to the Print menu and select Print Preview. In Firefox, select Print Preview from the File menu.

Fig. 5.13 | CSS media types. (Part 2 of 2.)

between the colors. Also, `sans-serif` fonts like `arial`, `helvetica`, and `geneva` are easier to read on a screen, while `serif` fonts like `times new roman` are easier to read on paper.

Look-and-Feel Observation 5.1

Pages with dark background colors and light text use a lot of ink and may be difficult to read when printed, especially on a black-and white-printer. Use the `print` media type to avoid this.

Look-and-Feel Observation 5.2

In general, sans-serif fonts look better on a screen, while serif fonts look better on paper. The `print` media type allows your web page to display sans-serif font on a screen and change to a serif font when it is printed.

To solve these problems, we apply specific styles for the `print` media type. We change the `background-color` of the `body`, the `color` of the `h1` tag, and the `font-size`, `color`, and `font-family` of the `p` tag to be more suited for printing and viewing on paper. Notice that most of these styles conflict with the declarations in the section for all media types. Since the `print` media type has higher specificity than `all` media types, the `print` styles override the styles for all media types when the page is printed. Since the `font-family` property of the h1 tag is not overridden in the print section, it retains its old value even when the page is printed.

5.11 Building a CSS Drop-Down Menu

Drop-down menus are a good way to provide navigation links on a website without using a lot of screen space. In this section, we take a second look at the :hover pseudoclass and introduce the display property to create a drop-down menu using CSS and XHTML.

We've already seen the :hover pseudoclass used to change a link's style when the mouse hovers over it. We will use this feature in a more advanced way to cause a menu to appear when the mouse hovers over a menu button. The other important property we need is the **display** property. This allows a programmer to decide whether an element is rendered on the page or not. Possible values include block, inline and none. The block and inline values display the element as a block element or an inline element, while none stops the element from being rendered. The code for the drop-down menu is shown in Fig. 5.14.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 5.14: dropdown.html -->
<!-- CSS drop-down menu. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>
         Drop-Down Menu
      </title>
      <style type = "text/css">
         body                  { font-family: arial, sans-serif }
         div.menu              { font-weight: bold;
                                 color: white;
                                 border: 2px solid #225599;
                                 text-align: center;
                                 width: 10em;
                                 background-color: #225599 }
         div.menu:hover a      { display: block }
         div.menu a            { display: none;
                                 border-top: 2px solid #225599;
                                 background-color: white;
                                 width: 10em;
                                 text-decoration: none;
                                 color: black }
         div.menu a:hover      { background-color: #dfeeff }
      </style>
   </head>
   <body>
      <div class = "menu">Menu
         <a href = "#">Home</a>
         <a href = "#">News</a>
```

Fig. 5.14 | CSS drop-down menu. (Part 1 of 2.)

```
34             <a href = "#">Articles</a>
35             <a href = "#">Blog</a>
36             <a href = "#">Contact</a>
37          </div>
38       </body>
39    </html>
```

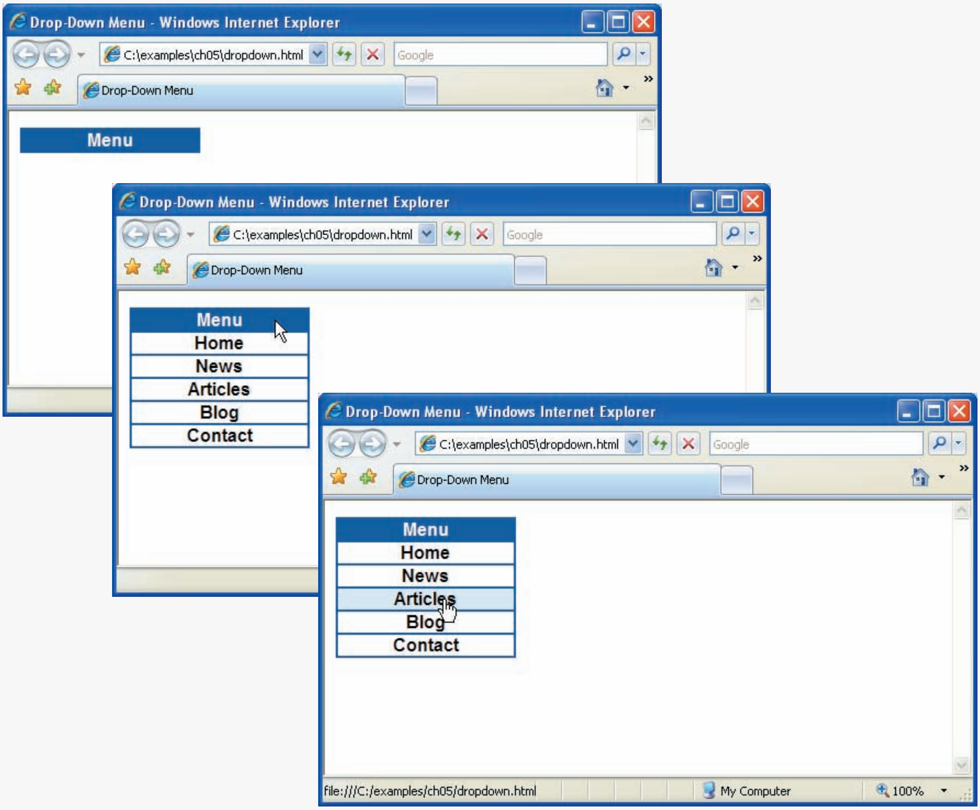

Drop-Down Menu - Windows Internet Explorer
C:\examples\ch05\dropdown.html
Google
Drop-Down Menu
Menu
Home
News
Articles
Blog
Contact
file:///C:/examples/ch05/dropdown.html
My Computer
100%

Fig. 5.14 | CSS drop-down menu. (Part 2 of 2.)

First let's look at the XHTML code. In lines 31–37, a `div` of class `menu` has the text "Menu" and five links inside it. This is our drop-down menu. The behavior we want is as follows: the text that says "Menu" should be the only thing visible on the page, unless the mouse is over the menu `div`. When the mouse cursor hovers over the menu `div`, we want the links to appear below the menu for the user to choose from.

To see how we get this functionality, let's look at the CSS code. There are two lines that give us the drop-down functionality. Line 21 selects all the links inside the menu `div` and sets their `display` value to `none`. This instructs the browser not to render the links. The other important style is in line 20. The selectors in this line are similar to those in line 21, except that this line selects only the `a` (anchor) elements that are children of a menu `div` that has the mouse over it. The `display: block` in this line means that when the mouse is over the menu `div`, the links inside it will be displayed as block-level elements.

The selectors in line 27 are also similar to lines 20 and 21. This time, however, the style is applied only to any a element that is a child of the menu div when that child has the mouse cursor over it. This style changes the background-color of the currently highlighted menu option. The rest of the CSS simply adds aesthetic style to the components of our menu. Look at the screen captures or run the code example to see the menu in action.

This drop-down menu is just one example of more advanced CSS formatting. Many additional resources are available online for CSS navigation menus and lists. Specifically, check out List-o-Matic, an automatic CSS list generator located at www.accessify.com/tools-and-wizards/developer-tools/list-o-matic/ and Dynamic Drive's library of vertical and horizontal CSS menus at www.dynamicdrive.com/style/.

5.12 User Style Sheets

Users can define their own **user style sheets** to format pages based on their preferences. For example, people with visual impairments may want to increase the page's text size. Web page authors need to be careful not to inadvertently override user preferences with defined styles. This section discusses possible conflicts between **author styles** and **user styles**.

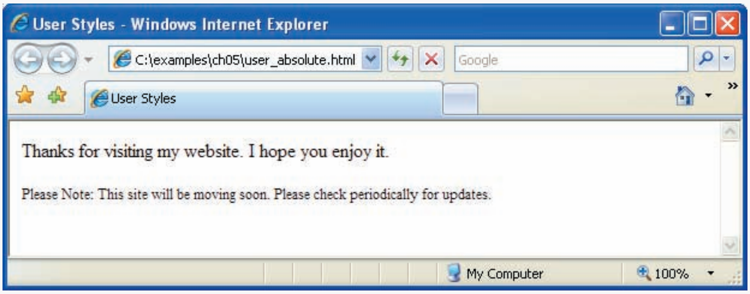
Figure 5.15 contains an author style. The font-size is set to 9pt for all <p> tags that have class note applied to them.

User style sheets are external style sheets. Figure 5.16 shows a user style sheet that sets the body's font-size to 20pt, color to yellow and background-color to #000080.

User style sheets are not linked to a document; rather, they are set in the browser's options. To add a user style sheet in IE7, select **Internet Options...**, located in the **Tools** menu. In the **Internet Options** dialog (Fig. 5.17) that appears, click **Accessibility...**, check the **Format documents using my style sheet** checkbox, and type the location of the user style sheet. Internet Explorer 7 applies the user style sheet to any document it loads. To add a user style sheet in Firefox, find your Firefox profile using the instructions at

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 5.15: user_absolute.html -->
<!-- pt measurement for text size. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>User Styles</title>
      <style type = "text/css">
         .note { font-size: 9pt }
      </style>
   </head>
   <body>
      <p>Thanks for visiting my website. I hope you enjoy it.
      </p><p class = "note">Please Note: This site will be
      moving soon. Please check periodically for updates.</p>
   </body>
</html>
```

Fig. 5.15 | pt measurement for text size. (Part 1 of 2.)


User Styles - Windows Internet Explorer
C:\examples\ch05\user_absolute.html
Google
User Styles
My Computer
100%

Thanks for visiting my website. I hope you enjoy it.

Please Note: This site will be moving soon. Please check periodically for updates.

Fig. 5.15 | `pt` measurement for text size. (Part 2 of 2.)

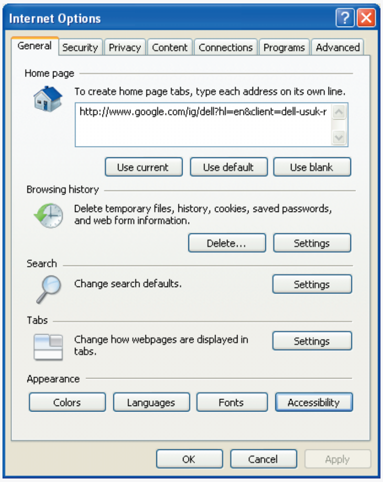
```
/* Fig. 5.16: userstyles.css */
/* A user stylesheet */
body     { font-size: 20pt;
           color: yellow;
           background-color: #000080 }
```

Fig. 5.16 | User style sheet.

`www.mozilla.org/support/firefox/profile#locate` and place a style sheet called `userContent.css` in the `chrome` subdirectory.

The web page from Fig. 5.15 is displayed in Fig. 5.18, with the user style sheet from Fig. 5.16 applied.

In this example, if users define their own `font-size` in a user style sheet, the author style has a higher precedence and overrides the user style. The `9pt` font specified in the author style sheet overrides the `20pt` font specified in the user style sheet. This small font



To create home page tabs, type each address on its own line.

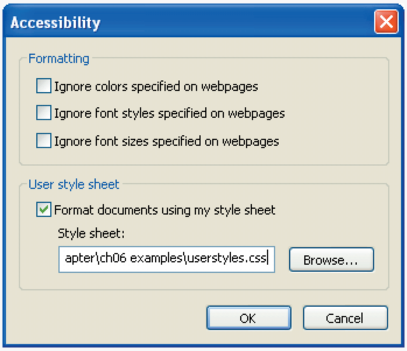
http://www.google.com/ig/dell?hl=en&client=dell-usuk-r

Delete temporary files, history, cookies, saved passwords, and web form information.

Change search defaults.

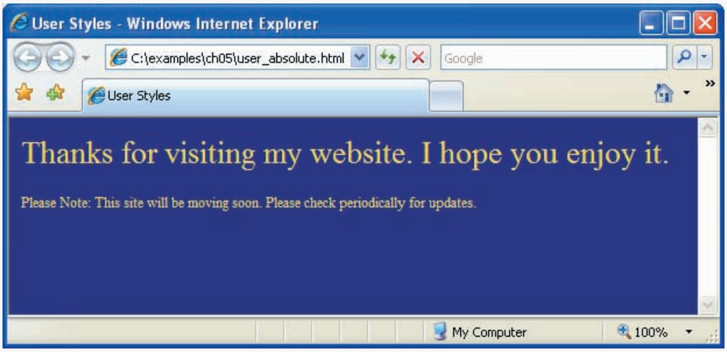
Change how webpages are displayed in tabs.

Fig. 5.17 | User style sheet in Internet Explorer 7. (Part 1 of 2.)



□ Ignore colors specified on webpages
□ Ignore font styles specified on webpages
□ Ignore font sizes specified on webpages
☑ Format documents using my style sheet
Style sheet: apter\ch06 examples\userstyles.css

Fig. 5.17 | User style sheet in Internet Explorer 7. (Part 2 of 2.)


User Styles - Windows Internet Explorer
C:\examples\ch05\user_absolute.html
Google
User Styles
My Computer
100%

Thanks for visiting my website. I hope you enjoy it.

Please Note: This site will be moving soon. Please check periodically for updates.

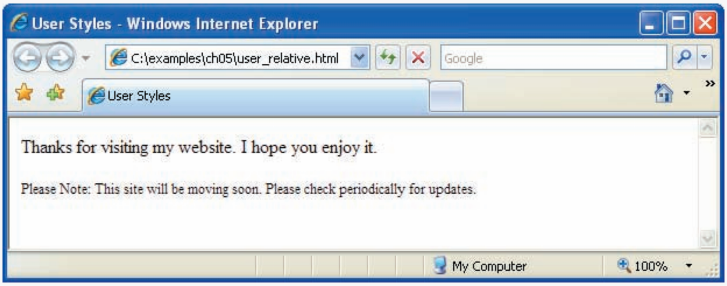
Fig. 5.18 | User style sheet applied with `pt` measurement.

may make pages difficult to read, especially for individuals with visual impairments. You can avoid this problem by using relative measurements (e.g., `em` or `ex`) instead of absolute measurements, such as `pt`. Figure 5.19 changes the `font-size` property to use a relative measurement (line 11) that does not override the user style set in Fig. 5.16. Instead, the font size displayed is relative to the one specified in the user style sheet. In this case, text enclosed in the <p> tag displays as `20pt`, and <p> tags that have class `note` applied to them are displayed in `15pt` (`.75` times `20pt`).

```
1  <?xml version = "1.0" encoding = "utf-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 5.19: user_relative.html -->
6  <!-- em measurement for text size. -->
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8     <head>
9        <title>User Styles</title>
```

Fig. 5.19 | `em` measurement for text size. (Part 1 of 2.)

```
      <style type = "text/css">
         .note { font-size: .75em }
      </style>
   </head>
   <body>
      <p>Thanks for visiting my website. I hope you enjoy it.
      </p><p class = "note">Please Note: This site will be
      moving soon. Please check periodically for updates.</p>
   </body>
</html>
```

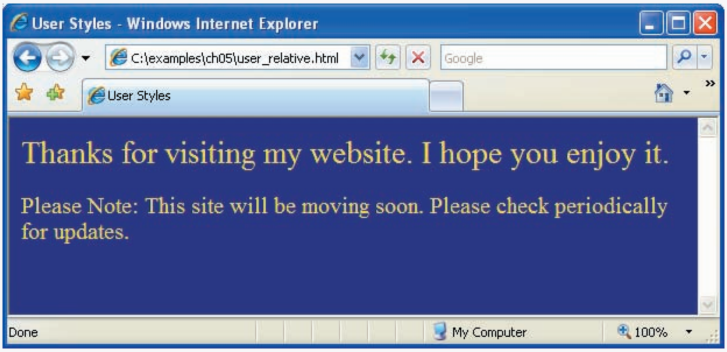

User Styles - Windows Internet Explorer
C:\examples\ch05\user_relative.html
Google
User Styles
My Computer
100%

Thanks for visiting my website. I hope you enjoy it.

Please Note: This site will be moving soon. Please check periodically for updates.

Fig. 5.19 | em measurement for text size. (Part 2 of 2.)

Figure 5.20 displays the web page from Fig. 5.19 with the user style sheet from Fig. 5.16 applied. Note that the second line of text displayed is larger than the same line of text in Fig. 5.18.


User Styles - Windows Internet Explorer
C:\examples\ch05\user_relative.html
Google
User Styles
Done
My Computer
100%

Thanks for visiting my website. I hope you enjoy it.

Please Note: This site will be moving soon. Please check periodically for updates.

Fig. 5.20 | User style sheet applied with `em` measurement.

5.13 CSS 3

The W3C is currently developing CSS 3 and some browsers are beginning to implement some of the new features that will be in the CSS 3 specification. We discuss a few of the upcoming features that will most likely be included in CSS 3.

CSS 3 will allow for more advanced control of borders. In addition to the `border-style`, `border-color`, and `border-width` properties, you will be able to set multiple border colors, use images for borders, add shadows to boxes, and create borders with rounded corners.

Background images will be more versatile in CSS 3, allowing the programmer to set the size of a background image, specify an offset to determine where in the element the image should be positioned, and use multiple background images in one element. There will also be properties to set shadow effects on text and more options for text overflow when the text is too long to fit in its containing element.

Additional features will include resizable boxes, enhanced selectors, multicolumn layouts, and more developed speech (aural) styles. The Web Resources section points you to the Deitel CSS Resource Center, where you can find links to the latest information on the development and features of CSS 3.

5.14 Wrap-Up

In this chapter we introduced Cascading Style Sheets, a technology that allows us to create flexibility in formatting XHTML documents. We learned how to define styles inline, using embedded style sheets, and linking to external style sheets. We learned that external style sheets achieve true separation of content and presentation by putting each in its own file. We discussed how style sheets "cascade" to resolve conflicting style property definitions. We used absolute and relative positioning, background images and colors, and set the dimensions of XHTML elements.

We also learned about the box model and how to use borders, padding, and margins, as well as how to float elements to one side of the page. We provided examples of a web page with separate styles for the `print` media type, a drop-down menu using the `display` property, and learned more about accessibility and user styles.

XHTML and CSS are the basic technologies behind web pages. In the next chapter, we begin our treatment of JavaScript, which allows us to make more interactive and dynamic pages.

5.15 Web Resources

`http://www.deitel.com/css21`

The Deitel CSS Resource Center contains links to some of the best CSS information on the web. There you'll find categorized links to tutorials, references, code examples, demos, videos, and more. Check out the demos section for more advanced examples of layouts, menus, and other web page components.

Summary

Section 5.2 Inline Styles

- The inline style allows you to declare a style for an individual element by using the `style` attribute in the element's start tag.
- Each CSS property is followed by a colon and the value of the attribute. Multiple property declarations are separated by a semicolon.
- The `color` property sets text color. Color names and hexadecimal codes may be used as the value.

Section 5.3 Embedded Style Sheets

- Styles that are placed in a `style` element use selectors to apply style elements throughout the entire document.
- `style` element attribute `type` specifies the MIME type (the specific encoding format) of the style sheet. Style sheets use `text/css`.
- Each rule body in a style sheet begins and ends with a curly brace (`{` and `}`).
- The `font-weight` property specifies the "boldness" of text. Possible values are `bold`, `normal` (the default), `bolder` (bolder than bold text) and `lighter` (lighter than `normal` text).
- Boldness also can be specified with multiples of 100, from 100 to 900 (e.g., `100`, `200`, …, `900`). Text specified as `normal` is equivalent to `400`, and `bold` text is equivalent to `700`.
- Style-class declarations are preceded by a period and are applied to elements of the specific class. The `class` attribute applies a style class to an element.
- The CSS rules in a style sheet use the same format as inline styles: The property is followed by a colon (`:`) and the value of that property. Multiple properties are separated by semicolons (`;`).
- The `background-color` attribute specifies the background color of the element.
- The `font-family` attribute names a specific font that should be displayed. Generic font families allow authors to specify a type of font instead of a specific font, in case a browser does not support a specific font. The `font-size` property specifies the size used to render the font.

Section 5.4 Conflicting Styles

- Most styles are inherited from parent elements. Styles defined for children have higher specificity and take precedence over the parent's styles.
- Pseudoclasses give the author access to content not specifically declared in the document. The `hover` pseudoclass is activated when the user moves the mouse cursor over an element.
- The `text-decoration` property applies decorations to text in an element, such as `underline`, `overline`, `line-through` and `blink`.
- To apply rules to multiple elements, separate the elements with commas in the style sheet.
- To apply rules to only a certain type of element that is a child of another type, separate the element names with spaces.
- A pixel is a relative-length measurement: It varies in size based on screen resolution. Other relative lengths are `em`, `ex` and percentages.
- The other units of measurement available in CSS are absolute-length measurements—that is, units that do not vary in size. These units can be `in` (inches), `cm` (centimeters), `mm` (millimeters), `pt` (points; 1 `pt` = 1/72 `in`) or `pc` (picas; 1 `pc` = 12 `pt`).

Section 5.5 Linking External Style Sheets

- External linking of style sheets can create a uniform look for a website; separate pages can all use the same styles. Modifying a single style-sheet file makes changes to styles across an entire website.
- `link`'s `rel` attribute specifies a relationship between two documents. For style sheets, the `rel` attribute declares the linked document to be a `stylesheet` for the document. The `type` attribute specifies the MIME type of the related document as `text/css`. The `href` attribute provides the URL for the document containing the style sheet.

Section 5.6 Positioning Elements

- The CSS `position` property allows absolute positioning, which provides greater control over where on a page elements reside. Specifying an element's `position` as `absolute` removes it from the normal flow of elements on the page and positions it according to distance from the top, left, right or bottom margin of its parent element.

- The z-index property allows a developer to layer overlapping elements. Elements that have higher z-index values are displayed in front of elements with lower z-index values.
- Unlike absolute positioning, relative positioning keeps elements in the general flow on the page and offsets them by the specified top, left, right or bottom value.
- Element span is a grouping element—it does not apply any inherent formatting to its contents. Its primary purpose is to apply CSS rules or id attributes to a section of text.
- span is an inline-level element—it applies formatting to text without changing the flow of the document. Examples of inline elements include span, img, a, em and strong.
- The div element is also a grouping element, but it is a block-level element. This means it is displayed on its own line and has a virtual box around it. Examples of block-level elements include div, p and heading elements (h1 through h6).

Section 5.7 Backgrounds

- Property background-image specifies the URL of the image, in the format url(fileLocation). The property background-position places the image on the page using the values top, bottom, center, left and right individually or in combination for vertical and horizontal positioning. You can also position by using lengths.
- The background-repeat property controls the tiling of the background image. Setting the tiling to no-repeat displays one copy of the background image on screen. The background-repeat property can be set to repeat (the default) to tile the image vertically and horizontally, to repeat-x to tile the image only horizontally or to repeat-y to tile the image only vertically.
- The property setting background-attachment: fixed fixes the image in the position specified by background-position. Scrolling the browser window will not move the image from its set position. The default value, scroll, moves the image as the user scrolls the window.
- The text-indent property indents the first line of text in the element by the specified amount.
- The font-style property allows you to set text to none, italic or oblique (oblique will default to italic if the system does not have a separate font file for oblique text, which is normally the case).

Section 5.8 Element Dimensions

- The dimensions of elements on a page can be set with CSS by using properties height and width.
- Text in an element can be centered using text-align; other values for the text-align property are left and right.
- One problem with setting both vertical and horizontal dimensions of an element is that the content inside the element might sometimes exceed the set boundaries, in which case the element must be made large enough for all the content to fit. However, a developer can set the overflow property to scroll; this setting adds scroll bars if the text overflows the boundaries set for it.

Section 5.9 Box Model and Text Flow

- The border-width property may be set to any of the CSS lengths or to the predefined value of thin, medium or thick.
- The border-styles available are none, hidden, dotted, dashed, solid, double, groove, ridge, inset and outset.
- The border-color property sets the color used for the border.
- The class attribute allows more than one class to be assigned to an XHTML element by separating each class name from the next with a space.

- Browsers normally place text and elements on screen in the order in which they appear in the XHTML file. Elements can be removed from the normal flow of text. Floating allows you to move an element to one side of the screen; other content in the document will then flow around the floated element.
- CSS uses a box model to render elements on screen. The content of each element is surrounded by padding, a border and margins. The properties of this box are easily adjusted.
- The `margin` property determines the distance between the element's edge and any outside text.
- Margins for individual sides of an element can be specified by using `margin-top`, `margin-right`, `margin-left` and `margin-bottom`.
- The `padding` property determines the distance between the content inside an element and the edge of the element. Padding also can be set for each side of the box by using `padding-top`, `padding-right`, `padding-left` and `padding-bottom`.

Section 5.10 Media Types

- CSS media types allow a programmer to decide what a page should look like depending on the kind of media being used to display the page. The most common media type for a web page is the `screen` media type, which is a standard computer screen.
- A block of styles that applies to all media types is declared by `@media all` and enclosed in curly braces. To create a block of styles that apply to a single media type such as `print`, use `@media print` and enclose the style rules in curly braces.
- Other media types in CSS 2 include `handheld`, `braille`, `aural` and `print`. The `handheld` medium is designed for mobile Internet devices, while `braille` is for machines that can read or print web pages in braille. `aural` styles allow the programmer to give a speech-synthesizing web browser more information about the content of the web page. The `print` media type affects a web page's appearance when it is printed.

Section 5.11 Building a CSS Drop-Down Menu

- The `:hover` pseudoclass is used to apply styles to an element when the mouse cursor is over it.
- The `display` property allows a programmer to decide if an element is displayed as a `block` element, `inline` element, or is not rendered at all (`none`).

Section 5.12 User Style Sheets

- Users can define their own user style sheets to format pages based on their preferences.
- Absolute font size measurements override user style sheets, while relative font sizes will yield to a user-defined style.

Section 5.13 CSS 3

- While CSS 2 is the current W3C Recommendation, CSS 3 is in development, and some browsers are beginning to implement some of the new features that will be in the CSS 3 specification.
- CSS 3 will introduce new features related to borders, backgrounds, text effects, layout, and more.

Terminology

absolute positioning
absolute-length measurement
ancestor element
`arial` font
`aural` media type
author style
`background-attachment` property
`background-color` property
`background-image` property
`background-position` property
`background-repeat` property
`blink` text decoration

block-level element
border
border-color property
border-style property
border-width property
box model
braille media type
Cascading Style Sheets (CSS)
class attribute
cm (centimeter)
colon (:)
color property
CSS 2
CSS 3
CSS comment
CSS property
CSS rule
CSS selector
cursive generic font family
dashed border style
descendant element
display property
div element
dotted border style
double border style
em (*M*-height of font)
embedded style sheet
ex (*x*-height of font)
external style sheets
float property
floated element
font-family property
font-size property
font-style property
font-weight property
generic font family
groove border style
grouping element
handheld media type
height property
hidden border style
in (inch)
inheritance
inline-level element
inline style
inset border style
large relative font size
larger relative font size
left property value
line-through text decoration
link element
linking to an external style sheet
margin property
margin-bottom property
margin-left property
margin-right property
margin-top property
media types
medium relative border width
medium relative font size
MIME (Multipurpose Internet Mail Extensions) type
mm (millimeter)
monospace font
none border style
outset border style
overflow property
overline text decoration
padding property
parent element
pc (pica)
position property
print media type
pseudoclass
pt (point)
rel attribute (link)
relative positioning
relative-length measurement
repeat property value
ridge border style
right property value
sans-serif generic font family
screen media type
scroll property value
selector
separation of structure from content
serif generic font family
small relative font size
smaller relative font size
solid border style
span element
specificity
style
style attribute
style class
style in header section of document
text flow
text/css MIME type
text-align property
text-decoration property

`text-indent` property
`thick` border width
`thin` border width
user agent
user style sheet
`width` property
`x-large` relative font size
`x-small` relative font size
`xx-large` relative font size
`xx-small` relative font size
`z-index` property

Self-Review Exercises

5.1 Assume that the size of the base font on a system is 12 points.
a) How big is a 36-point font in `ems`?
b) How big is a 9-point font in `ems`?
c) How big is a 24-point font in picas?
d) How big is a 12-point font in inches?
e) How big is a 1-inch font in picas?

5.2 Fill in the blanks in the following statements:
a) Using the ________ element allows authors to use external style sheets in their pages.
b) To apply a CSS rule to more than one element at a time, separate the element names with a(n) ________.
c) Pixels are a(n) ________-length measurement unit.
d) The ________ pseudoclass is activated when the user moves the mouse cursor over the specified element.
e) Setting the `overflow` property to ________ provides a mechanism for containing inner content without compromising specified box dimensions.
f) ________ is a generic inline element that applies no inherent formatting and ________ is a generic block-level element that applies no inherent formatting.
g) Setting property `background-repeat` to ________ tiles the specified `background-image` vertically.
h) To begin a block of styles that applies to only the `print` media type, you use the declaration ________ `print`, followed by an opening curly brace (`{`).
i) The ________ property allows you to indent the first line of text in an element.
j) The three components of the box model are the ________, ________ and ________.

Answers to Self-Review Exercises

5.1 a) 3 `ems`. b) 0.75 `ems`. c) 2 picas. d) 1/6 inch. e) 6 picas.

5.2 a) `link`. b) comma. c) relative. d) `hover`. e) `scroll`. f) `span`, `div`. g) `repeat-y`. h) `@media`. i) `text-indent`. j) padding, border, margin.

Exercises

5.3 Write a CSS rule that makes all text 1.5 times larger than the base font of the system and colors the text red.

5.4 Write a CSS rule that places a background image halfway down the page, tiling it horizontally. The image should remain in place when the user scrolls up or down.

5.5 Write a CSS rule that gives all `h1` and `h2` elements a padding of 0.5 `ems`, a `dashed` border style and a margin of 0.5 `ems`.

5.6 Write a CSS rule that changes the color of all elements containing attribute `class = "greenMove"` to green and shifts them down 25 pixels and right 15 pixels.

5.7 Make a layout template that contains a header and two columns. Use `div`s for the each layout component, and use `float` to line up the columns side by side. Give each component a border and/or a background color so you can see where your `div`s are.

5.8 Add an embedded style sheet to the XHTML document in Fig. 4.5. The style sheet should contain a rule that displays `h1` elements in blue. In addition, create a rule that displays all links in blue without underlining them. When the mouse hovers over a link, change the link's background color to yellow.

5.9 Make a navigation button using a `div` with a link inside it. Give it a border, background, and text color, and make them change when the user hovers the mouse over the button. Use an external style sheet. Make sure your style sheet validates at `http://jigsaw.w3.org/css-validator/`. Note that some warnings may be unavoidable, but your CSS should have no errors.

6

JavaScript: Introduction to Scripting

Comment is free, but facts are sacred.
—C. P. Scott

The creditor hath a better memory than the debtor.
—James Howell

When faced with a decision, I always ask, "What would be the most fun?"
—Peggy Walker

Equality, in a social sense, may be divided into that of condition and that of rights.
—James Fenimore Cooper

OBJECTIVES

In this chapter you will learn:

- To write simple JavaScript programs.
- To use input and output statements.
- Basic memory concepts.
- To use arithmetic operators.
- The precedence of arithmetic operators.
- To write decision-making statements.
- To use relational and equality operators.

Outline

6.1 Introduction

In the first five chapters, we introduced the Internet and Web, web browsers, Web 2.0, XHTML and Cascading Style Sheets (CSS). In this chapter, we begin our introduction to the **JavaScript**[1] **scripting language**, which facilitates a disciplined approach to designing computer programs that enhance the functionality and appearance of web pages.[2]

In Chapters 6–11, we present a detailed discussion of JavaScript—the *de facto* standard client-side scripting language for web-based applications due to its highly portable nature. Our treatment of JavaScript serves two purposes—it introduces client-side scripting (used in Chapters 6–13), which makes web pages more dynamic and interactive, and it provides the programming foundation for the more complex server-side scripting presented later in the book.

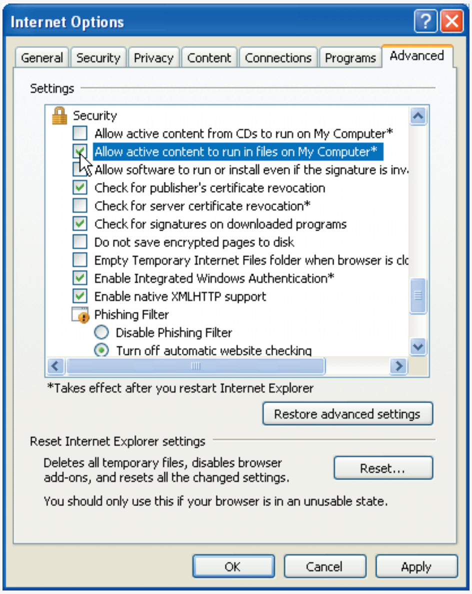
We now introduce JavaScript programming and present examples that illustrate several important features of JavaScript. Each example is carefully analyzed one line at a time. In Chapters 7–8, we present a detailed treatment of program development and program control in JavaScript.

Before you can run code examples with JavaScript on your computer, you may need to change your browser's security settings. By default, Internet Explorer 7 prevents scripts on your local computer from running, displaying a yellow warning bar at the top of the window instead. To allow scripts to run in files on your computer, select **Internet Options** from the **Tools** menu. Click the **Advanced** tab and scroll down to the **Security** section of

[1] Many people confuse the scripting language JavaScript with the programming language Java (from Sun Microsystems, Inc.). Java is a full-fledged object-oriented programming language. It can be used to develop applications that execute on a range of devices—from the smallest devices (such as cell phones and PDAs) to supercomputers. Java is popular for developing large-scale distributed enterprise applications and web applications. JavaScript is a browser-based scripting language developed by Netscape and implemented in all major browsers.

[2] JavaScript was originally created by Netscape. Both Netscape and Microsoft have been instrumental in the standardization of JavaScript by ECMA International as ECMAScript. Detailed information about the current ECMAScript standard can be found at `www.ecma-international.org/publications/standards/ECMA-262.htm`.

the **Settings** list. Check the box labeled **Allow active content to run in files on My Computer** (Fig. 6.1). Click **OK** and restart Internet Explorer. XHTML documents on your own computer that contain JavaScript code will now run properly. Firefox has JavaScript enabled by default.



Settings

Security
☐ Allow active content from CDs to run on My Computer*
☑ Allow active content to run in files on My Computer*
☐ Allow software to run or install even if the signature is inv
☑ Check for publisher's certificate revocation
☐ Check for server certificate revocation*
☑ Check for signatures on downloaded programs
☐ Do not save encrypted pages to disk
☐ Empty Temporary Internet Files folder when browser is clc
☑ Enable Integrated Windows Authentication*
☑ Enable native XMLHTTP support
Phishing Filter
○ Disable Phishing Filter
● Turn off automatic website checking

*Takes effect after you restart Internet Explorer

Reset Internet Explorer settings

Deletes all temporary files, disables browser add-ons, and resets all the changed settings.

You should only use this if your browser is in an unusable state.

Fig. 6.1 | Enabling JavaScript in Internet Explorer 7

6.2 Simple Program: Displaying a Line of Text in a Web Page

JavaScript uses notations that may appear strange to nonprogrammers. We begin by considering a simple script (or program) that displays the text `"Welcome to JavaScript Programming!"` in the body of an XHTML document. All major web browsers contain JavaScript interpreters, which process the commands written in JavaScript. The JavaScript code and its output in Internet Explorer are shown in Fig. 6.2.

This program illustrates several important JavaScript features. We consider each line of the XHTML document and script in detail. As in the preceding chapters, we have given each XHTML document line numbers for the reader's convenience; the line numbers are not part of the XHTML document or of the JavaScript programs. Lines 12–13 do the "real work" of the script, namely, displaying the phrase `Welcome to JavaScript Programming!` in the web page.

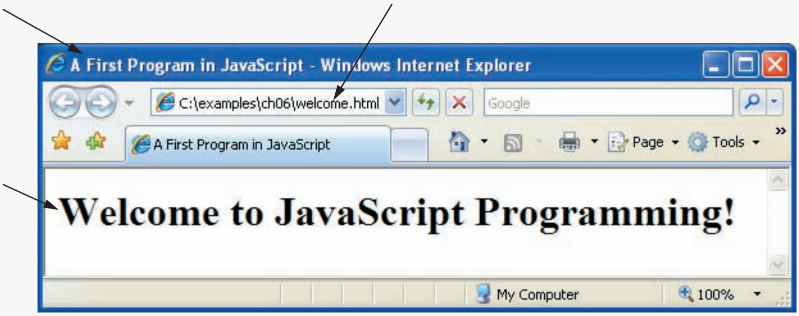
Line 8 indicates the beginning of the `<head>` section of the XHTML document. For the moment, the JavaScript code we write will appear in the `<head>` section. The browser interprets the contents of the `<head>` section first, so the JavaScript programs we write there execute before the `<body>` of the XHTML document displays. In later chapters on JavaScript and in the chapters on dynamic HTML, we illustrate inline scripting, in which JavaScript code is written in the `<body>` of an XHTML document.

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 6.2: welcome.html -->
 6  <!-- Displaying a line of text. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>A First Program in JavaScript</title>
10        <script type = "text/javascript">
11           <!--
12           document.writeln(
13              "<h1>Welcome to JavaScript Programming!</h1>" );
14           // -->
15        </script>
16     </head><body></body>
17  </html>
```

Title of the XHTML document

Location and name of the loaded XHTML document

Script result



A First Program in JavaScript - Windows Internet Explorer

C:\examples\ch06\welcome.html

Welcome to JavaScript Programming!

Fig. 6.2 | Displaying a line of text.

Line 10 uses the `<script>` tag to indicate to the browser that the text which follows is part of a script. The `type` attribute specifies the type of file as well as the scripting language used in the script—in this case, a `text` file written in `javascript`. Both Internet Explorer and Firefox use JavaScript as the default scripting language.

Line 11 contains the XHTML opening comment tag `<!--`. Some older web browsers do not support scripting. In such browsers, the actual text of a script often will display in the web page. To prevent this from happening, many script programmers enclose the script code in an XHTML comment, so that browsers that do not support scripts will simply ignore the script. The syntax used is as follows:

```
<script type = "text/javascript">
   <!--
   script code here
   // -->
</script>
```

When a browser that does not support scripts encounters the preceding code, it ignores the `<script>` and `</script>` tags and the script code in the XHTML comment. Browsers that do support scripting will interpret the JavaScript code as expected. [*Note:* Some browsers require the **JavaScript single-line comment `//`** (see Section 6.4 for an explanation) before the ending XHTML comment delimiter (`-->`) to interpret the script properly. The opening HTML comment tag (`<!--`) also serves as a single line comment delimiter in JavaScript, therefore it does not need to be commented.]

Portability Tip 6.1

Some browsers do not support the `<script>…</script>` tags. If your document is to be rendered with such browsers, enclose the script code between these tags in an XHTML comment, so that the script text does not get displayed as part of the web page. The closing comment tag of the XHTML comment (`-->`) is preceded by a JavaScript comment (`//`) to prevent the browser from trying to interpret the XHTML comment as a JavaScript statement.

Lines 12–13 instruct the browser's JavaScript interpreter to perform an **action**, namely, to display in the web page the **string** of characters contained between the **double quotation (`"`) marks**. A string is sometimes called a **character string**, a **message** or a **string literal**. We refer to characters between double quotation marks as strings. Individual white-space characters between words in a string are not ignored by the browser. However, if consecutive spaces appear in a string, browsers condense them to a single space. Also, in most cases, browsers ignore leading white-space characters (i.e., white space at the beginning of a string).

Software Engineering Observation 6.1

Strings in JavaScript can be enclosed in either double quotation marks (`"`) or single quotation marks (`'`).

Lines 12–13 use the browser's **`document` object**, which represents the XHTML document the browser is currently displaying. The `document` object allows you to specify text to display in the XHTML document. The browser contains a complete set of objects that allow script programmers to access and manipulate every element of an XHTML document. In the next several chapters, we overview some of these objects as we discuss the Document Object Model (DOM).

An object resides in the computer's memory and contains information used by the script. The term **object** normally implies that **attributes** (**data**) and **behaviors** (**methods**) are associated with the object. The object's methods use the attributes to perform useful actions for the **client of the object** (i.e., the script that calls the methods). A method may require additional information (**arguments**) to perform its action; this information is enclosed in parentheses after the name of the method in the script. In lines 12–13, we call the `document` object's **`writeln` method** to write a line of XHTML markup in the XHTML document. The parentheses following the method name `writeln` contain the one argument that method `writeln` requires (in this case, the string of XHTML that the browser is to display). Method `writeln` instructs the browser to display the argument string. If the string contains XHTML elements, the browser interprets these elements and renders them on the screen. In this example, the browser displays the phrase `Welcome to JavaScript Programming!` as an `h1`-level XHTML heading, because the phrase is enclosed in an `h1` element.

The code elements in lines 12–13, including `document.writeln`, its argument in the parentheses (the string) and the **semicolon** (`;`), together are called a **statement**. Every statement ends with a semicolon (also known as the statement terminator), although this practice is not required by JavaScript. Line 15 indicates the end of the script.

Good Programming Practice 6.1

Always include a semicolon at the end of a statement to terminate the statement. This notation clarifies where one statement ends and the next statement begins.

Common Programming Error 6.1

Forgetting the ending `</script>` tag for a script may prevent the browser from interpreting the script properly and may prevent the XHTML document from loading properly.

The `</head>` tag in line 16 indicates the end of the `<head>` section. Also in line 16, the tags `<body>` and `</body>` specify that this XHTML document has an empty body. Line 17 indicates the end of this XHTML document.

We are now ready to view our XHTML document in a web browser—open it in Internet Explorer or Firefox. If the script contains no syntax errors, it should produce the output shown in Fig. 6.2.

Common Programming Error 6.2

JavaScript is case sensitive. Not using the proper uppercase and lowercase letters is a syntax error. *A syntax error occurs when the script interpreter cannot recognize a statement. The interpreter normally issues an error message to help you locate and fix the incorrect statement. Syntax errors are violations of the rules of the programming language. The interpreter notifies you of a syntax error when it attempts to execute the statement containing the error. The JavaScript interpreter in Internet Explorer reports all syntax errors by indicating in a separate popup window that a* "runtime error" *has occurred (i.e., a problem occurred while the interpreter was running the script). [*Note: *To enable this feature in IE7, select* **Internet Options...** *from the* **Tools** *menu. In the* **Internet Options** *dialog that appears, select the* **Advanced** *tab and click the checkbox labelled* **Display a notification about every script error** *under the* **Browsing** *category. Firefox has an error console that reports JavaScript errors and warnings. It is accessible by choosing* **Error Console** *from the* **Tools** *menu.]*

Error-Prevention Tip 6.1

When the interpreter reports a syntax error, sometimes the error is not on the line number indicated by the error message. First, check the line for which the error was reported. If that line does not contain errors, check the preceding several lines in the script.

6.3 Modifying Our First Program

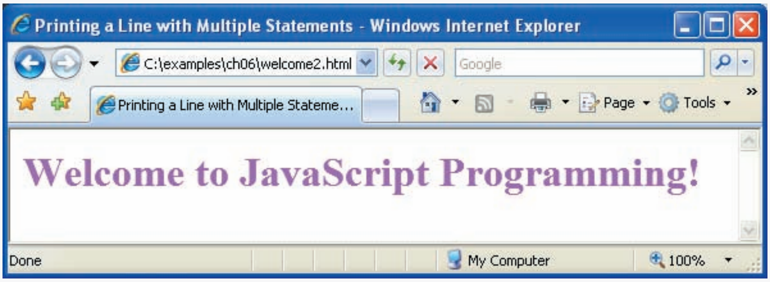
This section continues our introduction to JavaScript programming with two examples that modify the example in Fig. 6.2.

Displaying a Line of Colored Text

A script can display `Welcome to JavaScript Programming!` several ways. Figure 6.3 uses two JavaScript statements to produce one line of text in the XHTML document. This example also displays the text in a different color, using the CSS `color` property.

Most of this XHTML document is identical to Fig. 6.2, so we concentrate only on lines 12–14 of Fig. 6.3, which display one line of text in the XHTML document. The first statement uses `document` method `write` to display a string. Unlike `writeln`, `write` does

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 6.3: welcome2.html -->
 6  <!-- Printing one line with multiple statements. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Printing a Line with Multiple Statements</title>
10        <script type = "text/javascript">
11           <!--
12           document.write( "<h1 style = \"color: magenta\">" );
13           document.write( "Welcome to JavaScript " +
14              "Programming!</h1>" );
15           // -->
16        </script>
17     </head><body></body>
18  </html>
```


Printing a Line with Multiple Statements - Windows Internet Explorer
C:\examples\ch06\welcome2.html
Google
Printing a Line with Multiple Stateme...
Page
Tools
Done
My Computer
100%

Welcome to JavaScript Programming!

Fig. 6.3 | Printing one line with separate statements.

not position the output cursor in the XHTML document at the beginning of the next line after writing its argument. [*Note:* The output cursor keeps track of where the next character appears in the XHTML document, not where the next character appears in the web page as rendered by the browser.] The next character written in the XHTML document appears immediately after the last character written with `write`. Thus, when lines 13–14 execute, the first character written, "`W`," appears immediately after the last character displayed with `write` (the `>` character inside the right double quote in line 12). Each `write` or `writeln` statement resumes writing characters where the last `write` or `writeln` statement stopped writing characters. So, after a `writeln` statement, the next output appears on the beginning of the next line. In effect, the two statements in lines 12–14 result in one line of XHTML text. Remember that statements in JavaScript are separated by semicolons (`;`). Therefore, lines 13–14 represent only one complete statement. JavaScript allows large statements to be split over many lines. However, you cannot split a statement in the middle of a string. The `+` operator (called the "concatenation operator" when used in this manner) in line 13 joins two strings together and is explained in more detail later in this chapter.

Common Programming Error 6.3

Splitting a statement in the middle of a string is a syntax error.

Note that the characters \" (in line 12) are not displayed in the browser. The backslash (\) in a string is an escape character. It indicates that a “special” character is to be used in the string. When a backslash is encountered in a string of characters, the next character is combined with the backslash to form an escape sequence. The escape sequence \" is the double-quote character, which causes a double-quote character to be inserted into the string. We use this escape sequence to insert double quotes around the attribute value for `style` without terminating the string. Note that we could also have used single quotes for the attribute value, as in `document.write( "<h1 style = 'color: magenta'>" );`, because the single quotes do not terminate a double-quoted string. We discuss escape sequences in greater detail momentarily.

It is important to note that the preceding discussion has nothing to do with the actual rendering of the XHTML text. Remember that the browser does not create a new line of text unless the browser window is too narrow for the text being rendered or the browser encounters an XHTML element that explicitly starts a new line—for example, `<br/>` to start a new line or `<p>` to start a new paragraph.

Common Programming Error 6.4

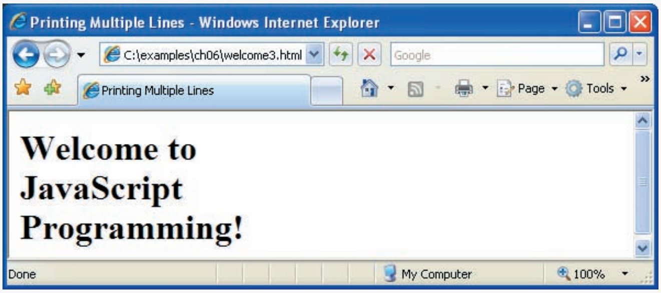
Many people confuse the writing of XHTML text with the rendering of XHTML text. Writing XHTML text creates the XHTML that will be rendered by the browser for presentation to the user.

Displaying Multiple Lines of Text

In the next example, we demonstrate that a single statement can cause the browser to display multiple lines by using line-break XHTML tags (`<br/>`) throughout the string of XHTML text in a `write` or `writeln` method call. Figure 6.4 demonstrates the use of line-break XHTML tags. Lines 12–13 produce three separate lines of text when the browser renders the XHTML document.

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 6.4: welcome3.html -->
 6  <!-- Printing on multiple lines with a single statement. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Printing Multiple Lines</title>
10        <script type = "text/javascript">
11           <!--
12           document.writeln( "<h1>Welcome to<br />JavaScript" +
13              "<br />Programming!</h1>" );
14           // -->
15        </script>
16     </head><body></body>
17  </html>
```

Fig. 6.4 | Printing on multiple lines with a single statement. (Part 1 of 2.)


Printing Multiple Lines - Windows Internet Explorer
C:\examples\ch06\welcome3.html
Google
Printing Multiple Lines
Page
Tools
Done
My Computer
100%

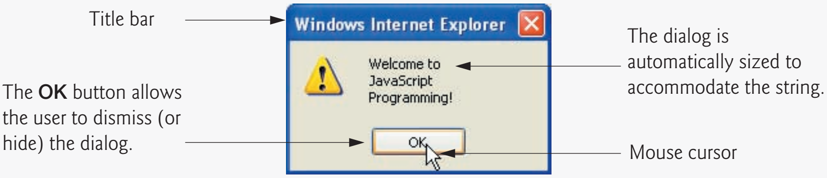
Welcome to
JavaScript
Programming!

Fig. 6.4 | Printing on multiple lines with a single statement. (Part 2 of 2.)

Displaying Text in an Alert Dialog

The first several programs in this chapter display text in the XHTML document. Sometimes it is useful to display information in windows called **dialogs** (or **dialog boxes**) that "pop up" on the screen to grab the user's attention. Dialogs typically display important messages to users browsing the web page. JavaScript allows you easily to display a dialog box containing a message. The program in Fig. 6.5 displays `Welcome to JavaScript Programming!` as three lines in a predefined dialog called an **`alert` dialog**.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 6.5: welcome4.html -->
<!-- Alert dialog displaying multiple lines. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Printing Multiple Lines in a Dialog Box</title>
      <script type = "text/javascript">
         <!--
            window.alert( "Welcome to\nJavaScript\nProgramming!" );
         // -->
      </script>
   </head>
   <body>
      <p>Click Refresh (or Reload) to run this script again.</p>
   </body>
</html>
```



Welcome to
JavaScript
Programming!

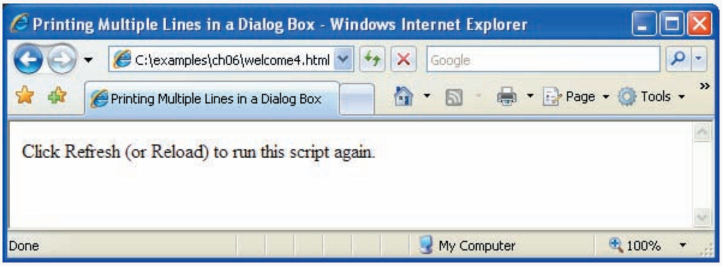
Title bar

The dialog is automatically sized to accommodate the string.

The **OK** button allows the user to dismiss (or hide) the dialog.

Mouse cursor

Fig. 6.5 | Alert dialog displaying multiple lines. (Part 1 of 2.)


Printing Multiple Lines in a Dialog Box - Windows Internet Explorer
C:\examples\ch06\welcome4.html
Google
Printing Multiple Lines in a Dialog Box
Page
Tools
Done
My Computer
100%

Click Refresh (or Reload) to run this script again.

Fig. 6.5 | Alert dialog displaying multiple lines. (Part 2 of 2.)

Line 12 in the script uses the browser's **window** object to display an alert dialog. The argument to the `window` object's **alert** method is the string to display. Executing the preceding statement displays the dialog shown in the first window of Fig. 6.5. The **title bar** of the dialog contains the string **Windows Internet Explorer** to indicate that the browser is presenting a message to the user. The dialog provides an **OK** button that allows the user to **dismiss** (i.e., **close**) the dialog by clicking the button. To dismiss the dialog, position the **mouse cursor** (also called the **mouse pointer**) over the **OK** button and click the mouse. Firefox's alert dialog looks similar, but the title bar contains the text **[JavaScript Application]**.

Common Programming Error 6.5

Dialogs display plain text; they do not render XHTML. Therefore, specifying XHTML elements as part of a string to be displayed in a dialog results in the actual characters of the tags being displayed.

Note that the `alert` dialog contains three lines of plain text. Normally, a dialog displays the characters in a string exactly as they appear between the double quotes. Note, however, that the dialog does not display the characters `\n`. The escape sequence `\n` is the **newline character**. In a dialog, the newline character causes the **cursor** (i.e., the current screen position indicator) to move to the beginning of the next line in the dialog. Some other common escape sequences are listed in Fig. 6.6. The `\n`, `\t` and `\r` escape sequences in the table do not affect XHTML rendering unless they are in a **pre element** (this element displays the text between its tags in a fixed-width font exactly as it is formatted between the tags, including leading white-space characters and consecutive white-space characters). The other escape sequences result in characters that will be displayed in plain text dialogs and in XHTML.

Escape sequence	Description
`\n`	New line. Position the screen cursor at the beginning of the next line.
`\t`	Horizontal tab. Move the screen cursor to the next tab stop.

Fig. 6.6 | Some common escape sequences. (Part 1 of 2.)

Escape sequence	Description
`\r`	Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line. Any characters output after the carriage return overwrite the characters previously output on that line.
`\\`	Backslash. Used to represent a backslash character in a string.
`\"`	Double quote. Used to represent a double-quote character in a string contained in double quotes. For example, `window.alert( "\"in quotes\"" );` displays `"in quotes"` in an `alert` dialog.
`\'`	Single quote. Used to represent a single-quote character in a string. For example, `window.alert( '\'in quotes\'' );` displays `'in quotes'` in an `alert` dialog.

Fig. 6.6 | Some common escape sequences. (Part 2 of 2.)

Common Programming Error 6.6

XHTML elements in an alert dialog's message are not interpreted as XHTML. This means that using `<br />`, for example, to create a line break in an alert box is an error. The string `<br/>` will simply be included in your message.

6.4 Obtaining User Input with `prompt` Dialogs

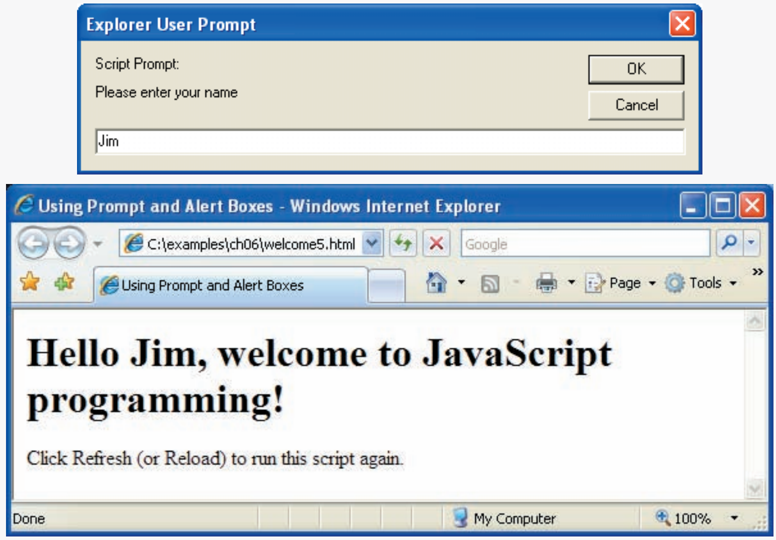
Scripting gives you the ability to generate part or all of a web page's content at the time it is shown to the user. A script can adapt the content based on input from the user or other variables, such as the time of day or the type of browser used by the client. Such web pages are said to be **dynamic**, as opposed to static, since their content has the ability to change. The next two subsections use scripts to demonstrate dynamic web pages.

6.4.1 Dynamic Welcome Page

Our next script builds on prior scripts to create a dynamic welcome page that obtains the user's name, then displays it on the page. The script uses another predefined dialog box from the `window` object—a **`prompt`** dialog—which allows the user to input a value that the script can use. The program asks the user to input a name, then displays the name in the XHTML document. Figure 6.7 presents the script and sample output. [*Note:* In later JavaScript chapters, we obtain input via GUI components in XHTML forms, as introduced in Chapter 4.]

Line 12 is a **declaration** that contains the JavaScript **keyword** `var`. Keywords are words that have special meaning in JavaScript. The keyword **`var`** at the beginning of the statement indicates that the word `name` is a **variable**. A variable is a location in the computer's memory where a value can be stored for use by a program. All variables have a name, type and value, and should be declared with a `var` statement before they are used in

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 6.7: welcome5.html -->
<!-- Prompt box used on a welcome screen. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Using Prompt and Alert Boxes</title>
      <script type = "text/javascript">
         <!--
         var name; // string entered by the user

         // read the name from the prompt box as a string
         name = window.prompt( "Please enter your name" );

         document.writeln( "<h1>Hello, " + name +
            ", welcome to JavaScript programming!</h1>" );
         // -->
      </script>
   </head>
   <body>
      <p>Click Refresh (or Reload) to run this script again.</p>
   </body>
</html>
```



Explorer User Prompt

Script Prompt:

Please enter your name

Jim

Using Prompt and Alert Boxes - Windows Internet Explorer

C:\examples\ch06\welcome5.html

Hello Jim, welcome to JavaScript programming!

Click Refresh (or Reload) to run this script again.

Fig. 6.7 | Prompt box used on a welcome screen.

a program. Although using var to declare variables is not required, we will see in Chapter 9, JavaScript: Functions, that var sometimes ensures proper behavior of a script.

The name of a variable can be any valid identifier. An identifier is a series of characters consisting of letters, digits, underscores (_) and dollar signs ($) that does not begin with a digit and is not a reserved JavaScript keyword. [*Note:* A complete list of keywords can be found in Fig. 7.2.] Identifiers may not contain spaces. Some valid identifiers are `Welcome`, `$value`, `_value`, `m_inputField1` and `button7`. The name `7button` is not a valid identifier, because it begins with a digit, and the name `input field` is not valid, because it contains a space. Remember that JavaScript is **case sensitive**—uppercase and lowercase letters are considered to be different characters, so `name`, `Name` and `NAME` are different identifiers.

Good Programming Practice 6.2

Choosing meaningful variable names helps a script to be "self-documenting" (i.e., easy to understand by simply reading the script, rather than having to read manuals or extended comments).

Good Programming Practice 6.3

By convention, variable-name identifiers begin with a lowercase first letter. Each subsequent word should begin with a capital first letter. For example, identifier `itemPrice` *has a capital* `P` *in its second word,* `Price`*.*

Common Programming Error 6.7

Splitting a statement in the middle of an identifier is a syntax error.

Declarations end with a semicolon (`;`) and can be split over several lines with each variable in the declaration separated by a comma—known as a **comma-separated list** of variable names. Several variables may be declared either in one declaration or in multiple declarations.

Programmers often indicate the purpose of each variable in the program by placing a JavaScript comment at the end of each line in the declaration. In line 12, a **single-line comment** that begins with the characters `//` states the purpose of the variable in the script. This form of comment is called a single-line comment because it terminates at the end of the line in which it appears. A `//` comment can begin at any position in a line of JavaScript code and continues until the end of the line. Comments do not cause the browser to perform any action when the script is interpreted; rather, comments are ignored by the JavaScript interpreter.

Good Programming Practice 6.4

Some programmers prefer to declare each variable on a separate line. This format allows for easy insertion of a descriptive comment next to each declaration. This is a widely followed professional coding standard.

Another comment notation facilitates the writing of **multiline comments**. For example,

```
/* This is a multiline
   comment. It can be
   split over many lines. */
```

is a multiline comment spread over several lines. Such comments begin with the delimiter `/*` and end with the delimiter `*/`. All text between the delimiters of the comment is ignored by the interpreter.

Common Programming Error 6.8

Forgetting one of the delimiters of a multiline comment is a syntax error.

Common Programming Error 6.9

Nesting multiline comments (i.e., placing a multiline comment between the delimiters of another multiline comment) is a syntax error.

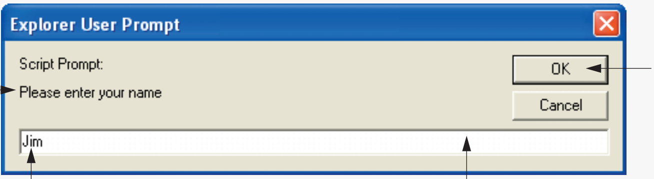
JavaScript adopted comments delimited with `/*` and `*/` from the C programming language and single-line comments delimited with `//` from the C++ programming language. JavaScript programmers generally prefer C++-style single-line comments over C-style comments. Throughout this book, we use C++-style single-line comments.

Line 14 is a comment indicating the purpose of the statement in the next line. Line 15 calls the `window` object's `prompt` method, which displays the dialog in Fig. 6.8. The dialog allows the user to enter a string representing the user's name.

The argument to `prompt` specifies a message telling the user what to type in the text field. This message is called a prompt because it directs the user to take a specific action. An optional second argument, separated from the first by a comma, may specify the default string displayed in the text field; our code does not supply a second argument. In this case, Internet Explorer displays the default value `undefined`, while Firefox and most other browsers leave the text field empty. The user types characters in the text field, then clicks the **OK** button to submit the string to the program. We normally receive input from a user through a GUI component such as the `prompt` dialog, as in this program, or through an XHTML form GUI component, as we will see in later chapters.

The user can type anything in the text field of the `prompt` dialog. For this program, whatever the user enters is considered the name. If the user clicks the **Cancel** button, no string value is sent to the program. Instead, the `prompt` dialog submits the value `null`, a JavaScript keyword signifying that a variable has no value. Note that `null` is not a string literal, but rather a predefined term indicating the absence of value. Writing a `null` value to the document, however, displays the word `null` in the web page.

The statement in line 15 assigns the value returned by the `window` object's `prompt` method (a string containing the characters typed by the user—or the default value or `null` if the **Cancel** button is clicked) to variable `name` by using the assignment operator, `=`. The statement is read as, "`name` gets the value returned by `window.prompt( "Please enter your name" )`." The `=` operator is called a binary operator because it has two operands—`name` and the result of the expression `window.prompt( "Please enter your name" )`. This entire statement is called an assignment statement because it assigns a value to a variable. The expression to the right of the assignment operator is always evaluated first.



This is the prompt to the user.

When the user clicks **OK**, the value typed by the user is returned to the program as a string.

This is the value the user types into the alert dialog.

This is the text field in which the user types the value.

Fig. 6.8 | Prompt dialog displayed by the `window` object's `prompt` method.

Good Programming Practice 6.5

Place spaces on either side of a binary operator. This format makes the operator stand out and makes the program more readable.

Lines 17–18 use `document.writeln` to display the new welcome message. The expression inside the parentheses uses the operator + to “add” a string (the literal `"<h1>Hello, "`), the variable `name` (the string that the user entered in line 15) and another string (the literal `", welcome to JavaScript programming!</h1>"`). JavaScript has a version of the + operator for **string concatenation** that enables a string and a value of another data type (including another string) to be combined. The result of this operation is a new (and normally longer) string. If we assume that `name` contains the string literal `"Jim"`, the expression evaluates as follows: JavaScript determines that the two operands of the first + operator (the string `"<h1>Hello, "` and the value of variable `name`) are both strings, then concatenates the two into one string. Next, JavaScript determines that the two operands of the second + operator (the result of the first concatenation operation, the string `"<h1>Hello, Jim"`, and the string `", welcome to JavaScript programming!</h1>"`) are both strings and concatenates the two. This results in the string `"<h1>Hello, Jim, welcome to JavaScript programming!</h1>"`. The browser renders this string as part of the XHTML document. Note that the space between `Hello,` and `Jim` is part of the string `"<h1>Hello, "`.

As we’ll illustrate later, the + operator used for string concatenation can convert other variable types to strings if necessary. Because string concatenation occurs between two strings, JavaScript must convert other variable types to strings before it can proceed with the operation. For example, if a variable `age` has an integer value equal to `21`, then the expression `"my age is " + age` evaluates to the string `"my age is 21"`. JavaScript converts the value of `age` to a string and concatenates it with the existing string literal `"my age is "`.

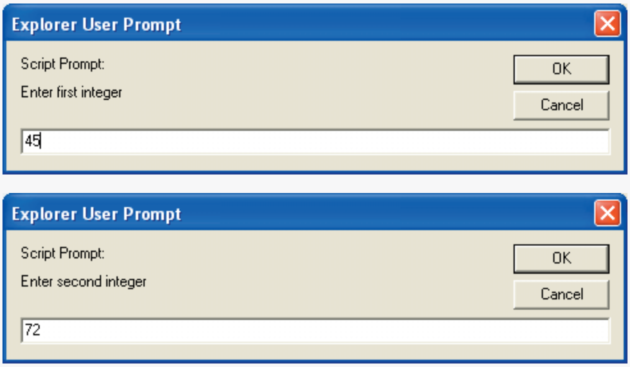
After the browser interprets the `<head>` section of the XHTML document (which contains the JavaScript), it then interprets the `<body>` of the XHTML document (lines 22–24) and renders the XHTML. Notice that the XHTML page is not rendered until the prompt is dismissed because the prompt pauses execution in the `head`, before the `body` is processed. If you click your browser’s **Refresh** (Internet Explorer) or **Reload** (Firefox) button after entering a name, the browser will reload the XHTML document, so that you can execute the script again and change the name. [*Note:* In some cases, it may be necessary to hold down the *Shift* key while clicking the **Refresh** or **Reload** button, to ensure that the XHTML document reloads properly. Browsers often save a recent copy of a page in memory, and holding the *Shift* key forces the browser to download the most recent version of a page.]

6.4.2 Adding Integers

Our next script illustrates another use of `prompt` dialogs to obtain input from the user. Figure 6.9 inputs two **integers** (whole numbers, such as 7, –11, 0 and 31914) typed by a user at the keyboard, computes the sum of the values and displays the result.

Lines 12–16 declare the variables `firstNumber`, `secondNumber`, `number1`, `number2` and `sum`. Single-line comments state the purpose of each of these variables. Line 19 employs a `prompt` dialog to allow the user to enter a string representing the first of the two

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 6.9: addition.html -->
<!-- Addition script. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>An Addition Program</title>
      <script type = "text/javascript">
         <!--
         var firstNumber; // first string entered by user
         var secondNumber; // second string entered by user
         var number1; // first number to add
         var number2; // second number to add
         var sum; // sum of number1 and number2

         // read in first number from user as a string
         firstNumber = window.prompt( "Enter first integer" );

         // read in second number from user as a string
         secondNumber = window.prompt( "Enter second integer" );

         // convert numbers from strings to integers
         number1 = parseInt( firstNumber );
         number2 = parseInt( secondNumber );

         sum = number1 + number2; // add the numbers

         // display the results
         document.writeln( "<h1>The sum is " + sum + "</h1>" );
         // -->
      </script>
   </head>
   <body>
      <p>Click Refresh (or Reload) to run the script again</p>
   </body>
</html>
```



Explorer User Prompt

Script Prompt:

Enter first integer

45

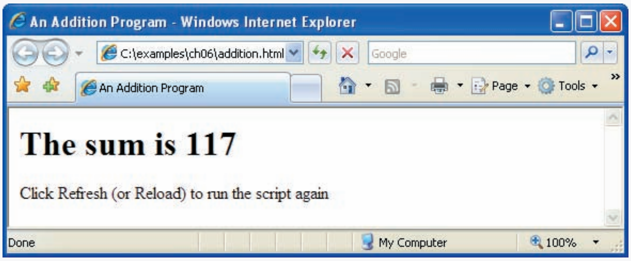
Explorer User Prompt

Script Prompt:

Enter second integer

72

Fig. 6.9 | Addition script. (Part 1 of 2.)


An Addition Program - Windows Internet Explorer
C:\examples\ch06\addition.html
Google
An Addition Program
Page
Tools
Done
My Computer
100%

The sum is 117

Click Refresh (or Reload) to run the script again

Fig. 6.9 | Addition script. (Part 2 of 2.)

integers that will be added. The script assigns the first value entered by the user to the variable `firstNumber`. Line 22 displays a `prompt` dialog to obtain the second number to add and assign this value to the variable `secondNumber`.

As in the preceding example, the user can type anything in the `prompt` dialog. For this program, if the user either types a noninteger value or clicks the **Cancel** button, a logic error will occur, and the sum of the two values will appear in the XHTML document as **NaN** (meaning **not a number**). A logic error is caused by syntactically correct code that produces an undesired result. In Chapter 11, JavaScript: Objects, we discuss the `Number` object and its methods that can determine whether a value is not a number.

Recall that a `prompt` dialog returns to the program as a string the value typed by the user. Lines 25–26 convert the two strings input by the user to integer values that can be used in a calculation. Function **`parseInt`** converts its string argument to an integer. Line 25 assigns to the variable `number1` the integer that function `parseInt` returns. Line 26 assigns an integer value to variable `number2` in a similar manner. Any subsequent references to `number1` and `number2` in the program use these integer values. [*Note:* We refer to `parseInt` as a **function** rather than a method because we do not precede the function call with an object name (such as `document` or `window`) and a dot (`.`). The term method means that the function belongs to a particular object. For example, method `writeln` belongs to the `document` object and method `prompt` belongs to the `window` object.]

Line 28 calculates the sum of the variables `number1` and `number2` using the **addition operator**, `+`, and assigns the result to variable `sum` by using the assignment operator, `=`. Notice that the `+` operator can perform both addition and string concatenation. In this case, the `+` operator performs addition, because both operands contain integers. After line 28 performs this calculation, line 31 uses `document.writeln` to display the result of the addition on the web page. Lines 33 and 34 close the `script` and `head` elements, respectively. Lines 35–37 render the `body` of XHTML document. Use your browser's **Refresh** or **Reload** button to reload the XHTML document and run the script again.

Common Programming Error 6.10

Confusing the + operator used for string concatenation with the + operator used for addition often leads to undesired results. For example, if integer variable `y` *has the value* `5`*, the expression* `"y + 2 = " + y + 2` *results in* `"y + 2 = 52"`*, not* `"y + 2 = 7"`*, because first the value of* `y` *(i.e.,* `5`*) is concatenated with the string* `"y + 2 = "`*, then the value* `2` *is concatenated with the new, larger string* `"y + 2 = 5"`*. The expression* `"y + 2 = " + (y + 2)` *produces the string* `"y + 2 = 7"` *because the parentheses ensure that* `y + 2` *is executed mathematically before it is conveted to a string.*

6.5 Memory Concepts

Variable names such as number1, number2 and sum actually correspond to locations in the computer's memory. Every variable has a name, a type and a value.

In the addition program in Fig. 6.9, when line 25 executes, the string firstNumber (previously entered by the user in a prompt dialog) is converted to an integer and placed into a memory location to which the name number1 has been assigned by the interpreter. Suppose the user entered the string 45 as the value for firstNumber. The program converts firstNumber to an integer, and the computer places the integer value 45 into location number1, as shown in Fig. 6.10. Whenever a value is placed in a memory location, the value replaces the previous value in that location. The previous value is lost.

Suppose that the user enters 72 as the second integer. When line 26 executes, the program converts secondNumber to an integer and places that integer value, 72, into location number2; then the memory appears as shown in Fig. 6.11.

Once the program has obtained values for number1 and number2, it adds the values and places the sum into variable sum. The statement

```
sum = number1 + number2;
```

performs the addition and also replaces sum's previous value. After sum is calculated, the memory appears as shown in Fig. 6.12. Note that the values of number1 and number2 appear exactly as they did before they were used in the calculation of sum. These values were used, but not destroyed, when the computer performed the calculation—when a value is read from a memory location, the process is nondestructive.

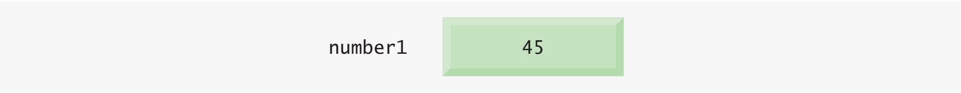


Fig. 6.10 | Memory location showing the name and value of variable number1.

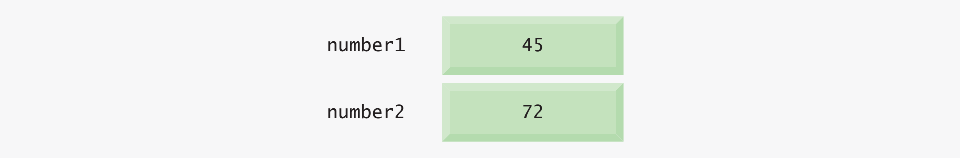


Fig. 6.11 | Memory locations after inputting values for variables number1 and number2.

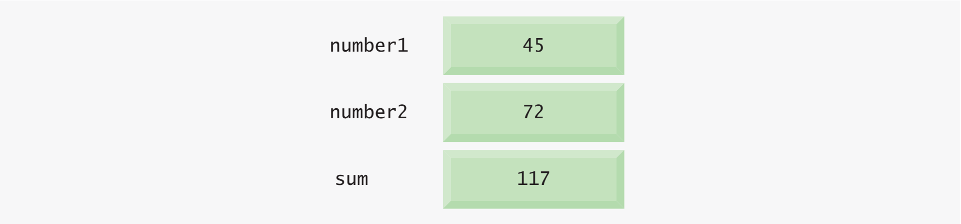


Fig. 6.12 | Memory locations after calculating the sum of number1 and number2.

Data Types in JavaScript

Unlike its predecessor languages C, C++ and Java, JavaScript does not require variables to have a declared type before they can be used in a program. A variable in JavaScript can contain a value of any data type, and in many situations JavaScript automatically converts between values of different types for you. For this reason, JavaScript is referred to as a **loosely typed language**. When a variable is declared in JavaScript, but is not given a value, the variable has an **undefined** value. Attempting to use the value of such a variable is normally a logic error.

When variables are declared, they are not assigned values unless specified by the programmer. Assigning the value `null` to a variable indicates that it does not contain a value.

6.6 Arithmetic

Many scripts perform arithmetic calculations. Figure 6.13 summarizes the **arithmetic operators**. Note the use of various special symbols not used in algebra. The **asterisk (*)** indicates multiplication; the **percent sign (%)** is the **remainder operator**, which will be discussed shortly. The arithmetic operators in Fig. 6.13 are binary operators, because each operates on two operands. For example, the expression `sum + value` contains the binary operator + and the two operands `sum` and `value`.

JavaScript provides the remainder operator, `%`, which yields the remainder after division. [*Note:* The `%` operator is known as the modulus operator in some programming languages.] The expression `x % y` yields the remainder after `x` is divided by `y`. Thus, `17 % 5` yields `2` (i.e., 17 divided by 5 is 3, with a remainder of 2), and `7.4 % 3.1` yields `1.2`. In later chapters, we consider applications of the remainder operator, such as determining whether one number is a multiple of another. There is no arithmetic operator for exponentiation in JavaScript. (Chapter 8, JavaScript: Control Statements II, shows how to perform exponentiation in JavaScript using the `Math` object's `pow` method.)

Arithmetic expressions in JavaScript must be written in **straight-line form** to facilitate entering programs into the computer. Thus, expressions such as "`a` divided by `b`" must be written as `a / b`, so that all constants, variables and operators appear in a straight line. The following algebraic notation is generally not acceptable to computers:

$$\frac{a}{b}$$

JavaScript operation	Arithmetic operator	Algebraic expression	JavaScript expression
Addition	+	$f + 7$	`f + 7`
Subtraction	-	$p - c$	`p - c`
Multiplication	*	bm	`b * m`
Division	/	x / y *or* $\frac{x}{y}$ *or* $x \div y$	`x / y`
Remainder	%	$r \bmod s$	`r % s`

Fig. 6.13 | Arithmetic operators.

Parentheses are used to group expressions in the same manner as in algebraic expressions. For example, to multiply a times the quantity b + c we write:

```
a * ( b + c )
```

JavaScript applies the operators in arithmetic expressions in a precise sequence determined by the following **rules of operator precedence**, which are generally the same as those followed in algebra:

1. Multiplication, division and remainder operations are applied first. If an expression contains several multiplication, division and remainder operations, operators are applied from left to right. Multiplication, division and remainder operations are said to have the same level of precedence.
2. Addition and subtraction operations are applied next. If an expression contains several addition and subtraction operations, operators are applied from left to right. Addition and subtraction operations have the same level of precedence.

The rules of operator precedence enable JavaScript to apply operators in the correct order. When we say that operators are applied from left to right, we are referring to the **associativity** of the operators—the order in which operators of equal priority are evaluated. We will see that some operators associate from right to left. Figure 6.14 summarizes the rules of operator precedence. The table in Fig. 6.14 will be expanded as additional JavaScript operators are introduced. A complete precedence chart is included in Appendix C.

Now, in light of the rules of operator precedence, let us consider several algebraic expressions. Each example lists an algebraic expression and the equivalent JavaScript expression.

The following is an example of an arithmetic mean (average) of five terms:

Algebra: $m = \dfrac{a+b+c+d+e}{5}$

JavaScript: `m = ( a + b + c + d + e ) / 5;`

The parentheses are required to group the addition operators, because division has higher precedence than addition. The entire quantity `( a + b + c + d + e )` is to be divided by `5`. If the parentheses are erroneously omitted, we obtain `a + b + c + d + e / 5`, which evaluates as

$$a + b + c + d + \frac{e}{5}$$

and would not lead to the correct answer.

Operator(s)	Operation(s)	Order of evaluation (precedence)
*, / or %	Multiplication Division Remainder	Evaluated first. If there are several such operations, they are evaluated from left to right.
+ or -	Addition Subtraction	Evaluated last. If there are several such operations, they are evaluated from left to right.

Fig. 6.14 | Precedence of arithmetic operators.

The following is an example of the equation of a straight line:

Algebra: $y = mx + b$

JavaScript: `y = m * x + b;`

No parentheses are required. The multiplication operator is applied first, because multiplication has a higher precedence than addition. The assignment occurs last, because it has a lower precedence than multiplication and addition.

The following example contains remainder (%), multiplication, division, addition and subtraction operations:

Algebra: $z = pr \% q + w/x - y$

Java:

```
z = p * r % q + w / x - y;
  ⑥   ①   ②   ④   ③   ⑤
```

The circled numbers under the statement indicate the order in which JavaScript applies the operators. The multiplication, remainder and division operations are evaluated first in left-to-right order (i.e., they associate from left to right), because they have higher precedence than addition and subtraction. The addition and subtraction operations are evaluated next. These operations are also applied from left to right.

To develop a better understanding of the rules of operator precedence, consider the evaluation of a second-degree polynomial ($y = ax^2 + bx + c$):

```
y = a * x * x + b * x + c;
  ⑥   ①   ②   ④   ③   ⑤
```

The circled numbers indicate the order in which JavaScript applies the operators.

Suppose that `a`, `b`, `c` and `x` are initialized as follows: `a = 2`, `b = 3`, `c = 7` and `x = 5`. Figure 6.15 illustrates the order in which the operators are applied in the preceding second-degree polynomial.

As in algebra, it is acceptable to use unnecessary parentheses in an expression to make the expression clearer. These are also called **redundant parentheses**. For example, the preceding second-degree polynomial might be parenthesized as follows:

```
y = ( a * x * x ) + ( b * x ) + c;
```

Good Programming Practice 6.6

Using parentheses for complex arithmetic expressions, even when the parentheses are not necessary, can make the arithmetic expressions easier to read.

6.7 Decision Making: Equality and Relational Operators

This section introduces a version of JavaScript's **`if`** statement that allows a program to make a decision based on the truth or falsity of a **condition**. If the condition is met (i.e., the condition is **true**), the statement in the body of the `if` statement is executed. If the condition is not met (i.e., the condition is **false**), the statement in the body of the `if` state-

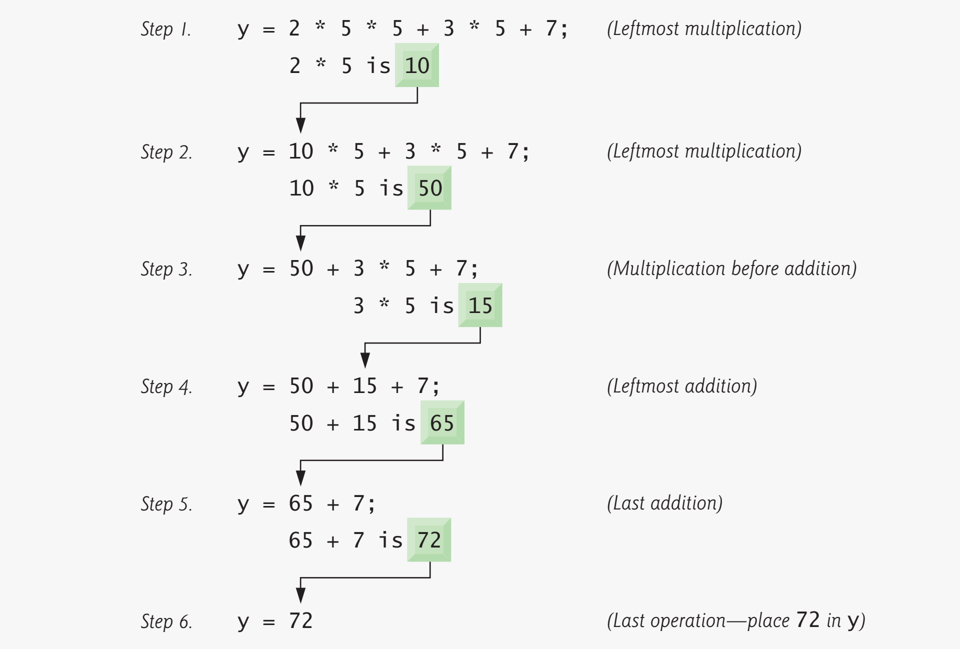

Step 1. y = 2 * 5 * 5 + 3 * 5 + 7; (Leftmost multiplication)
2 * 5 is 10
Step 2. y = 10 * 5 + 3 * 5 + 7; (Leftmost multiplication)
10 * 5 is 50
Step 3. y = 50 + 3 * 5 + 7; (Multiplication before addition)
3 * 5 is 15
Step 4. y = 50 + 15 + 7; (Leftmost addition)
50 + 15 is 65
Step 5. y = 65 + 7; (Last addition)
65 + 7 is 72
Step 6. y = 72 (Last operation—place 72 in y)

Fig. 6.15 | Order in which a second-degree polynomial is evaluated.

ment is not executed. We will see an example shortly. [*Note:* Other versions of the `if` statement are introduced in Chapter 7, JavaScript: Control Statements I.]

Conditions in `if` statements can be formed by using the **equality operators** and **relational operators** summarized in Fig. 6.16. The relational operators all have the same level

Standard algebraic equality operator or relational operator	JavaScript equality or relational operator	Sample JavaScript condition	Meaning of JavaScript condition
Equality operators			
=	`==`	`x == y`	`x` is equal to `y`
≠	`!=`	`x != y`	`x` is not equal to `y`
Relational operators			
>	`>`	`x > y`	`x` is greater than `y`
<	`<`	`x < y`	`x` is less than `y`
≥	`>=`	`x >= y`	`x` is greater than or equal to `y`
≤	`<=`	`x <= y`	`x` is less than or equal to `y`

Fig. 6.16 | Equality and relational operators.

of precedence and associate from left to right. The equality operators both have the same level of precedence, which is lower than the precedence of the relational operators. The equality operators also associate from left to right.

Common Programming Error 6.11

It is a syntax error if the operators ==, !=, >= and <= contain spaces between their symbols, as in = =, ! =, > = and < =, respectively.

Common Programming Error 6.12

Reversing the operators !=, >= and <=, as in =!, => and =<, respectively, is a syntax error.

Common Programming Error 6.13

Confusing the equality operator, ==, with the assignment operator, =, is a logic error. The equality operator should be read as "is equal to," and the assignment operator should be read as "gets" or "gets the value of." Some people prefer to read the equality operator as "double equals" or "equals equals."

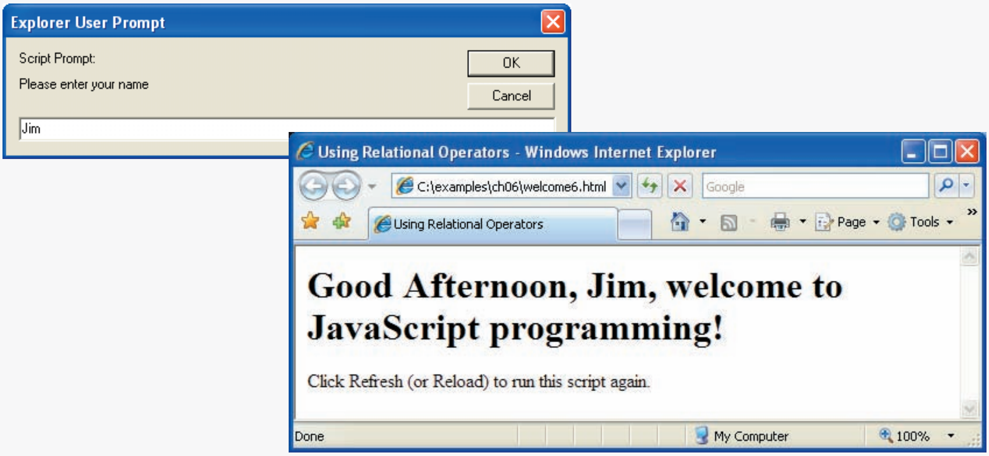
The script in Fig. 6.17 uses four `if` statements to display a time-sensitive greeting on a welcome page. The script obtains the local time from the user's computer and converts it from 24-hour clock format (0–23) to a 12-hour clock format (0–11). Using this value, the script displays an appropriate greeting for the current time of day. The script and sample output are shown in Fig. 6.17.

Lines 12–14 declare the variables used in the script. Remember that variables may be declared in one declaration or in multiple declarations. If more than one variable is declared in a single declaration (as in this example), the names are separated by commas (,). This list of names is referred to as a comma-separated list. Once again, note the comment at the end of each line, indicating the purpose of each variable in the program. Also note that some of the variables are assigned a value in the declaration—JavaScript allows you to assign a value to a variable when the variable is declared.

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 6.17: welcome6.html -->
 6  <!-- Using equality and relational operators. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Using Relational Operators</title>
10        <script type = "text/javascript">
11           <!--
12           var name; // string entered by the user
13           var now = new Date();      // current date and time
14           var hour = now.getHours(); // current hour (0-23)
15
16           // read the name from the prompt box as a string
17           name = window.prompt( "Please enter your name" );
```

Fig. 6.17 | Using equality and relational operators. (Part 1 of 2.)

```
18
19             // determine whether it is morning
20             if ( hour < 12 )
21                document.write( "<h1>Good Morning, " );
22
23             // determine whether the time is PM
24             if ( hour >= 12 )
25             {
26                // convert to a 12-hour clock
27                hour = hour - 12;
28
29                // determine whether it is before 6 PM
30                if ( hour < 6 )
31                   document.write( "<h1>Good Afternoon, " );
32
33                // determine whether it is after 6 PM
34                if ( hour >= 6 )
35                   document.write( "<h1>Good Evening, " );
36             } // end if
37
38             document.writeln( name +
39                ", welcome to JavaScript programming!</h1>" );
40             // -->
41          </script>
42       </head>
43       <body>
44          <p>Click Refresh (or Reload) to run this script again.</p>
45       </body>
46    </html>
```


Explorer User Prompt
OK
Cancel
Using Relational Operators - Windows Internet Explorer
C:\examples\ch06\welcome6.html
Google
Using Relational Operators
Page
Tools
Done
My Computer
100%

Script Prompt:
Please enter your name

Jim

Good Afternoon, Jim, welcome to JavaScript programming!

Click Refresh (or Reload) to run this script again.

Fig. 6.17 | Using equality and relational operators. (Part 2 of 2.)

Line 13 sets the variable now to a new **Date object**, which contains information about the current local time. In Section 6.2, we introduced the document object, an object that encapsulates data pertaining to the current web page. Programmers may choose to use other objects to perform specific tasks or obtain particular pieces of information. Here, we use JavaScript's built-in `Date` object to acquire the current local time. We create a new instance of an object by using the **`new`** operator followed by the type of the object, `Date`, and a pair of parentheses. Some objects require that arguments be placed in the parentheses to specify details about the object to be created. In this case, we leave the parentheses empty to create a default `Date` object containing information about the current date and time. After line 13 executes, the variable `now` refers to the new `Date` object. [*Note:* We did not need to use the `new` operator when we used the `document` and `window` objects because these objects always are created by the browser.] Line 14 sets the variable `hour` to an integer equal to the current hour (in a 24-hour clock format) returned by the `Date` object's `getHours` method. Chapter 11 presents a more detailed discussion of the `Date` object's attributes and methods, and of objects in general. As in the preceding example, the script uses `window.prompt` to allow the user to enter a name to display as part of the greeting (line 17).

To display the correct time-sensitive greeting, the script must determine whether the user is visiting the page during the morning, afternoon or evening. The first `if` statement (lines 20–21) compares the value of variable `hour` with `12`. If `hour` is less than `12`, then the user is visiting the page during the morning, and the statement at line 21 outputs the string `"Good morning"`. If this condition is not met, line 21 is not executed. Line 24 determines whether `hour` is greater than or equal to `12`. If `hour` is greater than or equal to `12`, then the user is visiting the page in either the afternoon or the evening. Lines 25–36 execute to determine the appropriate greeting. If `hour` is less than `12`, then the JavaScript interpreter does not execute these lines and continues to line 38.

The brace `{` in line 25 begins a block of statements (lines 27–35) that are all executed together if `hour` is greater than or equal to `12`—to execute multiple statements inside an `if` construct, enclose them in curly braces. Line 27 subtracts `12` from `hour`, converting the current hour from a 24-hour clock format (0–23) to a 12-hour clock format (0–11). The `if` statement (line 30) determines whether `hour` is now less than `6`. If it is, then the time is between noon and 6 PM, and line 31 outputs the beginning of an XHTML `h1` element (`"<h1>Good Afternoon, "`). If `hour` is greater than or equal to `6`, the time is between 6 PM and midnight, and the script outputs the greeting `"Good Evening"` (lines 34–35). The brace `}` in line 36 ends the block of statements associated with the `if` statement in line 24. Note that `if` statements can be **nested**, i.e., one `if` statement can be placed inside another `if` statement. The `if` statements that determine whether the user is visiting the page in the afternoon or the evening (lines 30–31 and lines 34–35) execute only if the script has already established that `hour` is greater than or equal to `12` (line 24). If the script has already determined the current time of day to be morning, these additional comparisons are not performed. (Chapter 7, JavaScript: Control Statements I, presents a more in-depth discussion of blocks and nested `if` statements.) Finally, lines 38–39 output the rest of the XHTML `h1` element (the remaining part of the greeting), which does not depend on the time of day.

Good Programming Practice 6.7

Include comments after the closing curly brace of control statements (such as `if` statements) to indicate where the statements end, as in line 36 of Fig. 6.17.

Note the indentation of the `if` statements throughout the program. Such indentation enhances program readability.

Good Programming Practice 6.8

Indent the statement in the body of an `if` *statement to make the body of the statement stand out and to enhance program readability.*

Good Programming Practice 6.9

Place only one statement per line in a program. This enhances program readability.

Common Programming Error 6.14

Forgetting the left and/or right parentheses for the condition in an `if` *statement is a syntax error. The parentheses are required.*

Note that there is no semicolon (`;`) at the end of the first line of each `if` statement. Including such a semicolon would result in a logic error at execution time. For example,

```
if ( hour < 12 ) ;
   document.write( "<h1>Good Morning, " );
```

would actually be interpreted by JavaScript erroneously as

```
if ( hour < 12 )
   ;

document.write( "<h1>Good Morning, " );
```

where the semicolon on the line by itself—called the **empty statement**—is the statement to execute if the condition in the `if` statement is true. When the empty statement executes, no task is performed in the program. The program then continues with the next statement, which executes regardless of whether the condition is true or false. In this example, `"<h1>Good Morning, "` would be printed regardless of the time of day.

Common Programming Error 6.15

Placing a semicolon immediately after the right parenthesis of the condition in an `if` *statement is normally a logic error. The semicolon would cause the body of the* `if` *statement to be empty, so the* `if` *statement itself would perform no action, regardless of whether its condition was true. Worse yet, the intended body statement of the* `if` *statement would now become a statement in sequence after the* `if` *statement and would always be executed.*

Common Programming Error 6.16

Leaving out a condition in a series of `if` *statements is normally a logic error. For instance, checking if* `hour` *is greater than* `12` *or less than* `12`*, but not if* `hour` *is equal to* `12`*, would mean that the script takes no action when* `hour` *is equal to* `12`*. Always be sure to handle every possible condition.*

Note the use of spacing in lines 38–39 of Fig. 6.17. Remember that white-space characters, such as tabs, newlines and spaces, are normally ignored by the browser. So, statements may be split over several lines and may be spaced according to the programmer's preferences without affecting the meaning of a program. However, it is incorrect to split identifiers and string literals. Ideally, statements should be kept small, but it is not always possible to do so.

Good Programming Practice 6.10

A lengthy statement may be spread over several lines. If a single statement must be split across lines, choose breaking points that make sense, such as after a comma in a comma-separated list or after an operator in a lengthy expression. If a statement is split across two or more lines, indent all subsequent lines.

The chart in Fig. 6.18 shows the precedence of the operators introduced in this chapter. The operators are shown from top to bottom in decreasing order of precedence. Note that all of these operators, with the exception of the assignment operator, `=`, associate from left to right. Addition is left associative, so an expression like `x + y + z` is evaluated as if it had been written as `(x + y) + z`. The assignment operator, `=`, associates from right to left, so an expression like `x = y = 0` is evaluated as if it had been written as `x = (y = 0)`, which first assigns the value `0` to variable `y`, then assigns the result of that assignment, `0`, to `x`.

Good Programming Practice 6.11

Refer to the operator precedence chart when writing expressions containing many operators. Confirm that the operations are performed in the order in which you expect them to be performed. If you are uncertain about the order of evaluation in a complex expression, use parentheses to force the order, exactly as you would do in algebraic expressions. Be sure to observe that some operators, such as assignment (=), associate from right to left rather than from left to right.

Operators	Associativity	Type
`*  /  %`	left to right	multiplicative
`+  -`	left to right	additive
`<  <=  >  >=`	left to right	relational
`==  !=`	left to right	equality
`=`	right to left	assignment

Fig. 6.18 | Precedence and associativity of the operators discussed so far.

6.8 Wrap-Up

We've introduced many important features of JavaScript, including how to display data, how to input data from the keyboard, how to perform calculations and how to make decisions. We also discussed the basics of memory and how variables are stored in a computer. In Chapter 7, we introduce structured programming. You will become more familiar with indentation techniques. We will study how to specify and vary the order in which statements are executed; this order is called the flow of control.

6.9 Web Resources

`www.deitel.com/javascript`

The Deitel JavaScript Resource Center contains links to some of the best JavaScript resources on the web. There you'll find categorized links to JavaScript tools, code generators, forums, books, libraries, frameworks and more. Also check out the tutorials for all skill levels, from introductory to advanced.

Summary

Section 6.1 Introduction

- The JavaScript language facilitates a disciplined approach to the design of computer programs that enhance web pages.

Section 6.2 Simple Program: Displaying a Line of Text in a Web Page

- The spacing displayed by a browser in a web page is determined by the XHTML elements used to format the page.
- Often, JavaScripts appear in the `<head>` section of the XHTML document.
- The browser interprets the contents of the `<head>` section first.
- The `<script>` tag indicates to the browser that the text that follows is part of a script. Attribute `type` specifies the scripting language used in the script—such as `text/javascript`.
- A string of characters can be contained between double (`"`) or single (`'`) quotation marks.
- A string is sometimes called a character string, a message or a string literal.
- The browser's `document` object represents the XHTML document currently being displayed in the browser. The `document` object allows a script programmer to specify XHTML text to be displayed in the XHTML document.
- The browser contains a complete set of objects that allow script programmers to access and manipulate every element of an XHTML document.
- An object resides in the computer's memory and contains information used by the script. The term object normally implies that attributes (data) and behaviors (methods) are associated with the object. The object's methods use the attributes' data to perform useful actions for the client of the object—the script that calls the methods.
- The `document` object's `writeln` method writes a line of XHTML text in the XHTML document.
- The parentheses following the name of a method contain the arguments that the method requires to perform its task (or its action).
- Using `writeln` to write a line of XHTML text into a `document` does not guarantee that a corresponding line of text will appear in the XHTML document. The text displayed is dependent on the contents of the string written, which is subsequently rendered by the browser. The browser will interpret the XHTML elements as it normally does to render the final text in the document.
- Every statement should end with a semicolon (also known as the statement terminator), although none is required by JavaScript.
- JavaScript is case sensitive. Not using the proper uppercase and lowercase letters is a syntax error.

Section 6.3 Modifying Our First Program

- Sometimes it is useful to display information in windows called dialogs that "pop up" on the screen to grab the user's attention. Dialogs are typically used to display important messages to the user browsing the web page. The browser's `window` object uses method `alert` to display an alert dialog. Method `alert` requires as its argument the string to be displayed.
- When a backslash is encountered in a string of characters, the next character is combined with the backslash to form an escape sequence. The escape sequence `\n` is the newline character. It causes the cursor in the XHTML document to move to the beginning of the next line.

Section 6.4 Obtaining User Input with `prompt` Dialogs

- Keywords are words with special meaning in JavaScript.

- The keyword var is used to declare the names of variables. A variable is a location in the computer's memory where a value can be stored for use by a program. All variables have a name, type and value, and should be declared with a var statement before they are used in a program.
- A variable name can be any valid identifier consisting of letters, digits, underscores (_) and dollar signs ($) that does not begin with a digit and is not a reserved JavaScript keyword.
- Declarations end with a semicolon (;) and can be split over several lines, with each variable in the declaration separated by a comma (forming a comma-separated list of variable names). Several variables may be declared in one declaration or in multiple declarations.
- Programmers often indicate the purpose of a variable in the program by placing a JavaScript comment at the end of the variable's declaration. A single-line comment begins with the characters / / and terminates at the end of the line. Comments do not cause the browser to perform any action when the script is interpreted; rather, comments are ignored by the JavaScript interpreter.
- Multiline comments begin with delimiter /* and end with delimiter */. All text between the delimiters of the comment is ignored by the interpreter.
- The window object's prompt method displays a dialog into which the user can type a value. The first argument is a message (called a prompt) that directs the user to take a specific action. The optional second argument is the default string to display in the text field.
- A variable is assigned a value with an assignment statement, using the assignment operator, =. The = operator is called a binary operator, because it has two operands.
- The null keyword signifies that a variable has no value. Note that null is not a string literal, but rather a predefined term indicating the absence of value. Writing a null value to the document, however, displays the word "null".
- Function parseInt converts its string argument to an integer.
- JavaScript has a version of the + operator for string concatenation that enables a string and a value of another data type (including another string) to be concatenated.

Section 6.5 Memory Concepts

- Variable names correspond to locations in the computer's memory. Every variable has a name, a type and a value.
- When a value is placed in a memory location, the value replaces the previous value in that location. When a value is read out of a memory location, the process is nondestructive.
- JavaScript does not require variables to have a type before they can be used in a program. A variable in JavaScript can contain a value of any data type, and in many situations, JavaScript automatically converts between values of different types for you. For this reason, JavaScript is referred to as a loosely typed language.
- When a variable is declared in JavaScript, but is not given a value, it has an undefined value. Attempting to use the value of such a variable is normally a logic error.
- When variables are declared, they are not assigned default values, unless specified otherwise by the programmer. To indicate that a variable does not contain a value, you can assign the value null to it.

Section 6.6 Arithmetic

- The basic arithmetic operators (+, -, *, /, and %) are binary operators, because they each operate on two operands.
- Parentheses can be used to group expressions as in algebra.
- Operators in arithmetic expressions are applied in a precise sequence determined by the rules of operator precedence.

- When we say that operators are applied from left to right, we are referring to the associativity of the operators. Some operators associate from right to left.

Section 6.7 Decision Making: Equality and Relational Operators

- JavaScript's `if` statement allows a program to make a decision based on the truth or falsity of a condition. If the condition is met (i.e., the condition is true), the statement in the body of the `if` statement is executed. If the condition is not met (i.e., the condition is false), the statement in the body of the `if` statement is not executed.
- Conditions in `if` statements can be formed by using the equality operators and relational operators.

Terminology

`\"` double-quote escape sequence
`\\` backslash escape sequence
`\'` single quote escape sequence
`\n` newline escape sequence
`\r` carriage return escape sequence
`\t` tab escape sequence
action
addition operator (+)
`alert` dialog
`alert` method of the `window` object
argument to a method
arithmetic expression in straight-line form
arithmetic operator
assignment
assignment operator (=)
assignment statement
associativity of operators
attribute
backslash (\) escape character
behavior
binary operator
case sensitive
character string
client of an object
comma-separated list
comment
condition
cursor
data
data type
`Date` object
decision making
declaration
dialog
division operator (/)
`document` object
double quotation (") marks
ECMAScript standard
empty statement
equality operators
error message
escape sequence
false
function
identifier
`if` statement
inline scripting
integer
interpreter
JavaScript
JavaScript interpreter
keyword
level of precedence
literal
location in the computer's memory
logic error
loosely typed language
meaningful variable name
method
mouse cursor
multiline comment (/* and */)
multiplication operator (*)
name of a variable
`NaN` (not a number)
nested `if` statements
`new` operator
newline character (`\n`)
`null`
object
operand
operator associativity
operator precedence
parentheses
`parseInt` function
perform an action
`pre` element

program
prompt
`prompt` dialog
`prompt` method of the `window` object
redundant parentheses
relational operator
remainder after division
remainder operator (`%`)
rules of operator precedence
runtime error
script
`script` element
scripting language
semicolon (`;`) statement terminator
single quotation (`'`) mark
single-line comment (`//`)
statement
straight-line form
string
string concatenation
string concatenation operator (`+`)
string literal
string of characters
subtraction operator (`-`)
syntax error
text field
title bar of a dialog
true
`type` attribute of the `<script>` tag
type of a variable
undefined
value of a variable
`var` keyword
variable
violation of the language rules
white-space character
whole number
`window` object
`write` method of the `document` object
`writeln` method of the `document` object

Self-Review Exercises

6.1 Fill in the blanks in each of the following statements:

a) ________ begins a single-line comment.
b) Every statement should end with a(n) ________.
c) The ________ statement is used to make decisions.
d) ________, ________, ________ and ________ are known as white space.
e) The ________ object displays `alert` dialogs and `prompt` dialogs.
f) ________ are words that are reserved for use by JavaScript.
g) Methods ________ and ________ of the ________ object write XHTML text into an XHTML document.

6.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Comments cause the computer to print the text after the `//` on the screen when the program is executed.
b) JavaScript considers the variables `number` and `NuMbEr` to be identical.
c) The remainder operator (`%`) can be used only with numeric operands.
d) The arithmetic operators `*`, `/`, `%`, `+` and `-` all have the same level of precedence.
e) Method `parseInt` converts an integer to a string.

6.3 Write JavaScript statements to accomplish each of the following tasks:

a) Declare variables `c`, `thisIsAVariable`, `q76354` and `number`.
b) Display a dialog asking the user to enter an integer. Show a default value of `0` in the text field.
c) Convert a string to an integer, and store the converted value in variable `age`. Assume that the string is stored in `stringValue`.
d) If the variable `number` is not equal to `7`, display `"The variable number is not equal to 7"` in a message dialog.
e) Output a line of XHTML text that will display the message `"This is a JavaScript program"` on one line in the XHTML document.

f) Output a line of XHTML text that will display the message `"This is a JavaScript program"` on two lines in the XHTML document. Use only one statement.

6.4 Identify and correct the errors in each of the following statements:

a)
```
if ( c < 7 );
   window.alert( "c is less than 7" );
```
b)
```
if ( c => 7 )
   window.alert( "c is equal to or greater than 7" );
```

6.5 Write a statement (or comment) to accomplish each of the following tasks:

a) State that a program will calculate the product of three integers [*Hint:* Use text that helps to document a program.]

b) Declare the variables `x`, `y`, `z` and `result`.

c) Declare the variables `xVal`, `yVal` and `zVal`.

d) Prompt the user to enter the first value, read the value from the user and store it in the variable `xVal`.

e) Prompt the user to enter the second value, read the value from the user and store it in the variable `yVal`.

f) Prompt the user to enter the third value, read the value from the user and store it in the variable `zVal`.

g) Convert `xVal` to an integer, and store the result in the variable `x`.

h) Convert `yVal` to an integer, and store the result in the variable `y`.

i) Convert `zVal` to an integer, and store the result in the variable `z`.

j) Compute the product of the three integers contained in variables `x`, `y` and `z`, and assign the result to the variable `result`.

k) Write a line of XHTML text containing the string `"The product is "` followed by the value of the variable `result`.

6.6 Using the statements you wrote in Exercise 6.5, write a complete program that calculates and prints the product of three integers.

Answers to Self-Review Exercises

6.1 a) `//`. b) Semicolon (`;`). c) `if`. d) Space characters, newline characters and tab characters. e) `window`. f) Keywords. g) `write`, `writeln`, `document`.

6.2 a) False. Comments do not cause any action to be performed when the program is executed. They are used to document programs and improve their readability. b) False. JavaScript is case sensitive, so these variables are distinct. c) True. d) False. The operators `*`, `/` and `%` are on the same level of precedence, and the operators `+` and `-` are on a lower level of precedence. e) False. Function `parseInt` converts a string to an integer value.

6.3

a) `var c, thisIsAVariable, q76354, number;`

b) `value = window.prompt( "Enter an integer", "0" );`

c) `var age = parseInt( stringValue );`

d)
```
if ( number != 7 )
   window.alert( "The variable number is not equal to 7" );
```
e) `document.writeln( "This is a JavaScript program" );`

f) `document.writeln( "This is a<br />JavaScript program" );`

6.4 a) Error: There should not be a semicolon after the right parenthesis of the condition in the `if` statement.

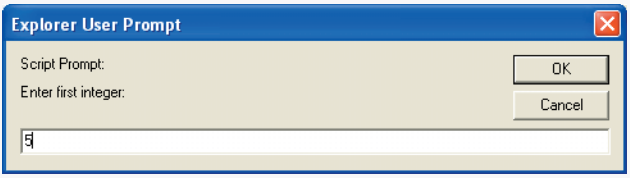
Correction: Remove the semicolon after the right parenthesis. [*Note:* The result of this error is that the output statement is executed whether or not the condition in the `if` statement is true. The semicolon after the right parenthesis is considered an empty statement—a statement that does nothing.]

b) Error: The relational operator => is incorrect.
Correction: Change => to >=.

6.5 a) `// Calculate the product of three integers`
b) `var x, y, z, result;`
c) `var xVal, yVal, zVal;`
d) `xVal = window.prompt( "Enter first integer:", "0" );`
e) `yVal = window.prompt( "Enter second integer:", "0" );`
f) `zVal = window.prompt( "Enter third integer:", "0" );`
g) `x = parseInt( xVal );`
h) `y = parseInt( yVal );`
i) `z = parseInt( zVal );`
j) `result = x * y * z;`
k) `document.writeln( "<h1>The product is " + result + "</h1>" );`

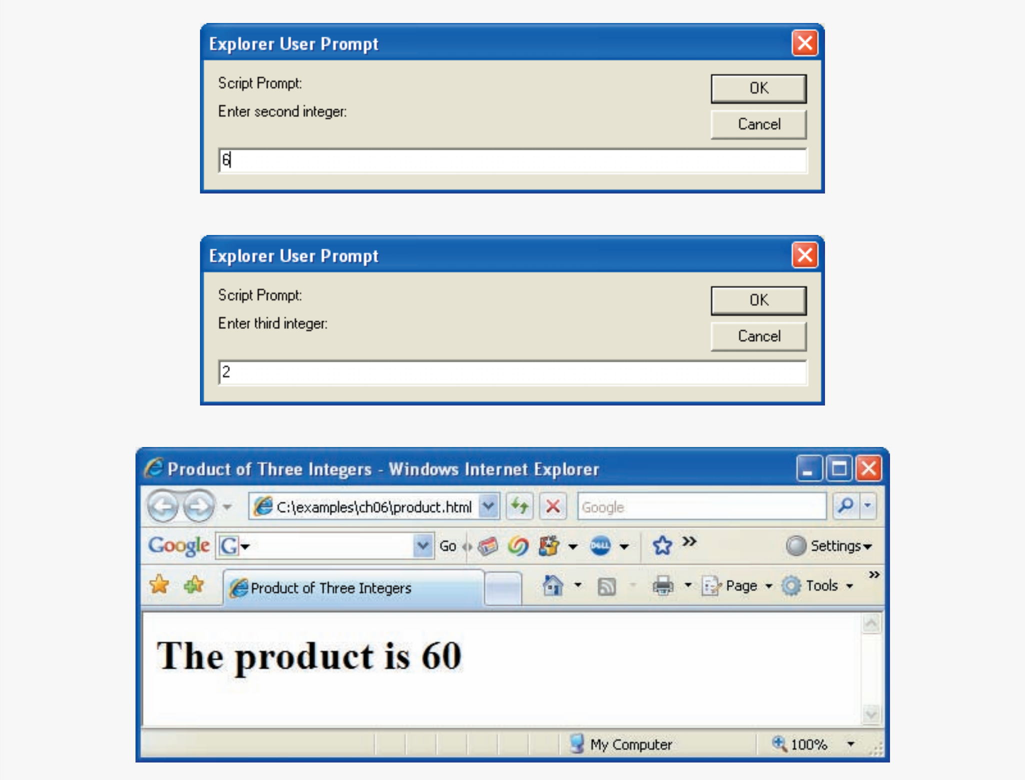
6.6 The program is as follows:

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 6.6: product.html -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Product of Three Integers</title>
      <script type = "text/javascript">
         <!--
         // Calculate the product of three integers
         var x, y, z, result;
         var xVal, yVal, zVal;

         xVal = window.prompt( "Enter first integer:", "0" );
         yVal = window.prompt( "Enter second integer:", "0" );
         zVal = window.prompt( "Enter third integer:", "0" );

         x = parseInt( xVal );
         y = parseInt( yVal );
         z = parseInt( zVal );

         result = x * y * z;
         document.writeln( "<h1>The product is " +
            result + "<h1>" );
         // -->
      </script>
   </head><body></body>
</html>
```



Script Prompt:
Enter first integer:

5



Script Prompt:
Enter second integer:
6

Script Prompt:
Enter third integer:
2

The product is 60

Explorer User Prompt
OK
Cancel
Explorer User Prompt
OK
Cancel
Product of Three Integers - Windows Internet Explorer
C:\examples\ch06\product.html
Google
Go
Settings
Product of Three Integers
Page
Tools
My Computer
100%

Exercises

6.7 Fill in the blanks in each of the following statements:

a) ________ are used to document a program and improve its readability.

b) A dialog capable of receiving input from the user is displayed with method ________ of object ________.

c) A JavaScript statement that makes a decision is the ________ statement.

d) Calculations are normally performed by ________ operators.

e) A dialog capable of showing a message to the user is displayed with method ________ of object ________.

6.8 Write JavaScript statements that accomplish each of the following tasks:

a) Display the message `"Enter two numbers"` using the `window` object.

b) Assign the product of variables `b` and `c` to variable `a`.

c) State that a program performs a sample payroll calculation.

6.9 State whether each of the following is *true* or *false*. If *false*, explain why.

a) JavaScript operators are evaluated from left to right.

b) The following are all valid variable names: `_under_bar_`, `m928134`, `t5`, `j7`, `her_sales$`, `his_$account_total`, `a`, `b$`, `c`, `z`, `z2`.

c) A valid JavaScript arithmetic expression with no parentheses is evaluated from left to right.

d) The following are all invalid variable names: `3g`, `87`, `67h2`, `h22`, `2h`.

6.10 Fill in the blanks in each of the following statements:

a) What arithmetic operations have the same precedence as multiplication? __________.

b) When parentheses are nested, which set of parentheses is evaluated first in an arithmetic expression? __________.

c) A location in the computer's memory that may contain different values at various times throughout the execution of a program is called a __________.

6.11 What displays in the message dialog when each of the given JavaScript statements is performed? Assume that `x = 2` and `y = 3`.

a) `window.alert( "x = " + x );`

b) `window.alert( "The value of x + x is " + ( x + x ) );`

c) `window.alert( "x =" );`

d) `window.alert( ( x + y ) + " = " + ( y + x ) );`

6.12 Which of the following JavaScript statements contain variables whose values are destroyed (i.e., changed or replaced)?

a) `p = i + j + k + 7;`

b) `window.alert( "variables whose values are destroyed" );`

c) `window.alert( "a = 5" );`

d) `stringVal = window.prompt( "Enter string:" );`

6.13 Given $y = ax^3 + 7$, which of the following are correct JavaScript statements for this equation?

a) `y = a * x * x * x + 7;`

b) `y = a * x * x * (x + 7);`

c) `y = (a * x) * x * (x + 7);`

d) `y = (a * x) * x * x + 7;`

e) `y = a * (x * x * x) + 7;`

f) `y = a * x * (x * x + 7);`

6.14 State the order of evaluation of the operators in each of the following JavaScript statements, and show the value of `x` after each statement is performed.

a) `x = 7 + 3 * 6 / 2 - 1;`

b) `x = 2 % 2 + 2 * 2 - 2 / 2;`

c) `x = ( 3 * 9 * ( 3 + ( 9 * 3 / ( 3 ) ) ) );`

6.15 Write a script that displays the numbers 1 to 4 on the same line, with each pair of adjacent numbers separated by one space. Write the program using the following methods:

a) Using one `document.writeln` statement.

b) Using four `document.write` statements.

6.16 Write a script that asks the user to enter two numbers, obtains the two numbers from the user and outputs text that displays the sum, product, difference and quotient of the two numbers. Use the techniques shown in Fig. 6.9.

6.17 Write a script that asks the user to enter two integers, obtains the numbers from the user and outputs text that displays the larger number followed by the words "`is larger`" in an alert dialog. If the numbers are equal, output XHTML text that displays the message "`These numbers are equal`." Use the techniques shown in Fig. 6.17.

6.18 Write a script that takes three integers from the user and displays the sum, average, product, smallest and largest of the numbers in an `alert` dialog.

6.19 Write a script that gets from the user the radius of a circle and outputs XHTML text that displays the circle's diameter, circumference and area. Use the constant value 3.14159 for π. Use the GUI techniques shown in Fig. 6.9. [*Note:* You may also use the predefined constant `Math.PI` for the value of π. This constant is more precise than the value 3.14159. The `Math` object is defined by Java-

Script and provides many common mathematical capabilities.] Use the following formulas (*r* is the radius): *diameter* = 2*r*, *circumference* = 2π*r*, *area* = πr^2.

6.20 Write a script that outputs XHTML text that displays in the XHTML document a rectangle, an oval, an arrow and a diamond using asterisks (*), as follows [*Note:* Use the `<pre>` and `</pre>` tags to specify that the asterisks should be displayed using a fixed-width font]:

```
*********          ***              *              *
*       *        *     *           ***            * *
*       *       *       *         *****          *   *
*       *       *       *           *           *     *
*       *       *       *           *          *       *
*       *       *       *           *           *     *
*       *       *       *           *            *   *
*       *        *     *            *             * *
*********          ***              *              *
```

6.21 Modify the program you created in Exercise 6.20 by removing the `<pre>` and `</pre>` tags. Does the program display the shapes exactly as in Exercise 6.20?

6.22 What does the following code print?

```
document.writeln( "*\n**\n***\n****\n*****" );
```

6.23 What does the following code print?

```
document.writeln( "*" );
document.writeln( "***" );
document.writeln( "*****" );
document.writeln( "****" );
document.writeln( "**" );
```

6.24 What does the following code print?

```
document.write( "*<br />" );
document.write( "***<br />" );
document.write( "*****<br />" );
document.write( "****<br />" );
document.writeln( "**" );
```

6.25 What does the following code print?

```
document.write( "*<br />" );
document.writeln( "***" );
document.writeln( "*****" );
document.write( "****<br />" );
document.writeln( "**" );
```

6.26 Write a script that reads five integers and determines and outputs XHTML text that displays the largest and smallest integers in the group. Use only the programming techniques you learned in this chapter.

6.27 Write a script that reads an integer and determines and outputs XHTML text that displays whether it is odd or even. [*Hint:* Use the remainder operator. An even number is a multiple of 2. Any multiple of 2 leaves a remainder of zero when divided by 2.]

6.28 Write a script that reads in two integers and determines and outputs XHTML text that displays whether the first is a multiple of the second. [*Hint*: Use the remainder operator.]

6.29 Write a script that outputs XHTML text that displays in the XHTML document a checkerboard pattern, as follows:

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

6.30 Write a script that inputs five numbers and determines and outputs XHTML text that displays the number of negative numbers input, the number of positive numbers input and the number of zeros input.

6.31 Write a script that calculates the squares and cubes of the numbers from 0 to 10 and outputs XHTML text that displays the resulting values in an XHTML table format, as follows:

```
number  square  cube
0       0       0
1       1       1
2       4       8
3       9       27
4       16      64
5       25      125
6       36      216
7       49      343
8       64      512
9       81      729
10      100     1000
```

[*Note:* This program does not require any input from the user.]

7

JavaScript: Control Statements I

Let's all move one place on.
—Lewis Carroll

The wheel is come full circle.
—William Shakespeare

How many apples fell on Newton's head before he took the hint!
—Robert Frost

OBJECTIVES

In this chapter you will learn:

- Basic problem-solving techniques.
- To develop algorithms through the process of top-down, stepwise refinement.
- To use the `if` and `if`...`else` selection statements to choose among alternative actions.
- To use the `while` repetition statement to execute statements in a script repeatedly.
- Counter-controlled repetition and sentinel-controlled repetition.
- To use the increment, decrement and assignment operators.

Outline

7.1 Introduction

Before writing a script to solve a problem, it is essential to have a thorough understanding of the problem and a carefully planned approach to solving the problem. When writing a script, it is equally essential to understand the types of building blocks that are available and to employ proven program-construction principles. In this chapter and in Chapter 8, we discuss these issues in our presentation of the theory and principles of structured programming. The techniques you will learn here are applicable to most high-level languages, including JavaScript.

7.2 Algorithms

Any computable problem can be solved by executing a series of actions in a specific order. A **procedure** for solving a problem in terms of

1. the **actions** to be executed, and
2. the **order** in which the actions are to be executed

is called an **algorithm**. The following example demonstrates that correctly specifying the order in which the actions are to execute is important.

Consider the “rise-and-shine algorithm” followed by one junior executive for getting out of bed and going to work: (1) get out of bed, (2) take off pajamas, (3) take a shower, (4) get dressed, (5) eat breakfast, (6) carpool to work. This routine gets the executive to work well prepared to make critical decisions. Suppose, however, that the same steps are performed in a slightly different order: (1) get out of bed, (2) take off pajamas, (3) get dressed, (4) take a shower, (5) eat breakfast, (6) carpool to work. In this case, our junior executive shows up for work soaking wet. Specifying the order in which statements are to be executed in a computer program is called **program control**. In this chapter and Chapter 8, we investigate the program-control capabilities of JavaScript.

7.3 Pseudocode

Pseudocode is an artificial and informal language that helps programmers develop algorithms. The pseudocode we present here is useful for developing algorithms that will be converted to structured portions of JavaScript programs. Pseudocode is similar to everyday English; it is convenient and user friendly, although it is not an actual computer programming language.

Software Engineering Observation 7.1

Pseudocode is often used to "think out" a program during the program-design process. Then the pseudocode program is converted to a programming language such as JavaScript.

The style of pseudocode we present consists purely of characters, so that programmers may conveniently type pseudocode in an editor program. The computer can produce a fresh printed copy of a pseudocode program on demand. Carefully prepared pseudocode may easily be converted to a corresponding JavaScript program. This process is done in many cases simply by replacing pseudocode statements with their JavaScript equivalents. In this chapter, we give several examples of pseudocode.

Pseudocode normally describes only executable statements—the actions that are performed when the program is converted from pseudocode to JavaScript and is run. Declarations are not executable statements. For example, the declaration

```
var value1;
```

instructs the JavaScript interpreter to reserve space in memory for the variable `value1`. This declaration does not cause any action—such as input, output or a calculation—to occur when the script executes. Some programmers choose to list variables and mention the purpose of each variable at the beginning of a pseudocode program.

7.4 Control Structures

Normally, statements in a program execute one after the other in the order in which they are written. This process is called **sequential execution**. Various JavaScript statements we will soon discuss enable the programmer to specify that the next statement to execute may not be the next one in sequence. This is known as **transfer of control**.

During the 1960s, it became clear that the indiscriminate use of transfers of control was the root of much difficulty experienced by software development groups. The finger of blame was pointed at the **goto statement**, which allowed the programmer to specify a transfer of control to one of a wide range of possible destinations in a program. The notion of so-called **structured programming** became almost synonymous with "**goto elimination**." JavaScript does not have a `goto` statement.

The research of Bohm and Jacopini demonstrated that programs could be written without `goto` statements.[1] The challenge of the era for programmers was to shift their styles to "`goto`-less programming." It was not until the 1970s that programmers started taking structured programming seriously. The results were impressive, as software development groups reported reduced development times, more frequent on-time delivery of

[1] Bohm, C., and G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5, May 1966, pp. 336–371.

systems and more frequent within-budget completion of software projects. The key to these successes is that structured programs are clearer, easier to debug and modify and more likely to be bug free in the first place.

Bohm and Jacopini's work demonstrated that all programs could be written in terms of only three control structures, namely the sequence structure, the selection structure and the repetition structure. The sequence structure is built into JavaScript. Unless directed otherwise, the computer executes JavaScript statements one after the other in the order in which they are written (i.e., in sequence). The flowchart segment of Fig. 7.1 illustrates a typical sequence structure in which two calculations are performed in order.

A flowchart is a graphical representation of an algorithm or of a portion of an algorithm. Flowcharts are drawn using certain special-purpose symbols, such as rectangles, diamonds, ovals and small circles; these symbols are connected by arrows called flowlines, which indicate the order in which the actions of the algorithm execute.

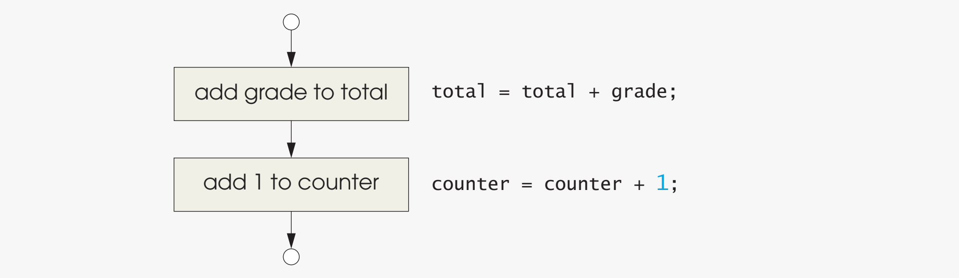
Like pseudocode, flowcharts often are useful for developing and representing algorithms, although pseudocode is strongly preferred by many programmers. Flowcharts show clearly how control structures operate; that is all we use them for in this text. Carefully compare the pseudocode and flowchart representations of each control structure.

Consider the flowchart segment for the sequence structure on the left side of Fig. 7.1. We use the rectangle symbol (or action symbol) to indicate any type of action, including a calculation or an input/output operation. The flowlines in the figure indicate the order in which the actions are performed—the first action adds `grade` to `total`, then the second action adds `1` to `counter`. JavaScript allows us to have as many actions as we want in a sequence structure. Anywhere a single action may be placed, as we will soon see, we may place several actions in sequence.

In a flowchart that represents a *complete* algorithm, an oval symbol containing the word “Begin” is the first symbol used; an oval symbol containing the word “End” indicates where the algorithm ends. In a flowchart that shows only a portion of an algorithm, as in Fig. 7.1, the oval symbols are omitted in favor of using small circle symbols, also called connector symbols.

Perhaps the most important flowcharting symbol is the diamond symbol, also called the decision symbol, which indicates that a decision is to be made. We discuss the diamond symbol in the next section.

JavaScript provides three types of selection structures; we discuss each in this chapter and in Chapter 8. The `if` selection statement performs (selects) an action if a condition is true or skips the action if the condition is false. The `if...else` selection statement per-



```
total = total + grade;

counter = counter + 1;
```

Fig. 7.1 | Flowcharting JavaScript's sequence structure.

forms an action if a condition is true and performs a different action if the condition is false. The `switch` selection statement (Chapter 8) performs one of many different actions, depending on the value of an expression.

The **`if`** statement is called a **single-selection structure** because it selects or ignores a single action (or, as we will soon see, a single group of actions). The `if…else` statement is a **double-selection structure** because it selects between two different actions (or groups of actions). The `switch` statement is a **multiple-selection structure** because it selects among many different actions (or groups of actions).

JavaScript provides four repetition structure types, namely **`while`**, `do…while`, `for` and `for…in`. (`do…while` and `for` are covered in Chapter 8; `for…in` is covered in Chapter 10.) Each of the words `if`, `else`, `switch`, `while`, `do`, `for` and `in` is a JavaScript **keyword**. These words are reserved by the language to implement various features, such as JavaScript's control structures. Keywords cannot be used as identifiers (e.g., for variable names). A complete list of JavaScript keywords is shown in Fig. 7.2.

Common Programming Error 7.1

Using a keyword as an identifier is a syntax error.

As we have shown, JavaScript has only eight control structures: sequence, three types of selection and four types of repetition. A program is formed by combining control structures as necessary to implement the program's algorithm. As with the sequence structure in Fig. 7.1, we will see that each control structure is flowcharted with two small circle symbols, one at the entry point to the control structure and one at the exit point.

JavaScript keywords				
`break`	`case`	`catch`	`continue`	`default`
`delete`	`do`	`else`	`false`	`finally`
`for`	`function`	`if`	`in`	`instanceof`
`new`	`null`	`return`	`switch`	`this`
`throw`	`true`	`try`	`typeof`	`var`
`void`	`while`	`with`		
Keywords that are reserved but not used by JavaScript				
`abstract`	`boolean`	`byte`	`char`	`class`
`const`	`debugger`	`double`	`enum`	`export`
`extends`	`final`	`float`	`goto`	`implements`
`import`	`int`	`interface`	`long`	`native`
`package`	`private`	`protected`	`public`	`short`
`static`	`super`	`synchronized`	`throws`	`transient`
`volatile`				

Fig. 7.2 | JavaScript keywords.

Single-entry/single-exit control structures make it easy to build programs; the control structures are attached to one another by connecting the exit point of one to the entry point of the next. This process is similar to the way in which a child stacks building blocks, so we call it control-structure stacking. We will learn that there is only one other way in which control structures may be connected—control-structure nesting. Thus, algorithms in JavaScript programs are constructed from only eight different types of control structures combined in only two ways.

7.5 if Selection Statement

A selection structure is used to choose among alternative courses of action in a program. For example, suppose that the passing grade on an examination is 60 (out of 100). Then the pseudocode statement

```
If student's grade is greater than or equal to 60
    Print "Passed"
```

determines whether the condition "student's grade is greater than or equal to 60" is true or false. If the condition is true, then "Passed" is printed, and the next pseudocode statement in order is "performed" (remember that pseudocode is not a real programming language). If the condition is false, the print statement is ignored, and the next pseudocode statement in order is performed.

Note that the second line of this selection structure is indented. Such indentation is optional but is highly recommended, because it emphasizes the inherent structure of structured programs. The JavaScript interpreter ignores white-space characters—blanks, tabs and newlines used for indentation and vertical spacing. Programmers insert these white-space characters to enhance program clarity.

Good Programming Practice 7.1

Consistently applying reasonable indentation conventions throughout your programs improves program readability. We suggest a fixed-size tab of about 1/4 inch or three spaces per indent.

The preceding pseudocode *If* statement can be written in JavaScript as

```
if ( studentGrade >= 60 )
   document.writeln( "Passed" );
```

Note that the JavaScript code corresponds closely to the pseudocode. This similarity is the reason that pseudocode is a useful program-development tool. The statement in the body of the `if` statement outputs the character string `"Passed"` in the XHTML document.

The flowchart in Fig. 7.3 illustrates the single-selection `if` statement. This flowchart contains what is perhaps the most important flowcharting symbol—the diamond symbol (or decision symbol), which indicates that a decision is to be made. The decision symbol contains an expression, such as a condition, that can be either true or false. The decision symbol has two flowlines emerging from it. One indicates the path to follow in the program when the expression in the symbol is true; the other indicates the path to follow in the program when the expression is false. A decision can be made on any expression that evaluates to a value of JavaScript's boolean type (i.e., any expression that evaluates to `true` or `false`—also known as a boolean expression).

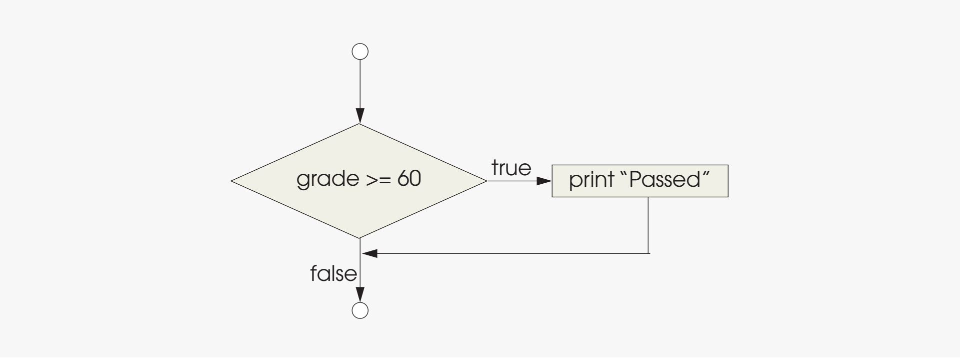


Fig. 7.3 | Flowcharting the single-selection `if` statement.

Software Engineering Observation 7.2

In JavaScript, any nonzero numeric value in a condition evaluates to `true`*, and 0 evaluates to* `false`*. For strings, any string containing one or more characters evaluates to* `true`*, and the empty string (the string containing no characters, represented as* `""`*) evaluates to* `false`*. Also, a variable that has been declared with* `var` *but has not been assigned a value evaluates to* `false`*.*

Note that the `if` statement is a single-entry/single-exit control structure. We will soon learn that the flowcharts for the remaining control structures also contain (besides small circle symbols and flowlines) only rectangle symbols, to indicate the actions to be performed, and diamond symbols, to indicate decisions to be made. This type of flowchart represents the action/decision model of programming.

We can envision eight bins, each containing only the control structures of one of the eight types. These control structures are empty. Nothing is written in the rectangles or in the diamonds. The programmer's task, then, is to assemble a program from as many of each type of control structure as the algorithm demands, combining them in only two possible ways (stacking or nesting), then filling in the actions and decisions in a manner appropriate for the algorithm. We will discuss the variety of ways in which actions and decisions may be written.

7.6 `if...else` Selection Statement

The `if` selection statement performs an indicated action only when the condition evaluates to `true`; otherwise, the action is skipped. The `if...else` selection statement allows the programmer to specify that a different action is to be performed when the condition is true than when the condition is false. For example, the pseudocode statement

```
If student's grade is greater than or equal to 60
    Print "Passed"
Else
    Print "Failed"
```

prints `Passed` if the student's grade is greater than or equal to 60 and prints `Failed` if the student's grade is less than 60. In either case, after printing occurs, the next pseudocode statement in sequence (i.e., the next statement after the whole `if...else` structure) is performed. Note that the body of the *Else* part of the structure is also indented.

Good Programming Practice 7.2

Indent both body statements of an `if...else` *statement.*

The indentation convention you choose should be applied carefully throughout your programs (both in pseudocode and in JavaScript). It is difficult to read programs that do not use uniform spacing conventions.

The preceding pseudocode *If…Else* statement may be written in JavaScript as

```
if ( studentGrade >= 60 )
   document.writeln( "Passed" );
else
   document.writeln( "Failed" );
```

The flowchart shown in Fig. 7.4 illustrates the `if...else` selection statement's flow of control. Once again, note that the only symbols in the flowchart (besides small circles and arrows) are rectangles (for actions) and a diamond (for a decision). We continue to emphasize this action/decision model of computing. Imagine again a deep bin containing as many empty double-selection structures as might be needed to build a JavaScript algorithm. The programmer's job is to assemble the selection structures (by stacking and nesting) with other control structures required by the algorithm and to fill in the empty rectangles and empty diamonds with actions and decisions appropriate to the algorithm's implementation.

JavaScript provides an operator, called the **conditional operator** (`?:`), that is closely related to the `if...else` statement. The operator `?:` is JavaScript's only **ternary operator**—it takes three operands. The operands together with the `?:` form a **conditional expression**. The first operand is a boolean expression, the second is the value for the conditional expression if the expression evaluates to `true` and the third is the value for the conditional expression if the expression evaluates to `false`. For example, consider the following statement

```
document.writeln( studentGrade >= 60 ? "Passed" : "Failed" );
```

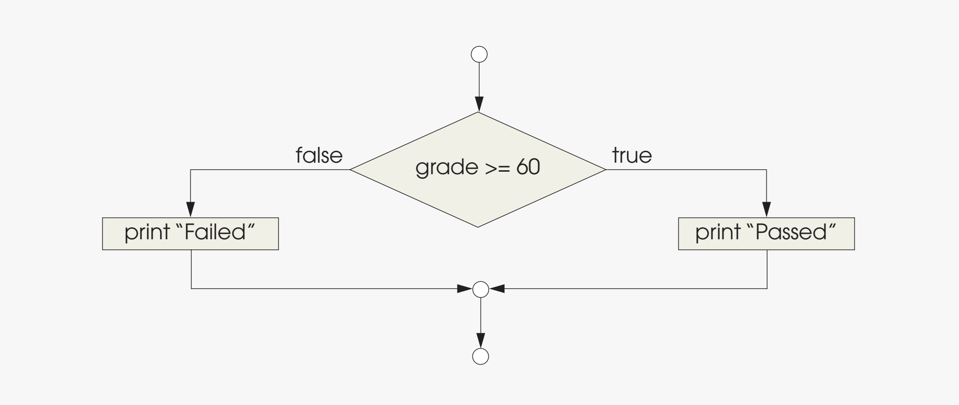


Fig. 7.4 | Flowcharting the double-selection `if...else` statement.

contains a conditional expression that evaluates to the string "Passed" if the condition studentGrade >= 60 is true and evaluates to the string "Failed" if the condition is false. Thus, this statement with the conditional operator performs essentially the same operation as the preceding if…else statement. The precedence of the conditional operator is low, so the entire conditional expression is normally placed in parentheses to ensure that it evaluates correctly.

Nested if…else statements test for multiple cases by placing if…else statements inside if…else statements. For example, the following pseudocode statement indicates that the program should print A for exam grades greater than or equal to 90, B for grades in the range 80 to 89, C for grades in the range 70 to 79, D for grades in the range 60 to 69 and F for all other grades:

```
If student's grade is greater than or equal to 90
    Print "A"
Else
    If student's grade is greater than or equal to 80
        Print "B"
    Else
        If student's grade is greater than or equal to 70
            Print "C"
        Else
            If student's grade is greater than or equal to 60
                Print "D"
            Else
                Print "F"
```

This pseudocode may be written in JavaScript as

```
if ( studentGrade >= 90 )
   document.writeln( "A" );
else
   if ( studentGrade >= 80 )
      document.writeln( "B" );
   else
      if ( studentGrade >= 70 )
         document.writeln( "C" );
      else
         if ( studentGrade >= 60 )
            document.writeln( "D" );
         else
            document.writeln( "F" );
```

If studentGrade is greater than or equal to 90, all four conditions will be true, but only the document.writeln statement after the first test will execute. After that particular document.writeln executes, the else part of the outer if…else statements is skipped.

Good Programming Practice 7.3

If there are several levels of indentation, each level should be indented the same additional amount of space.

Most JavaScript programmers prefer to write the preceding `if` structure as

```
if ( grade >= 90 )
   document.writeln( "A" );
else if ( grade >= 80 )
   document.writeln( "B" );
else if ( grade >= 70 )
   document.writeln( "C" );
else if ( grade >= 60 )
   document.writeln( "D" );
else
   document.writeln( "F" );
```

The two forms are equivalent. The latter form is popular because it avoids the deep indentation of the code to the right. Such deep indentation often leaves little room on a line, forcing lines to be split and decreasing program readability.

It is important to note that the JavaScript interpreter always associates an `else` with the previous `if`, unless told to do otherwise by the placement of braces (`{}`). This situation is referred to as the **dangling-else problem**. For example,

```
if ( x > 5 )
   if ( y > 5 )
      document.writeln( "x and y are > 5" );
else
   document.writeln( "x is <= 5" );
```

appears to indicate with its indentation that if `x` is greater than `5`, the `if` structure in its body determines whether `y` is also greater than `5`. If so, the body of the nested `if` structure outputs the string `"x and y are > 5"`. Otherwise, it *appears* that if `x` is not greater than `5`, the `else` part of the `if...else` structure outputs the string `"x is <= 5"`.

Beware! The preceding nested `if` statement does not execute as it appears. The interpreter actually interprets the preceding statement as

```
if ( x > 5 )
   if ( y > 5 )
      document.writeln( "x and y are > 5" );
   else
      document.writeln( "x is <= 5" );
```

in which the body of the first `if` statement is a nested `if...else` statement. This statement tests whether `x` is greater than `5`. If so, execution continues by testing whether `y` is also greater than `5`. If the second condition is true, the proper string—`"x and y are > 5"`—is displayed. However, if the second condition is false, the string `"x is <= 5"` is displayed, even though we know that `x` is greater than `5`.

To force the preceding nested `if` statement to execute as it was intended originally, it must be written as follows:

```
if ( x > 5 )
{
   if ( y > 5 )
      document.writeln( "x and y are > 5" );
}
else
   document.writeln( "x is <= 5" );
```

The braces ({}) indicate to the interpreter that the second `if` statement is in the body of the first `if` statement and that the `else` is matched with the first `if` statement. In Exercises 7.21 and 7.22, you will investigate the dangling-else problem further.

The `if` selection statement expects only one statement in its body. To include several statements in an `if` statement's body, enclose the statements in braces (`{` and `}`). This can also be done in the else section of an `if…else` statement. A set of statements contained within a pair of braces is called a **block**.

Software Engineering Observation 7.3

A block can be placed anywhere in a program that a single statement can be placed.

Software Engineering Observation 7.4

Unlike individual statements, a block does not end with a semicolon. However, each statement within the braces of a block should end with a semicolon.

The following example includes a block in the `else` part of an `if…else` statement:

```
if ( grade >= 60 )
   document.writeln( "Passed" );
else
{
   document.writeln( "Failed<br />" );
   document.writeln( "You must take this course again." );
}
```

In this case, if `grade` is less than 60, the program executes both statements in the body of the `else` and prints

```
Failed.
You must take this course again.
```

Note the braces surrounding the two statements in the `else` clause. These braces are important. Without them, the statement

```
document.writeln( "You must take this course again." );
```

would be outside the body of the `else` part of the `if` and would execute regardless of whether the grade is less than 60.

Common Programming Error 7.2

Forgetting one or both of the braces that delimit a block can lead to syntax errors or logic errors.

Syntax errors (e.g., when one brace in a block is left out of the program) are caught by the interpreter when it attempts to interpret the code containing the syntax error. A **logic error** (e.g., the one caused when both braces around a block are left out of the program) also has its effect at execution time. A **fatal logic error** causes a program to fail and terminate prematurely. A **nonfatal logic error** allows a program to continue executing, but the program produces incorrect results.

Good Programming Practice 7.4

Some programmers prefer to type the beginning and ending braces of blocks before typing the individual statements within the braces. This helps avoid omitting one or both of the braces.

Software Engineering Observation 7.5

Just as a block can be placed anywhere a single statement can be placed, it is also possible to have no statement at all (the empty statement) in such places. The empty statement is represented by placing a semicolon (`;`) where a statement would normally be.

Common Programming Error 7.3

Placing a semicolon after the condition in an `if` structure leads to a logic error in single-selection `if` structures and a syntax error in double-selection `if` structures (if the `if` part contains a nonempty body statement).

7.7 `while` Repetition Statement

A repetition structure (also known as a **loop**) allows the programmer to specify that a script is to repeat an action while some condition remains true. The pseudocode statement

```
While there are more items on my shopping list
    Purchase next item and cross it off my list
```

describes the repetition that occurs during a shopping trip. The condition “there are more items on my shopping list” may be true or false. If it’s true, then the action “Purchase next item and cross it off my list” is performed. This action is performed repeatedly while the condition remains true. The statement(s) contained in the *While* repetition structure constitute its body. The body of a loop such as the *While* structure may be a single statement or a block. Eventually, the condition becomes false (i.e., when the last item on the shopping list has been purchased and crossed off the list). At this point, the repetition terminates, and the first pseudocode statement after the repetition structure executes.

Common Programming Error 7.4

*If the body of a `while` statement never causes the while statement’s condition to become true, a logic error occurs. Normally, such a repetition structure will never terminate—an error called an **infinite loop**. Both Internet Explorer and Firefox show a dialog allowing the user to terminate a script that contains an infinite loop.*

Common Programming Error 7.5

Remember that JavaScript is a case-sensitive language. In code, spelling the keyword `while` with an uppercase `W`, as in `While`, is a syntax error. All of JavaScript’s reserved keywords, such as `while`, `if` and `else`, contain only lowercase letters.

As an example of a `while` statement, consider a program segment designed to find the first power of 2 larger than 1000. Variable `product` begins with the value 2. The statement is as follows:

```
var product = 2;

while ( product <= 1000 )
   product = 2 * product;
```

When the `while` statement finishes executing, `product` contains the result 1024. The flowchart in Fig. 7.5 illustrates the flow of control of the preceding `while` repetition statement. Once again, note that (besides small circles and arrows) the flowchart contains only a rectangle symbol and a diamond symbol.

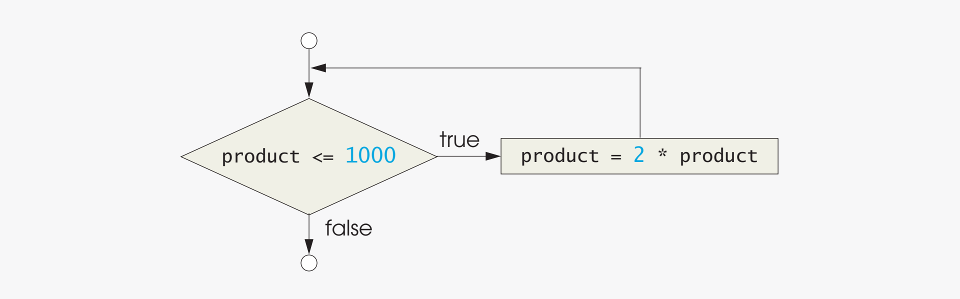


Fig. 7.5 | Flowcharting the `while` repetition statement.

When the script enters the `while` statement, `product` is 2. The script repeatedly multiplies variable `product` by 2, so `product` takes on the values 4, 8, 16, 32, 64, 128, 256, 512 and 1024 successively. When `product` becomes 1024, the condition `product <= 1000` in the `while` statement becomes `false`. This terminates the repetition, with 1024 as `product`'s final value. Execution continues with the next statement after the `while` statement. [*Note:* If a `while` statement's condition is initially `false`, the body statement(s) will never execute.]

The flowchart clearly shows the repetition. The flowline emerging from the rectangle wraps back to the decision, which the script tests each time through the loop until the decision eventually becomes false. At this point, the `while` statement exits, and control passes to the next statement in the program.

7.8 Formulating Algorithms: Counter-Controlled Repetition

To illustrate how to develop algorithms, we solve several variations of a class-averaging problem. Consider the following problem statement:

> *A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz.*

The class average is equal to the sum of the grades divided by the number of students (10 in this case). The algorithm for solving this problem on a computer must input each of the grades, perform the averaging calculation and display the result.

Let us use pseudocode to list the actions to execute and specify the order in which the actions should execute. We use **counter-controlled repetition** to input the grades one at a time. This technique uses a variable called a **counter** to control the number of times a set of statements executes. In this example, repetition terminates when the counter exceeds 10. In this section, we present a pseudocode algorithm (Fig. 7.6) and the corresponding program (Fig. 7.7). In the next section, we show how to develop pseudocode algorithms. Counter-controlled repetition often is called **definite repetition**, because the number of repetitions is known before the loop begins executing.

Note the references in the algorithm to a total and a counter. A **total** is a variable in which a script accumulates the sum of a series of values. A counter is a variable a script uses to count—in this case, to count the number of grades entered. Variables that store totals should normally be initialized to zero before they are used in a program.

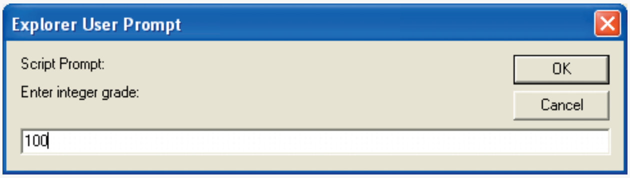
```
Set total to zero
Set grade counter to one

While grade counter is less than or equal to ten
    Input the next grade
    Add the grade into the total
    Add one to the grade counter

Set the class average to the total divided by ten
Print the class average
```

Fig. 7.6 | Pseudocode algorithm that uses counter-controlled repetition to solve the class-average problem.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 7.7: average.html -->
<!-- Counter-controlled repetition to calculate a class average. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Class Average Program</title>
      <script type = "text/javascript">
         <!--
         var total; // sum of grades
         var gradeCounter; // number of grades entered
         var grade; // grade typed by user (as a string)
         var gradeValue; // grade value (converted to integer)
         var average; // average of all grades

         // Initialization Phase
         total = 0; // clear total
         gradeCounter = 1; // prepare to loop

         // Processing Phase
         while ( gradeCounter <= 10 ) // loop 10 times
         {

            // prompt for input and read grade from user
            grade = window.prompt( "Enter integer grade:", "0" );

            // convert grade from a string to an integer
            gradeValue = parseInt( grade );

            // add gradeValue to total
            total = total + gradeValue;

```

Fig. 7.7 | Counter-controlled repetition to calculate a class average. (Part 1 of 2.)

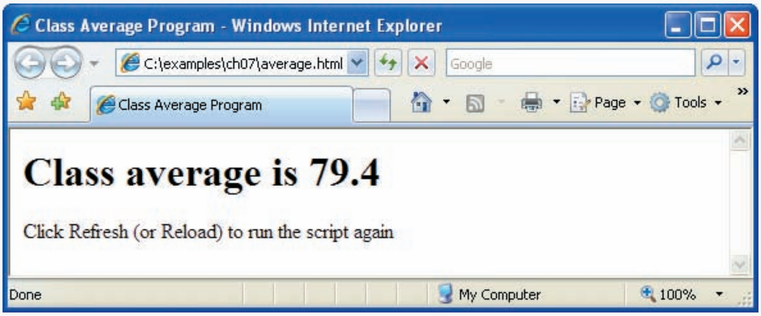
```
35             // add 1 to gradeCounter
36             gradeCounter = gradeCounter + 1;
37          } // end while
38
39          // Termination Phase
40          average = total / 10; // calculate the average
41
42          // display average of exam grades
43          document.writeln(
44             "<h1>Class average is " + average + "</h1>" );
45          // -->
46       </script>
47    </head>
48    <body>
49       <p>Click Refresh (or Reload) to run the script again<p>
50    </body>
51 </html>
```



Script Prompt:
Enter integer grade:

100

This dialog is displayed 10 times. User input is 100, 88, 93, 55, 68, 77, 83, 95, 73 and 62.


Class Average Program - Windows Internet Explorer
C:\examples\ch07\average.html
Google
Class Average Program
Page
Tools
Done
My Computer
100%

Class average is 79.4

Click Refresh (or Reload) to run the script again

Fig. 7.7 | Counter-controlled repetition to calculate a class average. (Part 2 of 2.)

Lines 12–16 declare variables `total`, `gradeCounter`, `grade`, `gradeValue`, `average`. The variable `grade` will store the string the user types into the `prompt` dialog. The variable `gradeValue` will store the integer value of the `grade` the user enters into the `prompt` dialog.

Lines 19–20 are assignment statements that initialize `total` to `0` and `gradeCounter` to `1`. Note that variables `total` and `gradeCounter` are initialized before they are used in a calculation.

Common Programming Error 7.6

Not initializing a variable that will be used in a calculation results in a logic error that produces the value `NaN—Not a Number`*. You must initialize the variable before it is used in a calculation.*

Line 23 indicates that the `while` statement continues iterating while the value of `gradeCounter` is less than or equal to 10. Line 27 corresponds to the pseudocode statement “*Input the next grade.*” The statement displays a `prompt` dialog with the prompt `"Enter integer grade:"` on the screen.

After the user enters the `grade`, line 30 converts it from a string to an integer. We must convert the string to an integer in this example; otherwise, the addition statement in line 33 will be a string-concatenation statement rather than a numeric sum.

Next, the program updates the `total` with the new `gradeValue` entered by the user. Line 33 adds `gradeValue` to the previous value of `total` and assigns the result to `total`. This statement seems a bit strange, because it does not follow the rules of algebra. Keep in mind that JavaScript operator precedence evaluates the addition (+) operation before the assignment (=) operation. The value of the expression on the right side of the assignment operator always replaces the value of the variable on the left side of the assignment operator.

The program now is ready to increment the variable `gradeCounter` to indicate that a grade has been processed and to read the next grade from the user. Line 36 adds `1` to `gradeCounter`, so the condition in the `while` statement will eventually become `false` and terminate the loop. After this statement executes, the program continues by testing the condition in the `while` statement in line 23. If the condition is still true, the statements in lines 27–36 repeat. Otherwise the program continues execution with the first statement in sequence after the body of the loop (i.e., line 40).

Line 40 assigns the results of the average calculation to variable `average`. Lines 43–44 write a line of XHTML text in the document that displays the string `"Class average is "` followed by the value of variable `average` as an `<h1>` element in the browser.

Execute the script in a web browser by double clicking the XHTML document (from Windows Explorer). This script parses any user input as an integer. In the sample program execution in Fig. 7.7, the sum of the values entered (100, 88, 93, 55, 68, 77, 83, 95, 73 and 62) is 794. Although the treats all input as integers, the averaging calculation in the program does not produce an integer. Rather, the calculation produces a **floating-point number** (i.e., a number containing a decimal point). The average of the 10 integers input by the user in this example is 79.4.

Software Engineering Observation 7.6

If the string passed to `parseInt` *contains a floating-point numeric value,* `parseInt` *simply truncates the floating-point part. For example, the string "27.95" results in the integer 27, and the string "–123.45" results in the integer –123. If the string passed to* `parseInt` *is not a numeric value,* `parseInt` *returns* `NaN` *(not a number).*

JavaScript actually represents all numbers as floating-point numbers in memory. Floating-point numbers often develop through division, as shown in this example. When we divide 10 by 3, the result is 3.3333333…, with the sequence of 3’s repeating infinitely. The computer allocates only a fixed amount of space to hold such a value, so the stored floating-point value can be only an approximation. Although floating-point numbers are not always 100 percent precise, they have numerous applications. For example, when we speak of a “normal” body temperature of 98.6, we do not need to be precise to a large number of digits. When we view the temperature on a thermometer and read it as 98.6, it may actually be 98.5999473210643. The point here is that few applications require high-precision floating-point values, so calling this number simply 98.6 is fine for most applications.

Common Programming Error 7.7

Using floating-point numbers in a manner that assumes they are represented precisely can lead to incorrect results. Real numbers are represented only approximately by computers. For example, no fixed-size floating-point representation of π can ever be precise, because π is a transcendental number whose value cannot be expressed as digits in a finite amount of space.

7.9 Formulating Algorithms: Sentinel-Controlled Repetition

Let us generalize the class-average problem. Consider the following problem:

> *Develop a class-averaging program that will process an arbitrary number of grades each time the program is run.*

In the first class-average example, the number of grades (10) was known in advance. In this example, no indication is given of how many grades the user will enter. The program must process an arbitrary number of grades. How can the program determine when to stop the input of grades? How will it know when to calculate and display the class average?

One way to solve this problem is to use a special value called a **sentinel value** (also called a **signal value**, a **dummy value** or a **flag value**) to indicate the end of data entry. The user types in grades until all legitimate grades have been entered. Then the user types the sentinel value to indicate that the last grade has been entered. Sentinel-controlled repetition is often called **indefinite repetition**, because the number of repetitions is not known before the loop begins executing.

Clearly, one must choose a sentinel value that cannot be confused with an acceptable input value. –1 is an acceptable sentinel value for this problem because grades on a quiz are normally nonnegative integers from 0 to 100. Thus, an execution of the class-average program might process a stream of inputs such as 95, 96, 75, 74, 89 and –1. The program would compute and print the class average for the grades 95, 96, 75, 74 and 89 (–1 is the sentinel value, so it should not enter into the average calculation).

Common Programming Error 7.8

Choosing a sentinel value that is also a legitimate data value results in a logic error and may prevent a sentinel-controlled loop from terminating properly.

We approach the class-average program with a technique called **top-down, stepwise refinement**, a technique that is essential to the development of well-structured algorithms. We begin with a pseudocode representation of the **top**:

```
Determine the class average for the quiz
```

The top is a single statement that conveys the program's overall purpose. As such, the top is, in effect, a complete representation of a program. Unfortunately, the top rarely conveys sufficient detail from which to write the JavaScript algorithm. Therefore we must begin a refinement process. First, we divide the top into a series of smaller tasks and list them in the order in which they need to be performed, creating the following **first refinement**:

```
Initialize variables
Input, sum up and count the quiz grades
Calculate and print the class average
```

Here, only the sequence structure is used; the steps listed are to be executed in order, one after the other.

Software Engineering Observation 7.7

Each refinement, as well as the top itself, is a complete *specification of the algorithm; only the level of detail varies.*

To proceed to the next level of refinement (the **second refinement**), we commit to specific variables. We need a running total of the numbers, a count of how many numbers have been processed, a variable to receive the string representation of each grade as it is input, a variable to store the value of the grade after it is converted to an integer and a variable to hold the calculated average. The pseudocode statement

```
Initialize variables
```

may be refined as follows:

```
Initialize total to zero
Initialize gradeCounter to zero
```

Note that only the variables *total* and *gradeCounter* are initialized before they are used; the variables *average*, *grade* and *gradeValue* (for the calculated average, the user input and the integer representation of the *grade*, respectively) need not be initialized, because their values are determined as they are calculated or input.

The pseudocode statement

```
Input, sum up and count the quiz grades
```

requires a repetition structure (a loop) that successively inputs each grade. We do not know in advance how many grades are to be processed, so we will use sentinel-controlled repetition. The user will enter legitimate grades, one at a time. After entering the last legitimate grade, the user will enter the sentinel value. The program will test for the sentinel value after the user enters each grade and will terminate the loop when the sentinel value is encountered. The second refinement of the preceding pseudocode statement is then

```
Input the first grade (possibly the sentinel)
While the user has not as yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Input the next grade (possibly the sentinel)
```

Note that in pseudocode, we do not use braces around the pseudocode that forms the body of the *While* structure. We simply indent the pseudocode under the *While*, to show that it belongs to the body of the *While*. Remember, pseudocode is only an informal program-development aid.

The pseudocode statement

```
Calculate and print the class average
```

may be refined as follows:

```
If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the average
Else
    Print "No grades were entered"
```

Note that we are testing for the possibility of **division by zero**—a logic error that, if undetected, would cause the program to produce invalid output. The complete second refinement of the pseudocode algorithm for the class-average problem is shown in Fig. 7.8.

Error-Prevention Tip 7.1

When performing division by an expression whose value could be zero, explicitly test for this case, and handle it appropriately in your program (e.g., by printing an error message) rather than allowing the division by zero to occur.

Good Programming Practice 7.5

Include completely blank lines in pseudocode programs to make the pseudocode more readable. The blank lines separate pseudocode control structures and separate the program phases.

Software Engineering Observation 7.8

Many algorithms can be divided logically into three phases: an initialization phase that initializes the program variables, a processing phase that inputs data values and adjusts program variables accordingly, and a termination phase that calculates and prints the results.

The pseudocode algorithm in Fig. 7.8 solves the more general class-averaging problem. This algorithm was developed after only two refinements. Sometimes more refinements are necessary.

```
Initialize total to zero
Initialize gradeCounter to zero

Input the first grade (possibly the sentinel)
While the user has not as yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Input the next grade (possibly the sentinel)

If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the average
Else
    Print "No grades were entered"
```

Fig. 7.8 | Sentinel-controlled repetition to solve the class-average problem.

Software Engineering Observation 7.9

The programmer terminates the top-down, stepwise refinement process after specifying the pseudocode algorithm in sufficient detail for the programmer to convert the pseudocode to a JavaScript program. Then, implementing the JavaScript program will normally be straightforward.

Good Programming Practice 7.6

When converting a pseudocode program to JavaScript, keep the pseudocode in the JavaScript program as comments.

Software Engineering Observation 7.10

Experience has shown that the most difficult part of solving a problem on a computer is developing the algorithm for the solution. Once a correct algorithm is specified, the process of producing a working JavaScript program from the algorithm is normally straightforward.

Software Engineering Observation 7.11

Many experienced programmers write programs without ever using program-development tools like pseudocode. As they see it, their ultimate goal is to solve the problem on a computer, and writing pseudocode merely delays the production of final outputs. Although this approach may work for simple and familiar problems, it can lead to serious errors in large, complex projects.

Figure 7.9 shows the JavaScript program and a sample execution. Although each grade is an integer, the averaging calculation is likely to produce a number with a decimal point (a real number).

In this example, we see that control structures may be stacked on top of one another (in sequence) just as a child stacks building blocks. The `while` statement (lines 31–45) is followed immediately by an `if...else` statement (lines 48–57) in sequence. Much of the code in this program is identical to the code in Fig. 7.7, so we concentrate in this example on the new features.

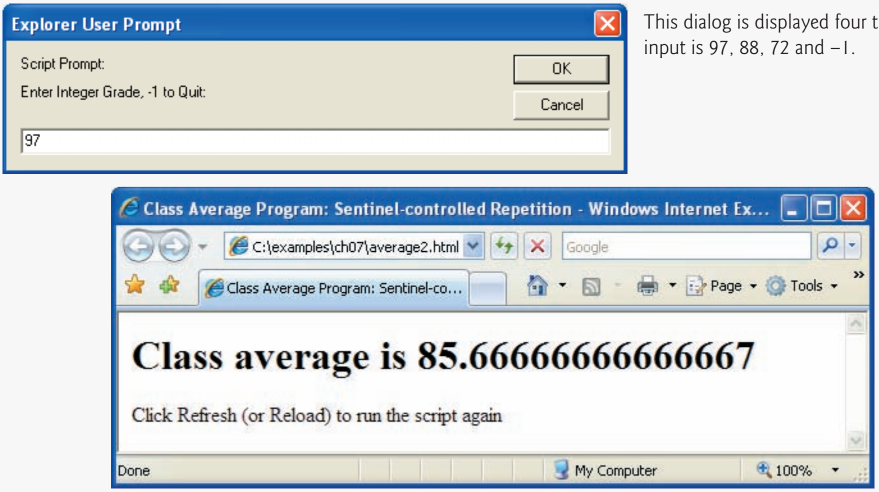
Line 21 initializes `gradeCounter` to `0`, because no grades have been entered yet. Remember that the program uses sentinel-controlled repetition. To keep an accurate record of the number of grades entered, the script increments `gradeCounter` only after processing a valid grade value.

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 7.9: average2.html -->
 6  <!-- Sentinel-controlled repetition to calculate a class average. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Class Average Program: Sentinel-controlled Repetition</title>
10
11        <script type = "text/javascript">
12           <!--
```

Fig. 7.9 | Sentinel-controlled repetition to calculate a class average. (Part 1 of 3.)

```
         var total; // sum of grades
         var gradeCounter; // number of grades entered
         var grade; // grade typed by user (as a string)
         var gradeValue; // grade value (converted to integer)
         var average; // average of all grades

         // Initialization phase
         total = 0; // clear total
         gradeCounter = 0; // prepare to loop

         // Processing phase
         // prompt for input and read grade from user
         grade = window.prompt(
              "Enter Integer Grade, -1 to Quit:", "0" );

         // convert grade from a string to an integer
         gradeValue = parseInt( grade );

         while ( gradeValue != -1 )
         {
            // add gradeValue to total
            total = total + gradeValue;

            // add 1 to gradeCounter
            gradeCounter = gradeCounter + 1;

            // prompt for input and read grade from user
            grade = window.prompt(
               "Enter Integer Grade, -1 to Quit:", "0" );

            // convert grade from a string to an integer
            gradeValue = parseInt( grade );
         } // end while

         // Termination phase
         if ( gradeCounter != 0 )
         {
            average = total / gradeCounter;

            // display average of exam grades
            document.writeln(
               "<h1>Class average is " + average + "</h1>" );
         } // end if
         else
            document.writeln( "<p>No grades were entered</p>" );
         // -->
      </script>
   </head>
   <body>
      <p>Click Refresh (or Reload) to run the script again</p>
   </body>
</html>
```

Fig. 7.9 | Sentinel-controlled repetition to calculate a class average. (Part 2 of 3.)


Explorer User Prompt
OK
Cancel
Class Average Program: Sentinel-controlled Repetition - Windows Internet Ex...
C:\examples\ch07\average2.html
Google
Class Average Program: Sentinel-co...
Page
Tools
Done
My Computer
100%

Script Prompt:
Enter Integer Grade, -1 to Quit:
97

Class average is 85.6666666666667

Click Refresh (or Reload) to run the script again

This dialog is displayed four times. User input is 97, 88, 72 and –1.

Fig. 7.9 | Sentinel-controlled repetition to calculate a class average. (Part 3 of 3.)

Note the difference in program logic for sentinel-controlled repetition as compared with the counter-controlled repetition in Fig. 7.7. In counter-controlled repetition, we read a value from the user during each iteration of the `while` statement's body for the specified number of iterations. In sentinel-controlled repetition, we read one value (lines 25–26) and convert it to an integer (line 29) before the program reaches the `while` statement. The script uses this value to determine whether the program's flow of control should enter the body of the `while` statement. If the `while` statement's condition is `false` (i.e., the user typed the sentinel as the first grade), the script ignores the body of the `while` statement (i.e., no grades were entered). If the condition is `true`, the body begins execution and processes the value entered by the user (i.e., adds the value to the `total` in line 34). After processing the value, the script increments `gradeCounter` by 1 (line 37), inputs the next `grade` from the user (lines 40–41) and converts the `grade` to an integer (line 44), before the end of the `while` statement's body. When the script reaches the closing right brace (`}`) of the body in line 45, execution continues with the next test of the condition of the `while` statement (line 31), using the new value just entered by the user to determine whether the `while` statement's body should execute again. Note that the next value always is input from the user immediately before the script evaluates the condition of the `while` statement. This order allows us to determine whether the value just entered by the user is the sentinel value *before* processing it (i.e., adding it to the `total`). If the value entered is the sentinel value, the `while` statement terminates and the script does not add the value to the `total`.

Good Programming Practice 7.7

In a sentinel-controlled loop, the prompts requesting data entry should explicitly remind the user what the sentinel value is.

Note the block in the `while` loop in Fig. 7.9 (lines 32–45). Without the braces, the last three statements in the body of the loop would fall outside of the loop, causing the computer to interpret the code incorrectly, as follows:

```
while ( gradeValue != -1 )
   // add gradeValue to total
   total = total + gradeValue;

// add 1 to gradeCounter
gradeCounter = gradeCounter + 1;

// prompt for input and read grade from user
grade = window.prompt(
   "Enter Integer Grade, -1 to Quit:", "0" );

// convert grade from a string to an integer
gradeValue = parseInt( grade );
```

This interpretation would cause an infinite loop in the program if the user does not input the sentinel `-1` as the first input value in lines 25–26 (i.e., before the `while` statement).

Common Programming Error 7.9

Omitting the braces that delineate a block can lead to logic errors such as infinite loops.

7.10 Formulating Algorithms: Nested Control Statements

Let us work through another complete problem. We once again formulate the algorithm using pseudocode and top-down, stepwise refinement, and write a corresponding JavaScript program.

Consider the following problem statement:

> *A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, several of the students who completed this course took the licensing exam. Naturally, the college wants to know how well its students performed. You have been asked to write a program to summarize the results. You have been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam and a 2 if the student failed.*
>
> *Your program should analyze the results of the exam as follows:*
>
> 1. *Input each test result (i.e., a 1 or a 2). Display the message "Enter result" on the screen each time the program requests another test result.*
> 2. *Count the number of test results of each type.*
> 3. *Display a summary of the test results indicating the number of students who passed and the number of students who failed.*
> 4. *If more than eight students passed the exam, print the message "Raise tuition."*

After reading the problem statement carefully, we make the following observations about the problem:

1. The program must process test results for 10 students. A counter-controlled loop will be used.
2. Each test result is a number—either a 1 or a 2. Each time the program reads a test result, the program must determine whether the number is a 1 or a 2. We test for a 1 in our algorithm. If the number is not a 1, we assume that it is a 2. (An exercise at the end of the chapter considers the consequences of this assumption.)

3. Two counters are used to keep track of the exam results—one to count the number of students who passed the exam and one to count the number of students who failed the exam.

After the program processes all the results, it must decide whether more than eight students passed the exam. Let us proceed with top-down, stepwise refinement. We begin with a pseudocode representation of the top:

```
Analyze exam results and decide whether tuition should be raised
```

Once again, it is important to emphasize that the top is a complete representation of the program, but that several refinements are necessary before the pseudocode can be evolved naturally into a JavaScript program. Our first refinement is as follows:

```
Initialize variables
Input the ten exam grades and count passes and failures
Print a summary of the exam results and decide whether tuition should be raised
```

Here, too, even though we have a complete representation of the entire program, further refinement is necessary. We now commit to specific variables. Counters are needed to record the passes and failures; a counter will be used to control the looping process, and a variable is needed to store the user input. The pseudocode statement

```
Initialize variables
```

may be refined as follows:

```
Initialize passes to zero
Initialize failures to zero
Initialize student to one
```

Note that only the counters for the number of passes, the number of failures and the number of students are initialized. The pseudocode statement

```
Input the ten quiz grades and count passes and failures
```

requires a loop that successively inputs the result of each exam. Here, it is known in advance that there are precisely 10 exam results, so counter-controlled looping is appropriate. Inside the loop (i.e., *nested* within the loop), a double-selection structure will determine whether each exam result is a pass or a failure and will increment the appropriate counter accordingly. The refinement of the preceding pseudocode statement is then

```
While student counter is less than or equal to ten
    Input the next exam result

    If the student passed
        Add one to passes
    Else
        Add one to failures

    Add one to student counter
```

Note the use of blank lines to set off the *If…Else* control structure to improve program readability. The pseudocode statement

```
Print a summary of the exam results and decide whether tuition should be raised
```

may be refined as follows:

```
Print the number of passes
Print the number of failures
If more than eight students passed
    Print “Raise tuition”
```

The complete second refinement appears in Fig. 7.10. Note that blank lines are also used to set off the *While* statement for program readability.

This pseudocode is now refined sufficiently for conversion to JavaScript. The JavaScript program and two sample executions are shown in Fig. 7.11.

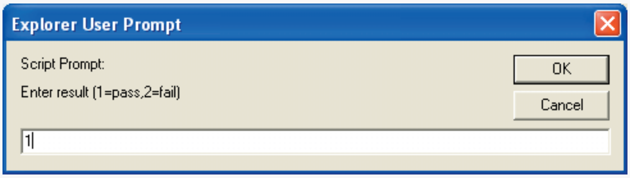
```
Initialize passes to zero
Initialize failures to zero
Initialize student to one

While student counter is less than or equal to ten
    Input the next exam result

    If the student passed
        Add one to passes
    Else
        Add one to failures

    Add one to student counter

Print the number of passes
Print the number of failures
If more than eight students passed
    Print “Raise tuition”
```

Fig. 7.10 | Examination-results problem pseudocode.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 7.11: analysis.html -->
<!-- Examination-results calculation. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Analysis of Examination Results</title>
      <script type = "text/javascript">
         <!--
         // initializing variables in declarations
         var passes = 0; // number of passes
         var failures = 0; // number of failures
         var student = 1; // student counter
         var result; // one exam result

```

Fig. 7.11 | Examination-results calculation. (Part 1 of 3.)

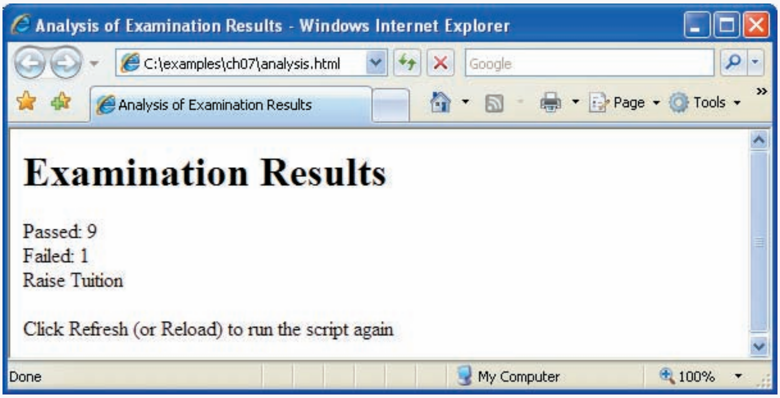
```
         // process 10 students; counter-controlled loop
         while ( student <= 10 )
         {
            result = window.prompt( "Enter result (1=pass,2=fail)", "0" );

            if ( result == "1" )
               passes = passes + 1;
            else
               failures = failures + 1;

            student = student + 1;
         } // end while

         // termination phase
         document.writeln( "<h1>Examination Results</h1>" );
         document.writeln(
            "Passed: " + passes + "<br />Failed: " + failures );

         if ( passes > 8 )
            document.writeln( "<br />Raise Tuition" );
         // -->
      </script>
   </head>
   <body>
      <p>Click Refresh (or Reload) to run the script again</p>
   </body>
</html>
```



Script Prompt:
Enter result (1=pass,2=fail)

1

This dialog is displayed 10 times. User input is 1, 2, 1, 1, 1, 1, 1, 1, 1 and 1.

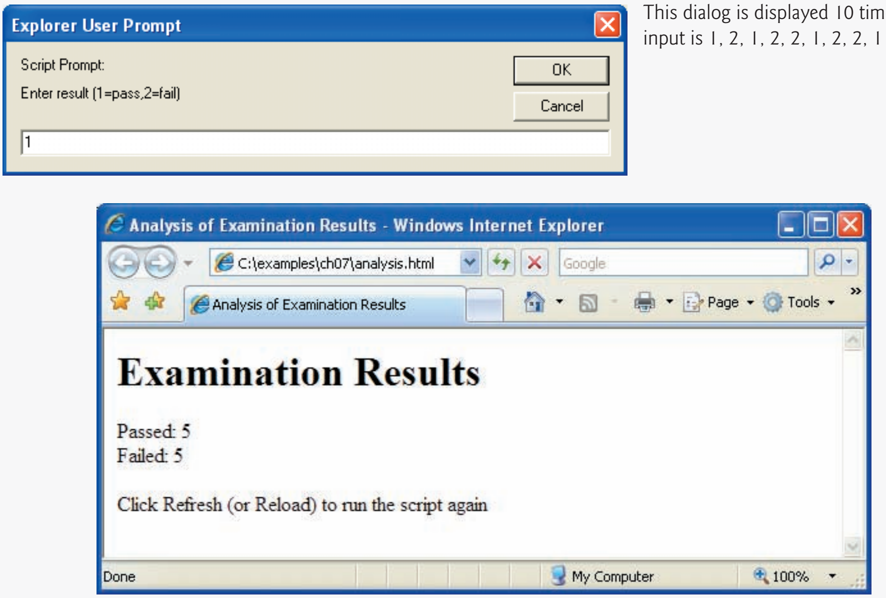

Analysis of Examination Results - Windows Internet Explorer
C:\examples\ch07\analysis.html
Google
Analysis of Examination Results
Page
Tools
Done
My Computer
100%

Examination Results

Passed: 9
Failed: 1
Raise Tuition

Click Refresh (or Reload) to run the script again

Fig. 7.11 | Examination-results calculation. (Part 2 of 3.)



This dialog is displayed 10 times. User input is 1, 2, 1, 2, 2, 1, 2, 2, 1 and 1.

Script Prompt:
Enter result (1=pass,2=fail)

```
1
```

Examination Results

Passed: 5
Failed: 5

Click Refresh (or Reload) to run the script again

Explorer User Prompt
OK
Cancel
Analysis of Examination Results - Windows Internet Explorer
C:\examples\ch07\analysis.html
Google
Analysis of Examination Results
Page
Tools
Done
My Computer
100%

Fig. 7.11 | Examination-results calculation. (Part 3 of 3.)

Lines 13–16 declare the variables used to process the examination results. Note that JavaScript allows variable initialization to be incorporated into declarations (`passes` is assigned `0`, `failures` is assigned `0` and `student` is assigned `1`). Some programs may require reinitialization at the beginning of each repetition; such reinitialization would normally occur in assignment statements.

The processing of the exam results occurs in the `while` statement in lines 19–29. Note that the `if...else` statement in lines 23–26 in the loop tests only whether the exam result was 1; it assumes that all other exam results are 2. Normally, you should validate the values input by the user (i.e., determine whether the values are correct). In the exercises, we ask you to modify this example to validate the input values to ensure that they are either 1 or 2.

Good Programming Practice 7.8

When inputting values from the user, validate the input to ensure that it is correct. If an input value is incorrect, prompt the user to input the value again.

7.11 Assignment Operators

JavaScript provides several assignment operators (called **compound assignment operators**) for abbreviating assignment expressions. For example, the statement

```
c = c + 3;
```

can be abbreviated with the **addition assignment operator**, **+=**, as

```
c += 3;
```

The += operator adds the value of the expression on the right of the operator to the value of the variable on the left of the operator and stores the result in the variable on the left of the operator. Any statement of the form

variable = *variable operator expression*;

where *operator* is one of the binary operators +, -, *, / or % (or others we will discuss later in the text), can be written in the form

variable operator = *expression*;

Thus, the assignment c += 3 adds 3 to c. Figure 7.12 shows the arithmetic assignment operators, sample expressions using these operators and explanations of the meaning of the operators.

Performance Tip 7.1

Programmers can write programs that execute a bit faster when the arithmetic assignment operators are used, because the variable on the left side of the assignment does not have to be evaluated twice.

Performance Tip 7.2

Many of the performance tips we mention in this text result in only nominal improvements, so the reader may be tempted to ignore them. Significant performance improvement often is realized when a supposedly nominal improvement is placed in a loop that may repeat a large number of times.

Assignment operator	Initial value of variable	Sample expression	Explanation	Assigns
+=	c = 3	c += 7	c = c + 7	10 to c
-=	d = 5	d -= 4	d = d - 4	1 to d
*=	e = 4	e *= 5	e = e * 5	20 to e
/=	f = 6	f /= 3	f = f / 3	2 to f
%=	g = 12	g %= 9	g = g % 9	3 to g

Fig. 7.12 | Arithmetic assignment operators.

7.12 Increment and Decrement Operators

JavaScript provides the unary **increment operator** (++) and **decrement operator** (--) (summarized in Fig. 7.13). If a variable c is incremented by 1, the increment operator, ++, can be used rather than the expression c = c + 1 or c += 1. If an increment or decrement operator is placed before a variable, it is referred to as the **preincrement** or **predecrement operator**, respectively. If an increment or decrement operator is placed after a variable, it is referred to as the **postincrement** or **postdecrement operator**, respectively.

Operator	Example	Called	Explanation
`++`	`++a`	preincrement	Increment `a` by 1, then use the new value of `a` in the expression in which `a` resides.
`++`	`a++`	postincrement	Use the current value of `a` in the expression in which `a` resides, then increment `a` by 1.
`--`	`--b`	predecrement	Decrement `b` by 1, then use the new value of `b` in the expression in which `b` resides.
`--`	`b--`	postdecrement	Use the current value of `b` in the expression in which `b` resides, then decrement `b` by 1.

Fig. 7.13 | Increment and decrement operators.

Error-Prevention Tip 7.2

The predecrement and postdecrement JavaScript operators cause the W3C XHTML Validator to incorrectly report errors. The validator attempts to interpret the decrement operator as part of an XHTML comment tag (`<!--` or `-->`). You can avoid this problem by using the subtraction assignment operator (`-=`) to subtract one from a variable. Note that the validator may report many more (nonexistent) errors once it improperly parses the decrement operator.

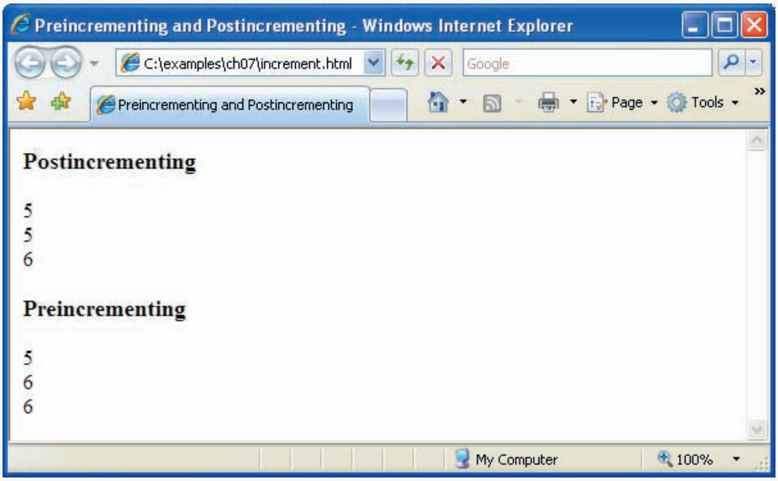
Preincrementing (or predecrementing) a variable causes the program to increment (decrement) the variable by 1, then use the new value of the variable in the expression in which it appears. Postincrementing (postdecrementing) the variable causes the program to use the current value of the variable in the expression in which it appears, then increment (decrement) the variable by 1.

The script in Fig. 7.14 demonstrates the difference between the preincrementing version and the postincrementing version of the `++` increment operator. Postincrementing the variable `c` causes it to be incremented after it is used in the `document.writeln` method call (line 18). Preincrementing the variable `c` causes it to be incremented before it is used in the `document.writeln` method call (line 25). The program displays the value of `c` before and after the `++` operator is used. The decrement operator (`--`) works similarly.

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 7.14: increment.html -->
 6  <!-- Preincrementing and Postincrementing. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Preincrementing and Postincrementing</title>
10        <script type = "text/javascript">
11           <!--
12           var c;
```

Fig. 7.14 | Preincrementing and postincrementing. (Part 1 of 2.)

```

         c = 5;
         document.writeln( "<h3>Postincrementing</h3>" );
         document.writeln( c ); // prints 5
         // prints 5 then increments
         document.writeln( "<br />" + c++ );
         document.writeln( "<br />" + c ); // prints 6

         c = 5;
         document.writeln( "<h3>Preincrementing</h3>" );
         document.writeln( c ); // prints 5
         // increments then prints 6
         document.writeln( "<br />" + ++c );
         document.writeln( "<br />" + c ); // prints 6
         // -->
      </script>
   </head><body></body>
</html>
```


Preincrementing and Postincrementing - Windows Internet Explorer
C:\examples\ch07\increment.html
Google
Preincrementing and Postincrementing
Page
Tools
My Computer
100%

```
Postincrementing

5
5
6

Preincrementing

5
6
6
```

Fig. 7.14 | Preincrementing and postincrementing. (Part 2 of 2.)

Good Programming Practice 7.9

For readability, unary operators should be placed next to their operands, with no intervening spaces.

The three assignment statements in Fig. 7.11 (lines 24, 26 and 28, respectively),

```
passes = passes + 1;
failures = failures + 1;
student = student + 1;
```

can be written more concisely with assignment operators as

```
passes += 1;
failures += 1;
student += 1;
```

with preincrement operators as

```
++passes;
++failures;
++student;
```

or with postincrement operators as

```
passes++;
failures++;
student++;
```

It is important to note here that when incrementing or decrementing a variable in a statement by itself, the preincrement and postincrement forms have the same effect, and the predecrement and postdecrement forms have the same effect. It is only when a variable appears in the context of a larger expression that preincrementing the variable and postincrementing the variable have different effects. Predecrementing and postdecrementing behave similarly.

Common Programming Error 7.10

Attempting to use the increment or decrement operator on an expression other than a **left-hand-side expression**—*commonly called an* **lvalue**—*is a syntax error. A left-hand-side expression is a variable or expression that can appear on the left side of an assignment operation. For example, writing* `++(x + 1)` *is a syntax error, because* `(x + 1)` *is not a left-hand-side expression.*

Figure 7.15 lists the precedence and associativity of the operators introduced to this point. The operators are shown top to bottom in decreasing order of precedence. The second column describes the associativity of the operators at each level of precedence. Notice that the conditional operator (`?:`), the unary operators increment (`++`) and decrement (`--`) and the assignment operators `=`, `+=`, `-=`, `*=`, `/=` and `%=` associate from right to left. All other operators in the operator precedence table (Fig. 7.15) associate from left to right. The third column names the groups of operators.

Operator	Associativity	Type
`++ --`	right to left	unary
`* / %`	left to right	multiplicative
`+ -`	left to right	additive
`< <= > >=`	left to right	relational
`== !=`	left to right	equality
`?:`	right to left	conditional
`= += -= *= /= %=`	right to left	assignment

Fig. 7.15 | Precedence and associativity of the operators discussed so far.

7.13 Wrap-Up

In this chapter, we introduced the concept of algorithms, and explained how to introduce structure into your programs. We used pseudocode and flowcharts to represent algorithms and demonstrated how to translate them into control structures, which form the basis of all programs. We explored selection and repetition statements, and how to integrate them to make decisions and repeat statements in your code. We also introduced assignment, increment and decrement operators, as well as different types of errors that can result from incorrect implementation of control statements.

7.14 Web Resources

`www.deitel.com/javascript/`

The Deitel JavaScript Resource Center contains links to some of the best JavaScript resources on the web. There you'll find categorized links to JavaScript tools, code generators, forums, books, libraries, frameworks and more. Also check out the tutorials for all skill levels, from introductory to advanced. Be sure to visit the related Resource Centers on XHTML (`www.deitel.com/xhtml/`) and CSS 2.1 (`www.deitel.com/css21/`).

Summary

Section 7.2 Algorithms

- Any computable problem can be solved by executing a series of actions in a specific order.
- A procedure for solving a problem in terms of the actions to execute and the order in which the actions are to execute is called an algorithm.
- Specifying the order in which statements are to be executed in a computer program is called program control.

Section 7.3 Pseudocode

- Pseudocode is an artificial and informal language that helps programmers develop algorithms.
- Carefully prepared pseudocode may be converted easily to a corresponding JavaScript program.
- Pseudocode normally describes only executable statements—the actions that are performed when the program is converted from pseudocode to JavaScript and executed.

Section 7.4 Control Structures

- Normally, statements in a program execute one after the other, in the order in which they are written. This process is called sequential execution.
- Various JavaScript statements enable the programmer to specify that the next statement to be executed may be other than the next one in sequence. This process is called transfer of control.
- All programs can be written in terms of only three control structures, namely, the sequence structure, the selection structure and the repetition structure.
- A flowchart is a graphical representation of an algorithm or of a portion of an algorithm. Flowcharts are drawn using certain special-purpose symbols, such as rectangles, diamonds, ovals and small circles; these symbols are connected by arrows called flowlines, which indicate the order in which the actions of the algorithm execute.
- JavaScript provides three selection structures. The `if` statement either performs (selects) an action if a condition is true or skips the action if the condition is false. The `if…else` statement performs an action if a condition is true and performs a different action if the condition is false.

The `switch` statement performs one of many different actions, depending on the value of an expression.

- JavaScript provides four repetition statements, namely, `while`, `do…while`, `for` and `for…in`.
- Keywords cannot be used as identifiers (e.g., for variable names).
- Single-entry/single-exit control structures make it easy to build programs. Control structures are attached to one another by connecting the exit point of one control structure to the entry point of the next. This procedure is called control-structure stacking. There is only one other way control structures may be connected: control-structure nesting.

Section 7.5 `if` Selection Statement

- The JavaScript interpreter ignores white-space characters: blanks, tabs and newlines used for indentation and vertical spacing. Programmers insert white-space characters to enhance program clarity.
- A decision can be made on any expression that evaluates to a value of JavaScript's boolean type (i.e., any expression that evaluates to `true` or `false`).
- The indentation convention you choose should be carefully applied throughout your programs. It is difficult to read programs that do not use uniform spacing conventions.

Section 7.6 `if…else` Selection Statement

- The conditional operator (`?:`) is closely related to the `if…else` statement. Operator `?:` is JavaScript's only ternary operator—it takes three operands. The operands together with the `?:` operator form a conditional expression. The first operand is a boolean expression, the second is the value for the conditional expression if the boolean expression evaluates to true and the third is the value for the conditional expression if the boolean expression evaluates to false.
- Nested `if…else` statements test for multiple cases by placing `if…else` statements inside other `if…else` structures.
- The JavaScript interpreter always associates an `else` with the previous `if`, unless told to do otherwise by the placement of braces (`{}`).
- The `if` selection statement expects only one statement in its body. To include several statements in the body of an `if` statement, enclose the statements in braces (`{` and `}`). A set of statements contained within a pair of braces is called a block.
- A logic error has its effect at execution time. A fatal logic error causes a program to fail and terminate prematurely. A nonfatal logic error allows a program to continue executing, but the program produces incorrect results.

Section 7.7 `while` Repetition Statement

- The `while` repetition structure allows the programmer to specify that an action is to be repeated while some condition remains true.

Section 7.8 Formulating Algorithms: Counter-Controlled Repetition

- Counter-controlled repetition is often called definite repetition, because the number of repetitions is known before the loop begins executing.
- Uninitialized variables used in mathematical calculations result in logic errors and produce the value `NaN` (not a number).
- JavaScript represents all numbers as floating-point numbers in memory. Floating-point numbers often develop through division. The computer allocates only a fixed amount of space to hold such a value, so the stored floating-point value can only be an approximation.

Section 7.9 Formulating Algorithms: Sentinel-Controlled Repetition

- In sentinel-controlled repetition, a special value called a sentinel value (also called a signal value, a dummy value or a flag value) indicates the end of data entry. Sentinel-controlled repetition often is called indefinite repetition, because the number of repetitions is not known in advance.
- It is necessary to choose a sentinel value that cannot be confused with an acceptable input value.

Section 7.10 Formulating Algorithms: Nested Control Statements.

- Top-down, stepwise refinement is a technique essential to the development of well-structured algorithms. The top is a single statement that conveys the overall purpose of the program. As such, the top is, in effect, a complete representation of a program. The stepwise refinement process divides the top into a series of smaller tasks. The programmer terminates the top-down, stepwise refinement process when the pseudocode algorithm is specified in sufficient detail for the programmer to be able to convert the pseudocode to a JavaScript program.

Section 7.11 Assignment Operators

- JavaScript provides the arithmetic assignment operators `+=`, `-=`, `*=`, `/=` and `%=`, which abbreviate certain common types of expressions.

Section 7.12 Increment and Decrement Operators

- The increment operator, `++`, and the decrement operator, `--`, increment or decrement a variable by 1, respectively. If the operator is prefixed to the variable, the variable is incremented or decremented by 1, then used in its expression. If the operator is postfixed to the variable, the variable is used in its expression, then incremented or decremented by 1.

Terminology

`--` operator
`?:` operator
`++` operator
action symbol
action/decision model
algorithm
arithmetic assignment operators:
    `+=`, `-=`, `*=`, `/=` and `%=`
arrow
block
body of a loop
boolean expression
braces (`{}`)
compound assignment operator
conditional expression
conditional operator (`?:`)
connector symbol
control structure
control-structure nesting
control-structure stacking
counter
counter-controlled repetition
dangling-else problem
decision symbol
decrement operator (`--`)
definite repetition
diamond symbol
division by zero
double-selection structure
dummy value
empty statement (`;`)
fatal logic error
first refinement
flag value
floating-point number
flowchart
flowlines
`goto` elimination
`goto` statement
`if` selection statement
`if...else` selection statement
increment operator (`++`)
indefinite repetition
infinite loop
initialization
initialization phase
keyword
left-hand-side expression
logic error
loop

loop-continuation condition
lvalue
multiple-selection structure
NaN (not a number)
nested control structures
nonfatal logic error
null
oval symbol
postdecrement operator
postincrement operator
predecrement operator
preincrement operator
procedure
processing phase
program control
pseudocode
rectangle symbol
repetition
repetition structure
second refinement
selection
selection structure
sentinel value
sentinel-controlled repetition
sequence structure
sequential execution
signal value
single-entry/single-exit control structure
single-selection structure
small circle symbol
stacked control structures
statement
structured programming
syntax error
terminate a loop
termination phase
ternary operator
top
top-down, stepwise refinement
transfer of control
unary operator
while repetition statement
white-space character

Self-Review Exercises

7.1 Fill in the blanks in each of the following statements:

a) All programs can be written in terms of three types of control structures: ________, ________ and ________.

b) The ________ double-selection statement is used to execute one action when a condition is true and another action when that condition is false.

c) Repeating a set of instructions a specific number of times is called ________ repetition.

d) When it is not known in advance how many times a set of statements will be repeated, a(n) ________ (or a(n) ________, ________ or ________) value can be used to terminate the repetition.

7.2 Write four JavaScript statements that each add 1 to variable x, which contains a number.

7.3 Write JavaScript statements to accomplish each of the following tasks:

a) Assign the sum of x and y to z, and increment the value of x by 1 after the calculation. Use only one statement.

b) Test whether the value of the variable count is greater than 10. If it is, print "Count is greater than 10".

c) Decrement the variable x by 1, then subtract it from the variable total. Use only one statement.

d) Calculate the remainder after q is divided by divisor, and assign the result to q. Write this statement in two different ways.

7.4 Write a JavaScript statement to accomplish each of the following tasks:

a) Declare variables sum and x.

b) Assign 1 to variable x.

c) Assign 0 to variable sum.

d) Add variable x to variable sum, and assign the result to variable sum.

e) Print "The sum is: ", followed by the value of variable sum.

7.5 Combine the statements that you wrote in Exercise 7.4 into a JavaScript program that calculates and prints the sum of the integers from 1 to 10. Use the `while` statement to loop through the calculation and increment statements. The loop should terminate when the value of `x` becomes 11.

7.6 Determine the value of each variable after the calculation is performed. Assume that, when each statement begins executing, all variables have the integer value 5.

a) `product *= x++;`
b) `quotient /= ++x;`

7.7 Identify and correct the errors in each of the following segments of code:

a)
```
while ( c <= 5 ) {
   product *= c;
   ++c;
```
b)
```
if ( gender == 1 )
   document.writeln( "Woman" );
else;
   document.writeln( "Man" );
```

7.8 What is wrong with the following `while` repetition statement?

```
while ( z >= 0 )
    sum += z;
```

Answers to Self-Review Exercises

7.1 a) Sequence, selection and repetition. b) `if…else`. c) Counter-controlled (or definite). d) Sentinel, signal, flag or dummy.

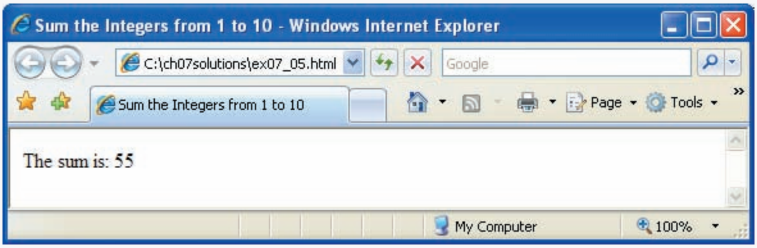
7.2
```
x = x + 1;
x += 1;
++x;
x++;
```

7.3 a) `z = x++ + y;`
b)
```
if ( count > 10 )
   document.writeln( "Count is greater than 10" );
```
c) `total -= --x;`
d)
```
q %= divisor;
q = q % divisor;
```

7.4 a) `var sum, x;`
b) `x = 1;`
c) `sum = 0;`
d) `sum += x;` or `sum = sum + x;`
e) `document.writeln( "The sum is: " + sum );`

7.5 The solution is as follows:

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 7.5: ex07_05.html -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head><title>Sum the Integers from 1 to 10</title>
      <script type = "text/javascript">
         <!--
         var sum; // stores the total
          var x; // counter control variable

         x = 1;
         sum = 0;

         while ( x <= 10 )
         {
            sum += x;
            ++x;
         } // end while

         document.writeln( "The sum is: " + sum );
         // -->
      </script>
   </head><body></body>
</html>
```

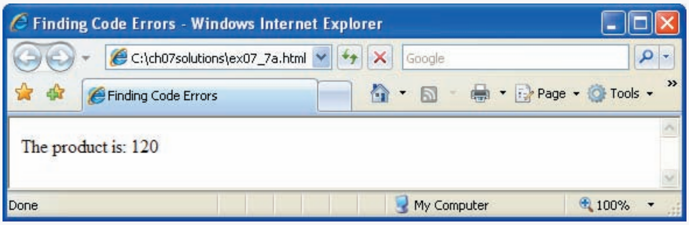

Sum the Integers from 1 to 10 - Windows Internet Explorer
C:\ch07solutions\ex07_05.html
Google
Sum the Integers from 1 to 10
Page
Tools
My Computer
100%

The sum is: 55

7.6 a) `product = 25, x = 6;`
b) `quotient = 0.833333..., x = 6;`

7.7 a) Error: Missing the closing right brace of the `while` body.
Correction: Add closing right brace after the statement `++c;`.

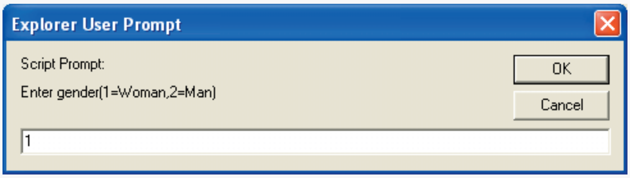
```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 7.7a: ex07_07a.html -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Finding Code Errors</title>
      <script type = "text/javascript">
         <!--
         var c;
         var product;

         c = 1;
         product = 1;

         while ( c <= 5 )
         {
            product *= c;
            ++c;
         } // end while

         document.writeln( "The product is: " + product );
         // -->
      </script>
   </head><body></body>
</html>
```


Finding Code Errors - Windows Internet Explorer
C:\ch07solutions\ex07_7a.html
Google
Finding Code Errors
Page
Tools
Done
My Computer
100%

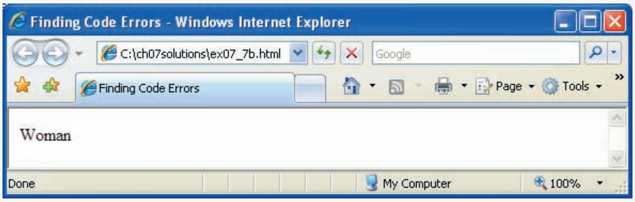
The product is: 120

b) Error: The semicolon after `else` results in a logic error. The second output statement will always be executed.
Correction: Remove the semicolon after `else`.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 7.7b: ex07_07b.html -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Finding Code Errors</title>
      <script type = "text/javascript">
         <!--
         var gender;
         gender = window.prompt( "Enter gender"
               + "(1=Woman,2=Man)", "1" );

         if ( gender == 1 )
            document.writeln( "Woman" );
         else
            document.writeln( "Man" );
         // -->
      </script>
   </head><body></body>
</html>
```



Script Prompt:
Enter gender(1=Woman,2=Man)
1


Finding Code Errors - Windows Internet Explorer
C:\ch07solutions\ex07_7b.html
Google
Finding Code Errors
Page
Tools
Done
My Computer
100%

Woman

7.8 The value of the variable `z` is never changed in the body of the `while` statement. Therefore, if the loop-continuation condition `( z >= 0 )` is true, an infinite loop is created. To prevent the creation of the infinite loop, `z` must be decremented so that it eventually becomes less than 0.

Exercises

7.9 Identify and correct the errors in each of the following segments of code [*Note:* There may be more than one error in each piece of code]:

a)
```
if ( age >= 65 );
   document.writeln( "Age greater than or equal to 65" );
else
   document.writeln( "Age is less than 65" );
```

b)
```
var x = 1, total;
while ( x <= 10 )
{
   total += x;
   ++x;
}
```

c)
```
var x = 1;
var total = 0;
While ( x <= 100 )
   total += x;
   ++x;
```

d)
```
var y = 5;
while ( y > 0 )
{
   document.writeln( y );
   ++y;
```

7.10 What does the following program print?

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 7.10: ex07_10.html -->
<html xmlns="http://www.w3.org/1999/xhtml">
   <head><title>Mystery Script</title>
      <script type = "text/javascript">
         <!--
         var y;
         var x = 1;
         var total = 0;

         while ( x <= 10 )
         {
            y = x * x;
            document.writeln( y + "<br />" );
            total += y;
            ++x;
         } // end while

         document.writeln( "<br />Total is " + total );
         // -->
      </script>
   </head><body></body>
</html>
```

For Exercises 7.11–7.14, perform each of the following steps:

a) Read the problem statement.
b) Formulate the algorithm using pseudocode and top-down, stepwise refinement.
c) Write a JavaScript program.
d) Test, debug and execute the JavaScript program.
e) Process three complete sets of data.

7.11 Drivers are concerned with the mileage obtained by their automobiles. One driver has kept track of several tankfuls of gasoline by recording the number of miles driven and the number of gallons used for each tankful. Develop a JavaScript program that will take as input the miles driven and gallons used (both as integers) for each tankful. The program should calculate and output XHTML text that displays the number of miles per gallon obtained for each tankful and prints the combined number of miles per gallon obtained for all tankfuls up to this point. Use `prompt` dialogs to obtain the data from the user.

7.12 Develop a JavaScript program that will determine whether a department-store customer has exceeded the credit limit on a charge account. For each customer, the following facts are available:

a) Account number
b) Balance at the beginning of the month
c) Total of all items charged by this customer this month
d) Total of all credits applied to this customer's account this month
e) Allowed credit limit

The program should input each of these facts from a `prompt` dialog as an integer, calculate the new balance (= *beginning balance* + *charges* – *credits*), display the new balance and determine whether the new balance exceeds the customer's credit limit. For customers whose credit limit is exceeded, the program should output XHTML text that displays the message “Credit limit exceeded.”

7.13 A large company pays its salespeople on a commission basis. The salespeople receive $200 per week, plus 9 percent of their gross sales for that week. For example, a salesperson who sells $5000 worth of merchandise in a week receives $200 plus 9 percent of $5000, or a total of $650. You have been supplied with a list of the items sold by each salesperson. The values of these items are as follows:

Item	Value
1	239.99
2	129.75
3	99.95
4	350.89

Develop a program that inputs one salesperson’s items sold for last week, calculates the salesperson's earnings and outputs XHTML text that displays the salesperson's earnings.

7.14 Develop a JavaScript program that will determine the gross pay for each of three employees. The company pays “straight time” for the first 40 hours worked by each employee and pays “time and a half” for all hours worked in excess of 40 hours. You are given a list of the employees of the company, the number of hours each employee worked last week and the hourly rate of each employee. Your program should input this information for each employee, determine the employee’s gross pay and output XHTML text that displays the employee's gross pay. Use `prompt` dialogs to input the data.

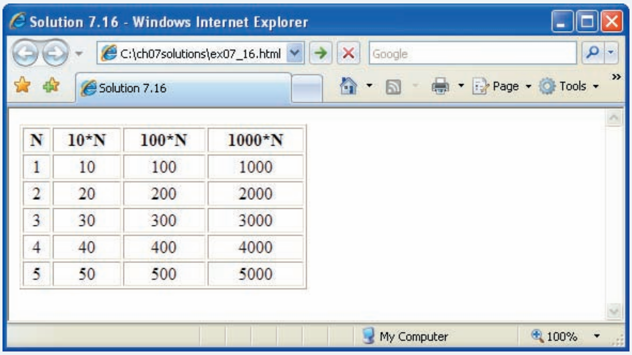
7.15 The process of finding the largest value (i.e., the maximum of a group of values) is used frequently in computer applications. For example, a program that determines the winner of a sales contest would input the number of units sold by each salesperson. The salesperson who sells the most units wins the contest. Write a pseudocode program and then a JavaScript program that inputs a series of 10 single-digit numbers as characters, determines the largest of the numbers and outputs a message that displays the largest number. Your program should use three variables as follows:

a) `counter`: A counter to count to 10 (i.e., to keep track of how many numbers have been input and to determine when all 10 numbers have been processed);

b) `number`: The current digit input to the program;

c) `largest`: The largest number found so far.

7.16 Write a JavaScript program that uses looping to print the following table of values. Output the results in an XHTML table. Use CSS to center the data in each column.


Solution 7.16 - Windows Internet Explorer
C:\ch07solutions\ex07_16.html
Google
Solution 7.16
Page
Tools
My Computer
100%

N	10*N	100*N	1000*N
1	10	100	1000
2	20	200	2000
3	30	300	3000
4	40	400	4000
5	50	500	5000

7.17 Using an approach similar to that in Exercise 7.15, find the *two* largest values among the 10 digits entered. [*Note:* You may input each number only once.]

7.18 Modify the program in Fig. 7.11 to validate its inputs. For every value input, if the value entered is other than 1 or 2, keep looping until the user enters a correct value.

7.19 What does the following program print?

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 7.19: ex07_19.html -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head><title>Mystery Script</title>
      <script type = "text/javascript">
         <!--
         var count = 1;

         while ( count <= 10 )
         {
            document.writeln(
               count % 2 == 1 ? "****<br />" : "++++++++<br />" );
            ++count;
         } // end while
         // -->
      </script>
   </head><body></body>
</html>
```

7.20 What does the following program print?

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 7.20: ex07_20.html -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head><title>Mystery Script</title>
      <script type = "text/javascript">
         <!--
         var row = 10;
         var column;

         while ( row >= 1 )
         {
            column = 1;

            while ( column <= 10 )
            {
               document.write( row % 2 == 1 ? "<" : ">" );
               ++column;
              } // end while

            --row;
            document.writeln( "<br />" );
         } // end while
         // -->
      </script>
   </head><body></body>
</html>
```

7.21 (*Dangling-Else Problem*) Determine the output for each of the given segments of code when `x` is `9` and `y` is `11`, and when `x` is `11` and `y` is `9`. Note that the interpreter ignores the indentation in a JavaScript program. Also, the JavaScript interpreter always associates an `else` with the previous `if`, unless told to do otherwise by the placement of braces (`{}`). You may not be sure at first glance which `if` an `else` matches. This situation is referred to as the "dangling-else" problem. We have eliminated the indentation from the given code to make the problem more challenging. [*Hint:* Apply the indentation conventions you have learned.]

a)
```
if ( x < 10 )
if ( y > 10 )
document.writeln( "*****<br />" );
else
document.writeln( "#####<br />" );
document.writeln( "$$$$$<br />" );
```

b)
```
if ( x < 10 )
{
if ( y > 10 )
document.writeln( "*****<br />" );
}
else
{
document.writeln( "#####<br />" );
document.writeln( "$$$$$<br />" );
}
```

7.22 *(Another Dangling-Else Problem)* Modify the given code to produce the output shown in each part of this problem. Use proper indentation techniques. You may not make any changes other than inserting braces, changing the code indentation and inserting blank lines. The interpreter ignores indentation in JavaScript. We have eliminated the indentation from the given code to make the problem more challenging. [*Note:* It is possible that no modification is necessary for some of the segments of code.]

```
if ( y == 8 )
if ( x == 5 )
document.writeln( "@@@@@<br />" );
else
document.writeln( "#####<br />" );
document.writeln( "$$$$$<br />" );
document.writeln( "&&&&&<br />" );
```

a) Assuming that `x = 5` and `y = 8`, the following output is produced:

```
@@@@@
$$$$$
&&&&&
```

b) Assuming that `x = 5` and `y = 8`, the following output is produced:

```
@@@@@
```

c) Assuming that `x = 5` and `y = 8`, the following output is produced:

```
@@@@@
&&&&&
```

d) Assuming that `x = 5` and `y = 7`, the following output is produced [*Note:* The last three output statements after the `else` statements are all part of a block]:

```
#####
$$$$$
&&&&&
```

7.23 Write a script that reads in the size of the side of a square and outputs XHTML text that displays a hollow square of that size constructed of asterisks. Use a `prompt` dialog to read the size from the user. Your program should work for squares of all side sizes between 1 and 20.

7.24 A palindrome is a number or a text phrase that reads the same backward and forward. For example, each of the following five-digit integers is a palindrome: 12321, 55555, 45554 and 11611. Write a script that reads in a five-digit integer and determines whether it is a palindrome. If the number is not five digits long, display an `alert` dialog indicating the problem to the user. Allow the user to enter a new value after dismissing the `alert` dialog. [*Hint:* It is possible to do this exercise with the techniques learned in this chapter. You will need to use both division and remainder operations to “pick off” each digit.]

7.25 Write a script that outputs XHTML text that displays the following checkerboard pattern:

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

Your program may use only three output statements to display the pattern, one of the form

```
document.write( "* " );
```

one of the form

```
document.write( " " );
```

and one of the form

```
document.writeln(); //writes a newline character
```

You may use XHTML tags (e.g., `<pre>`) for alignment purposes. [*Hint:* Repetition structures are required in this exercise.]

7.26 Write a script that outputs XHTML text that keeps displaying in the browser window the multiples of the integer 2, namely 2, 4, 8, 16, 32, 64, etc. Your loop should *not terminate* (i.e., you should create an infinite loop). What happens when you run this program?

7.27 A company wants to transmit data over the telephone, but it is concerned that its phones may be tapped. All of its data is transmitted as four-digit integers. It has asked you to write a program that will encrypt its data so that the data may be transmitted more securely. Your script should read a four-digit integer entered by the user in a `prompt` dialog and encrypt it as follows: Replace each digit by *(the sum of that digit plus 7) modulus 10*. Then swap the first digit with the third, and swap the second digit with the fourth. Then output XHTML text that displays the encrypted integer.

7.28 Write a program that inputs an encrypted four-digit integer (from Exercise 7.27) and decrypts it to form the original number.

8

JavaScript: Control Statements II

Not everything that can be counted counts, and not every thing that counts can be counted.

—Albert Einstein

Who can control his fate?

—William Shakespeare

The used key is always bright.

—Benjamin Franklin

Intelligence … is the faculty of making artificial objects, especially tools to make tools.

—Henri Bergson

Every advantage in the past is judged in the light of the final issue.

—Demosthenes

OBJECTIVES

In this chapter you will learn:

- The essentials of counter-controlled repetition
- To use the `for` and `do…while` repetition statements to execute statements in a program repeatedly.
- To perform multiple selection using the `switch` selection statement.
- To use the `break` and `continue` program-control statements
- To use the logical operators.

Outline

8.1 Introduction

Chapter 7 began our introduction to the types of building blocks that are available for problem solving and used them to employ proven program-construction principles. In this chapter, we continue our presentation of the theory and principles of structured programming by introducing JavaScript's remaining control statements (with the exception of `for…in`, which is presented in Chapter 10). As in Chapter 7, the JavaScript techniques you will learn here are applicable to most high-level languages. In later chapters, you'll see that control structures are helpful in manipulating objects.

8.2 Essentials of Counter-Controlled Repetition

Counter-controlled repetition requires:

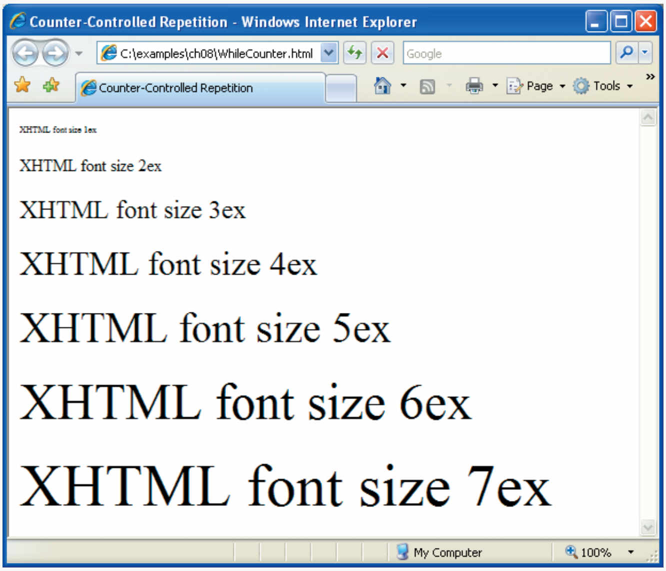
1. The *name* of a control variable (or loop counter).
2. The *initial value* of the control variable.
3. The *increment* (or *decrement*) by which the control variable is modified each time through the loop (also known as *each iteration of the loop*).
4. The condition that tests for the *final value* of the control variable to determine whether looping should continue.

To see the four elements of counter-controlled repetition, consider the simple script shown in Fig. 8.1, which displays lines of XHTML text that illustrate the seven different font sizes supported by XHTML. The declaration in line 12 *names* the control variable (`counter`), reserves space for it in memory and sets it to an *initial value* of `1`. The declaration and initialization of `counter` could also have been accomplished by the following declaration and assignment statement:

```
var counter; // declare counter
counter = 1; // initialize counter to 1
```

Lines 16–18 in the `while` statement write a paragraph element consisting of the string "`XHTML font size`" concatenated with the control variable `counter`'s value, which repre-

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 8.1: WhileCounter.html -->
<!-- Counter-controlled repetition. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Counter-Controlled Repetition</title>
      <script type = "text/javascript">
         <!--
         var counter = 1; // initialization

         while ( counter <= 7 ) // repetition condition
         {
            document.writeln( "<p style = \"font-size: " +
               counter + "ex\">XHTML font size " + counter +
               "ex</p>" );
            ++counter; // increment
         } //end while
         // -->
      </script>
   </head><body></body>
</html>
```


Counter-Controlled Repetition - Windows Internet Explorer
C:\examples\ch08\WhileCounter.html
Google
Counter-Controlled Repetition
Page
Tools
My Computer
100%

```
XHTML font size 1ex

XHTML font size 2ex

XHTML font size 3ex

XHTML font size 4ex

XHTML font size 5ex

XHTML font size 6ex

XHTML font size 7ex
```

Fig. 8.1 | Counter-controlled repetition.

sents the font size. An inline CSS `style` attribute sets the `font-size` property to the value of `counter` concatenated to `ex`. Note the use of the escape sequence `\"`, which is placed around attribute `style`'s value. Because the double-quote character delimits the beginning and end of a string literal in JavaScript, it cannot be used in the contents of the string unless it is preceded by a \ to create the escape sequence \". For example, if counter is 5, the preceding statement produces the markup

```
<p style = "font-size: 5ex">XHTML font size 5ex</p>
```

XHTML allows either single quotes (') or double quotes (") to be placed around the value specified for an attribute. JavaScript allows single quotes to be placed in a string literal. Thus, we could have placed single quotes around the font-size property to produce equivalent XHTML output without the use of escape sequences.

Common Programming Error 8.1

Placing a double-quote (") character inside a string literal that is delimited by double quotes causes a runtime error when the script is interpreted. To be displayed as part of a string literal, a double-quote (") character must be preceded by a \ to form the escape sequence \".

Line 19 in the while statement *increments* the control variable by 1 for each iteration of the loop (i.e., each time the body of the loop is performed). The loop-continuation condition (line 14) in the while statement tests whether the value of the control variable is less than or equal to 7 (the *final value* for which the condition is true). Note that the body of this while statement executes even when the control variable is 7. The loop terminates when the control variable exceeds 7 (i.e., counter becomes 8).

Good Programming Practice 8.1

Use integer values to control loop counting.

Good Programming Practice 8.2

Indent the statements in the body of each control structure.

Good Programming Practice 8.3

Put a blank line before and after each control structure, to make it stand out in the program.

Good Programming Practice 8.4

Too many levels of nesting can make a program difficult to understand. As a general rule, try to avoid using more than three levels of nesting.

Good Programming Practice 8.5

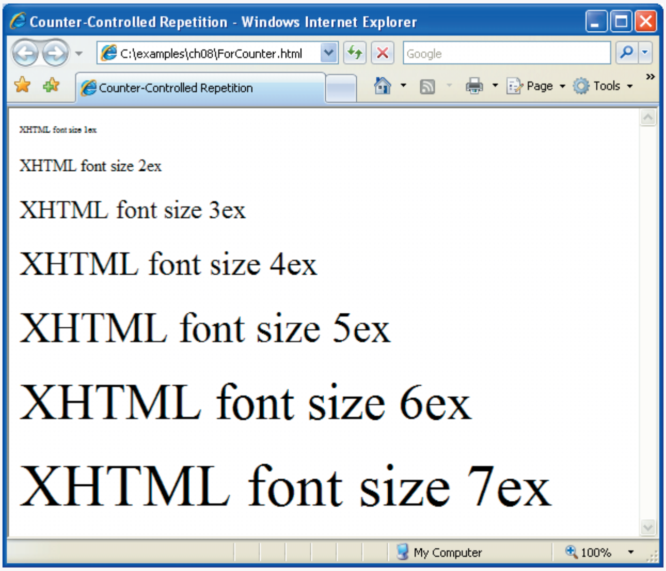
Vertical spacing above and below control structures and indentation of the bodies of control structures in the headers of the control structure give programs a two-dimensional appearance that enhances readability.

8.3 for Repetition Statement

The **for repetition statement** handles all the details of counter-controlled repetition. Figure 8.2 illustrates the power of the for statement by reimplementing the script of Fig. 8.1.

When the for statement begins executing (line 15), the control variable counter is declared and is initialized to 1 (i.e., the first statement of the for statement declares the control variable's *name* and provides the control variable's *initial value*). Next, the loop-

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 8.2: ForCounter.html -->
 6  <!-- Counter-controlled repetition with the for statement. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Counter-Controlled Repetition</title>
10        <script type = "text/javascript">
11           <!--
12           // Initialization, repetition condition and
13           // incrementing are all included in the for
14           // statement header.
15           for ( var counter = 1; counter <= 7; ++counter )
16              document.writeln( "<p style = \"font-size: " +
17                 counter + "ex\">XHTML font size " + counter +
18                 "ex</p>" );
19           // -->
20        </script>
21     </head><body></body>
22  </html>
```


Counter-Controlled Repetition - Windows Internet Explorer
C:\examples\ch08\ForCounter.html
Google
Counter-Controlled Repetition
Page
Tools
My Computer
100%

XHTML font size 1ex

XHTML font size 2ex

XHTML font size 3ex

XHTML font size 4ex

XHTML font size 5ex

XHTML font size 6ex

XHTML font size 7ex

Fig. 8.2 | Counter-controlled repetition with the `for` statement.

continuation condition, `counter` <= 7, is checked. The condition contains the *final value* (7) of the control variable. The initial value of `counter` is 1. Therefore, the condition is satisfied (i.e., `true`), so the body statement (lines 16–18) writes a paragraph element in the XHTML document. Then, variable `counter` is incremented in the expression ++`counter` and the loop continues execution with the loop-continuation test. The control variable is now equal to 2, so the final value is not exceeded and the program performs the body statement again (i.e., performs the next iteration of the loop). This process continues until the control variable `counter` becomes `8`, at which point the loop-continuation test fails and the repetition terminates.

The program continues by performing the first statement after the `for` statement. (In this case, the script terminates, because the interpreter reaches the end of the script.)

Figure 8.3 takes a closer look at the `for` statement at line 15 of Fig. 8.2. The `for` statement's first line (including the keyword `for` and everything in parentheses after `for`) is often called the **`for` statement header**. Note that the `for` statement "does it all"—it specifies each of the items needed for counter-controlled repetition with a control variable. Remember that a block is a group of statements enclosed in curly braces that can be placed anywhere that a single statement can be placed, so you can use a block to put multiple statements into the body of a `for` statement.

Note that Fig. 8.3 uses the loop-continuation condition `counter <= 7`. If you incorrectly write `counter < 7`, the loop will execute only six times. This is an example of the common logic error called an **off-by-one error**.

Common Programming Error 8.2

Using an incorrect relational operator or an incorrect final value of a loop counter in the condition of a `while`, `for` or `do...while` statement can cause an off-by-one error or an infinite loop.

Good Programming Practice 8.6

*Using the final value in the condition of a `while` or `for` statement and using the `<=` relational operator will help avoid off-by-one errors. For a loop used to print the values 1 to 10, for example, the initial value of `counter` should be `1`, and the loop-continuation condition should be `counter <= 10` rather than `counter < 10` (which is an off-by-one error) or `counter < 11` (which is correct). Many programmers, however, prefer so-called **zero-based counting**, in which, to count 10 times through the loop, `counter` would be initialized to zero and the loop-continuation test would be `counter < 10`.*

The general format of the `for` statement is

```
for ( initialization; loopContinuationTest; increment )
   statements
```

where the *initialization* expression names the loop's control variable and provides its initial value, *loopContinuationTest* is the expression that tests the loop-continuation condition (containing the final value of the control variable for which the condition is true), and

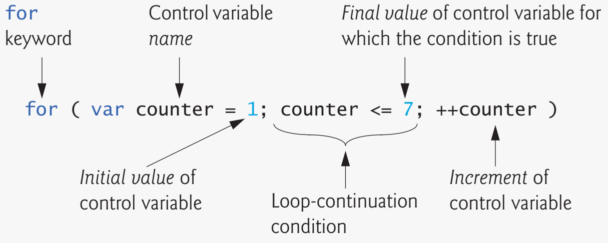


Fig. 8.3 | `for` statement header components.

increment is an expression that increments the control variable. The `for` statement can be represented by an equivalent `while` statement, with *initialization*, *loopContinuationTest* and *increment* placed as follows:

```
initialization;

while ( loopContinuationTest )
{
   statements
   increment;
}
```

In Section 8.7 we discuss an exception to this rule.

If the *initialization* expression in the `for` statement's header is the first definition of the control variable, the control variable can still be used after the `for` statement in the script. The part of a script in which a variable name can be used is known as the variable's **scope**. Scope is discussed in detail in Chapter 9, JavaScript: Functions.

Good Programming Practice 8.7

Place only expressions involving the control variable in the initialization and increment sections of a `for` *statement. Manipulations of other variables should appear either before the loop (if they execute only once, like initialization statements) or in the loop body (if they execute once per iteration of the loop, like incrementing or decrementing statements).*

The three expressions in the `for` statement are optional. If *loopContinuationTest* is omitted, JavaScript assumes that the loop-continuation condition is `true`, thus creating an infinite loop. One might omit the *initialization* expression if the control variable is initialized before the loop. One might omit the *increment* expression if the increment is calculated by statements in the body of the `for` statement or if no increment is needed. The increment expression in the `for` statement acts like a stand-alone statement at the end of the body of the `for` statement. Therefore, the expressions

```
counter = counter + 1
counter += 1
++counter
counter++
```

are all equivalent in the incrementing portion of the `for` statement. Many programmers prefer the form `counter++`. This is because the incrementing of the control variable occurs after the body of the loop is executed, and therefore the postincrementing form seems more natural. Preincrementing and postincrementing both have the same effect in our example, because the variable being incremented does not appear in a larger expression. The two semicolons in the `for` statement header are required.

Common Programming Error 8.3

Using commas instead of the two required semicolons in the header of a `for` *statement is a syntax error.*

Common Programming Error 8.4

Placing a semicolon immediately to the right of the right parenthesis of the header of a `for` *statement makes the body of that* `for` *statement an empty statement. This code is normally a logic error.*

The initialization, loop-continuation condition and increment portions of a for statement can contain arithmetic expressions. For example, assume that x = 2 and y = 10. If x and y are not modified in the body of the loop, then the statement

```
for ( var j = x; j <= 4 * x * y; j += y / x )
```

is equivalent to the statement

```
for ( var j = 2; j <= 80; j += 5 )
```

The "increment" of a for statement may be negative, in which case it is really a decrement and the loop actually counts downward.

If the loop-continuation condition initially is false, the for statement's body is not performed. Instead, execution proceeds with the statement following the for statement.

The control variable frequently is printed or used in calculations in the body of a for statement, but it does not have to be. Other times, the control variable is used for controlling repetition but never mentioned in the body of the for statement.

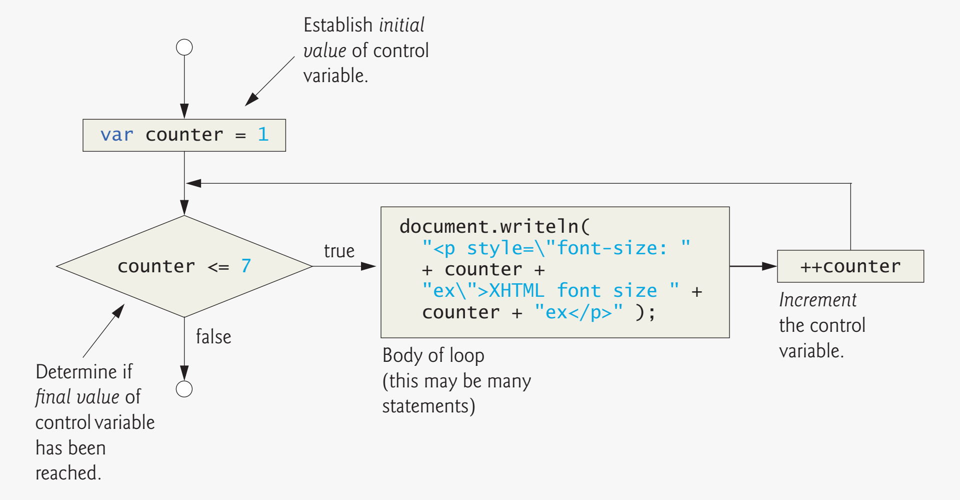
Error-Prevention Tip 8.1

Although the value of the control variable can be changed in the body of a for *statement, avoid changing it, because doing so can lead to subtle errors.*

The for statement is flowcharted much like the while statement. For example, Fig. 8.4 shows the flowchart of the for statement

```
for ( var counter = 1; counter <= 7; ++counter )
   document.writeln( "<p style = \"font-size: " +
      counter + "ex\">XHTML font size " + counter +
      "ex</p>" );
```

This flowchart makes it clear that the initialization occurs only once and that incrementing occurs *after* each execution of the body statement. Note that, besides small circles and arrows, the flowchart contains only rectangle symbols and a diamond symbol.



```
var counter = 1
```

```
counter <= 7
```

```
document.writeln(
  "<p style=\"font-size: "
  + counter +
  "ex\">XHTML font size " +
  counter + "ex</p>" );
```

```
++counter
```

Establish *initial value* of control variable.
true
false
Determine if *final value* of control variable has been reached.
Body of loop (this may be many statements)
Increment the control variable.

Fig. 8.4 | for repetition statement flowchart.

8.4 Examples Using the for Statement

The examples in this section show methods of varying the control variable in a for statement. In each case, we write the appropriate for header. Note the change in the relational operator for loops that decrement the control variable.

a) Vary the control variable from 1 to 100 in increments of 1.

```
for ( var i = 1; i <= 100; ++i )
```

b) Vary the control variable from 100 to 1 in increments of -1 (i.e., decrements of 1).

```
for ( var i = 100; i >= 1; --i )
```

c) Vary the control variable from 7 to 77 in steps of 7.

```
for ( var i = 7; i <= 77; i += 7 )
```

d) Vary the control variable from 20 to 2 in steps of -2.

```
for ( var i = 20; i >= 2; i -= 2 )
```

e) Vary the control variable over the following sequence of values: 2, 5, 8, 11, 14, 17, 20.

```
for ( var j = 2; j <= 20; j += 3 )
```

f) Vary the control variable over the following sequence of values: 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

```
for ( var j = 99; j >= 0; j -= 11 )
```

Common Programming Error 8.5

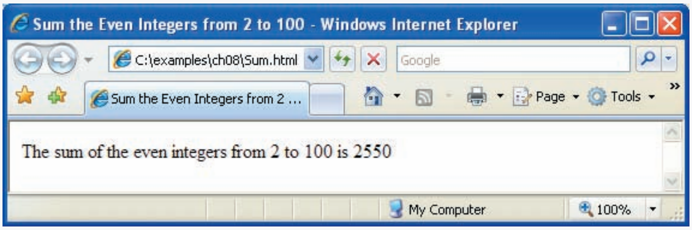
Not using the proper relational operator in the loop-continuation condition of a loop that counts downward (e.g., using i <= 1 in a loop that counts down to 1) is usually a logic error that will yield incorrect results when the program runs.

The next two scripts demonstrate the for repetition statement. Figure 8.5 uses the for statement to sum the even integers from 2 to 100. Note that the increment expression adds 2 to the control variable number after the body executes during each iteration of the loop. The loop terminates when number has the value 102 (which is not added to the sum).

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 8.5: Sum.html -->
<!-- Summation with the for repetition structure. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Sum the Even Integers from 2 to 100</title>
      <script type = "text/javascript">
         <!--
         var sum = 0;
```

Fig. 8.5 | Summation with the for repetition structure. (Part 1 of 2.)

```

         for ( var number = 2; number <= 100; number += 2 )
            sum += number;

         document.writeln( "The sum of the even integers " +
            "from 2 to 100 is " + sum );
         // -->
      </script>
   </head><body></body>
</html>
```


Sum the Even Integers from 2 to 100 - Windows Internet Explorer
C:\examples\ch08\Sum.html
Google
Sum the Even Integers from 2 ...
Page
Tools
My Computer
100%

The sum of the even integers from 2 to 100 is 2550

Fig. 8.5 | Summation with the for repetition structure. (Part 2 of 2.)

Note that the body of the for statement in Fig. 8.5 actually could be merged into the rightmost (increment) portion of the for header by using a comma, as follows:

```
for ( var number = 2; number <= 100; sum += number, number += 2)
   ;
```

Similarly, the initialization sum = 0 could be merged into the initialization section of the for statement.

Good Programming Practice 8.8

Although statements preceding a for *statement and in the body of a* for *statement can often be merged into the* for *header, avoid doing so, because it makes the program more difficult to read.*

Good Programming Practice 8.9

For clarity, limit the size of control-statement headers to a single line, if possible.

The next example computes compound interest (compounded yearly) using the for statement. Consider the following problem statement:

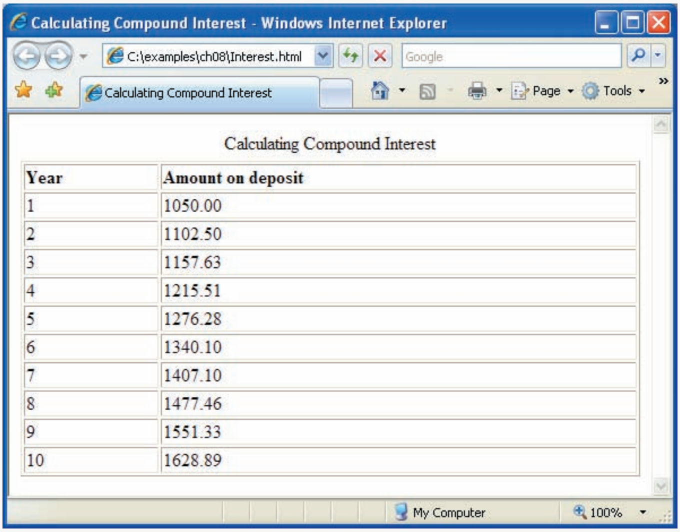
> *A person invests $1000.00 in a savings account yielding 5 percent interest. Assuming that all the interest is left on deposit, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula to determine the amounts:*
>
> $a = p\,(1 + r)^{\,n}$
>
> *where*
>
> *p* is the original amount invested (i.e., the principal)
> *r* is the annual interest rate
> *n* is the number of years
> *a* is the amount on deposit at the end of the *n*th year.

This problem involves a loop that performs the indicated calculation for each of the 10 years the money remains on deposit. Figure 8.6 presents the solution to this problem, displaying the results in a table.

Lines 16–18 declare three variables and initialize `principal` to `1000.0` and `rate` to `.05`. Lines 20–21 write an XHTML `<table>` tag, and lines 22–23 write the `caption` that summarizes the table's content. Lines 24–25 create the table's header section (`<thead>`), a

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 8.6: Interest.html -->
 6  <!-- Compound interest calculation with a for loop. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Calculating Compound Interest</title>
10        <style type = "text/css">
11           table { width: 100% }
12           th    { text-align: left }
13        </style>
14        <script type = "text/javascript">
15           <!--
16           var amount; // current amount of money
17           var principal = 1000.0; // principal amount
18           var rate = .05; // interest rate
19
20           document.writeln(
21              "<table border = \"1\">" ); // begin the table
22           document.writeln(
23              "<caption>Calculating Compound Interest</caption>" );
24           document.writeln(
25              "<thead><tr><th>Year</th>" ); // year column heading
26           document.writeln(
27              "<th>Amount on deposit</th>" ); // amount column heading
28           document.writeln( "</tr></thead><tbody>" );
29
30           // output a table row for each year
31           for ( var year = 1; year <= 10; ++year )
32           {
33              amount = principal * Math.pow( 1.0 + rate, year );
34              document.writeln( "<tr><td>" + year +
35                 "</td><td>" + amount.toFixed(2) +
36                 "</td></tr>" );
37           } //end for
38
39           document.writeln( "</tbody></table>" );
40           // -->
41        </script>
42     </head><body></body>
43  </html>
```

Fig. 8.6 | Compound interest calculation with a `for` loop. (Part 1 of 2.)


Calculating Compound Interest - Windows Internet Explorer
C:\examples\ch08\Interest.html
Google
Calculating Compound Interest
Page
Tools
My Computer
100%

Calculating Compound Interest

Year	Amount on deposit
1	1050.00
2	1102.50
3	1157.63
4	1215.51
5	1276.28
6	1340.10
7	1407.10
8	1477.46
9	1551.33
10	1628.89

Fig. 8.6 | Compound interest calculation with a for loop. (Part 2 of 2.)

row (<tr>) and a column heading (<th>) containing "Year." Lines 26–28 create a table heading for "Amount on deposit" and write the closing </tr> and </thead> tags.

The for statement (lines 31–37) executes its body 10 times, incrementing control variable year from 1 to 10 (note that year represents *n* in the problem statement). JavaScript does not include an exponentiation operator. Instead, we use the Math object's pow method for this purpose. Math.pow(x, y) calculates the value of x raised to the yth power. Method Math.pow takes two numbers as arguments and returns the result.

Line 33 performs the calculation using the formula given in the problem statement. Lines 34–36 write a line of XHTML markup that creates another row in the table. The first column is the current year value. The second column displays the value of amount. Line 39 writes the closing </tbody> and </table> tags after the loop terminates.

Line 35 introduces the **Number object** and its **toFixed method**. The variable amount contains a numerical value, so JavaScript represents it as a Number object. The toFixed method of a Number object formats the value by rounding it to the specified number of decimal places. On line 35, amount.toFixed(2) outputs the value of amount with two decimal places.

Variables amount, principal and rate represent numbers in this script. Remember that JavaScript represents all numbers as floating-point numbers. This feature is convenient in this example, because we are dealing with fractional parts of dollars and need a type that allows decimal points in its values.

Unfortunately, floating-point numbers can cause trouble. Here is a simple example of what can go wrong when using floating-point numbers to represent dollar amounts (assuming that dollar amounts are displayed with two digits to the right of the decimal point): Two dollar amounts stored in the machine could be 14.234 (which would normally be rounded to 14.23 for display purposes) and 18.673 (which would normally be rounded to 18.67 for display purposes). When these amounts are added, they produce the internal sum 32.907, which would normally be rounded to 32.91 for display purposes. Thus your printout could appear as

```
  14.23
+ 18.67
-------
  32.91
```

but a person adding the individual numbers as printed would expect the sum to be 32.90. You have been warned!

8.5 switch Multiple-Selection Statement

Previously, we discussed the if single-selection statement and the if…else double-selection statement. Occasionally, an algorithm will contain a series of decisions in which a variable or expression is tested separately for each of the values it may assume, and different actions are taken for each value. JavaScript provides the switch multiple-selection statement to handle such decision making. The script in Fig. 8.7 demonstrates three different CSS list formats determined by the value the user enters.

Line 12 in the script declares the variable choice. This variable stores the user's choice, which determines what type of XHTML list to display. Lines 13–14 declare variables startTag and endTag, which will store the XHTML tags that will be used to create the list element. Line 15 declares variable validInput and initializes it to true. The script uses this variable to determine whether the user made a valid choice (indicated by the value of true). If a choice is invalid, the script sets validInput to false. Line 16 declares variable listType, which will store an h1 element indicating the list type. This heading appears before the list in the XHTML document.

Lines 18–19 prompt the user to enter a 1 to display a numbered list, a 2 to display a lettered list and a 3 to display a list with roman numerals. Lines 21–40 define a **switch statement** that assigns to the variables startTag, endTag and listType values based on the value input by the user in the prompt dialog. We create these different lists using the CSS property **list-style-type**, which allows us to set the numbering system for the list. Possible values include decimal (numbers—the default), lower-roman (lowercase Roman numerals), upper-roman (uppercase Roman numerals), lower-alpha (lowercase letters), upper-alpha (uppercase letters), and several others.

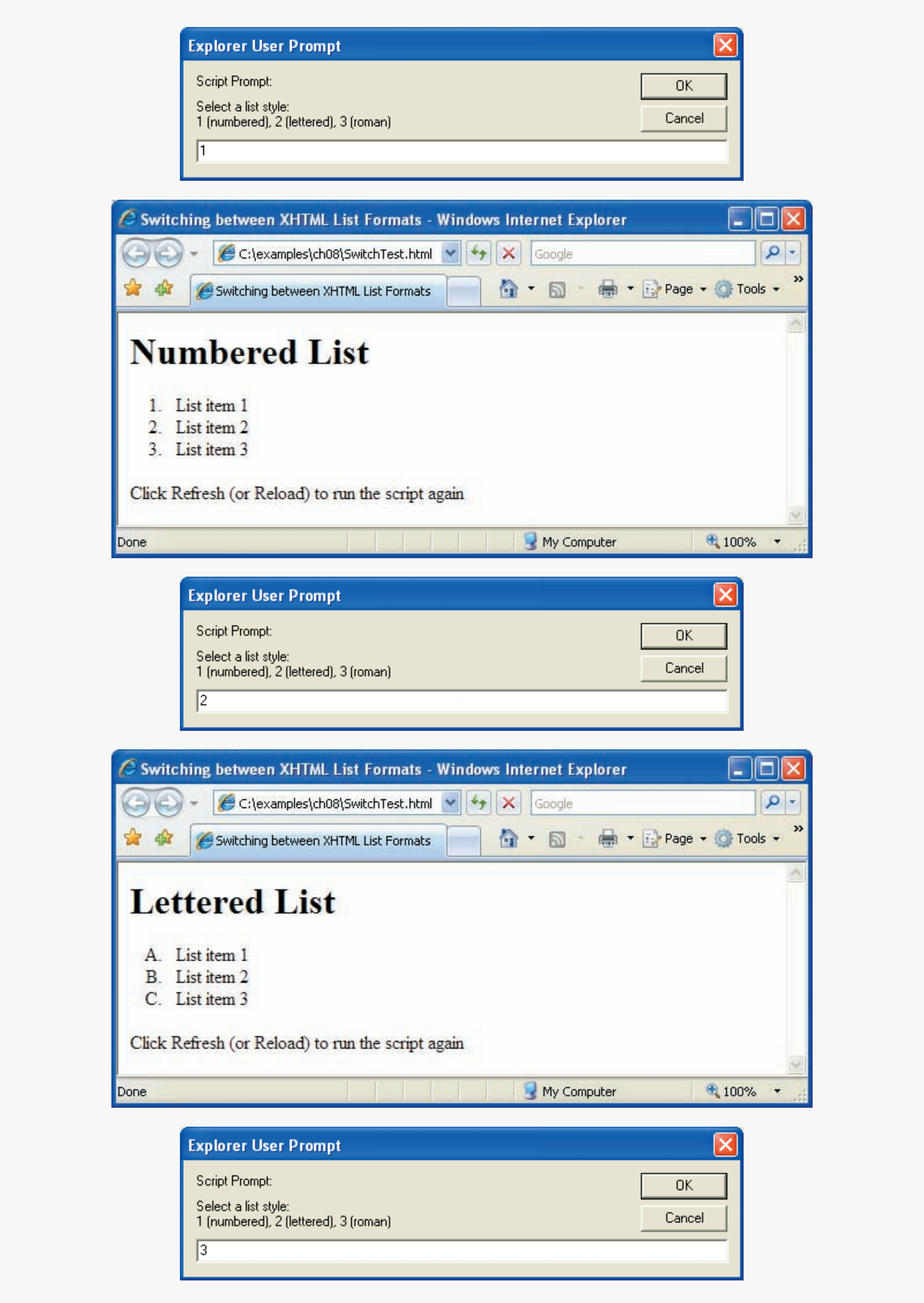
The switch statement consists of a series of **case labels** and an optional **default case**. When the flow of control reaches the switch statement, the script evaluates the **controlling expression** (choice in this example) in the parentheses following keyword switch. The value of this expression is compared with the value in each of the case labels, starting with the first case label. Assume that the user entered 2. Remember that the value typed

```
1  <?xml version = "1.0" encoding = "utf-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 8.7: SwitchTest.html -->
6  <!-- Using the switch multiple-selection statement. -->
```

Fig. 8.7 | Using the switch multiple-selection statement. (Part 1 of 4.)

```
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Switching between XHTML List Formats</title>
      <script type = "text/javascript">
         <!--
         var choice; // user's choice
         var startTag; // starting list item tag
         var endTag; // ending list item tag
         var validInput = true; // indicates if input is valid
         var listType; // type of list as a string

         choice = window.prompt( "Select a list style:\n" +
             "1 (numbered), 2 (lettered), 3 (roman)", "1" );

         switch ( choice )
         {
            case "1":
               startTag = "<ol>";
               endTag = "</ol>";
               listType = "<h1>Numbered List</h1>";
               break;
            case "2":
               startTag = "<ol style = \"list-style-type: upper-alpha\">";
               endTag = "</ol>";
               listType = "<h1>Lettered List</h1>";
               break;
            case "3":
               startTag = "<ol style = \"list-style-type: upper-roman\">";
               endTag = "</ol>";
               listType = "<h1>Roman Numbered List</h1>";
               break;
            default:
               validInput = false;
         } //end switch

         if ( validInput == true )
         {
            document.writeln( listType + startTag );

            for ( var i = 1; i <= 3; ++i )
               document.writeln( "<li>List item " + i + "</li>" );

            document.writeln( endTag );
         } //end if
         else
            document.writeln( "Invalid choice: " + choice );
         // -->
      </script>
   </head>
   <body>
      <p>Click Refresh (or Reload) to run the script again</p>
   </body>
</html>
```

Fig. 8.7 | Using the switch multiple-selection statement. (Part 2 of 4.)



Explorer User Prompt

Script Prompt:

Select a list style:
1 (numbered), 2 (lettered), 3 (roman)

1

Switching between XHTML List Formats - Windows Internet Explorer

C:\examples\ch08\SwitchTest.html

Numbered List

1. List item 1
2. List item 2
3. List item 3

Click Refresh (or Reload) to run the script again

Explorer User Prompt

Script Prompt:

Select a list style:
1 (numbered), 2 (lettered), 3 (roman)

2

Switching between XHTML List Formats - Windows Internet Explorer

C:\examples\ch08\SwitchTest.html

Lettered List

A. List item 1
B. List item 2
C. List item 3

Click Refresh (or Reload) to run the script again

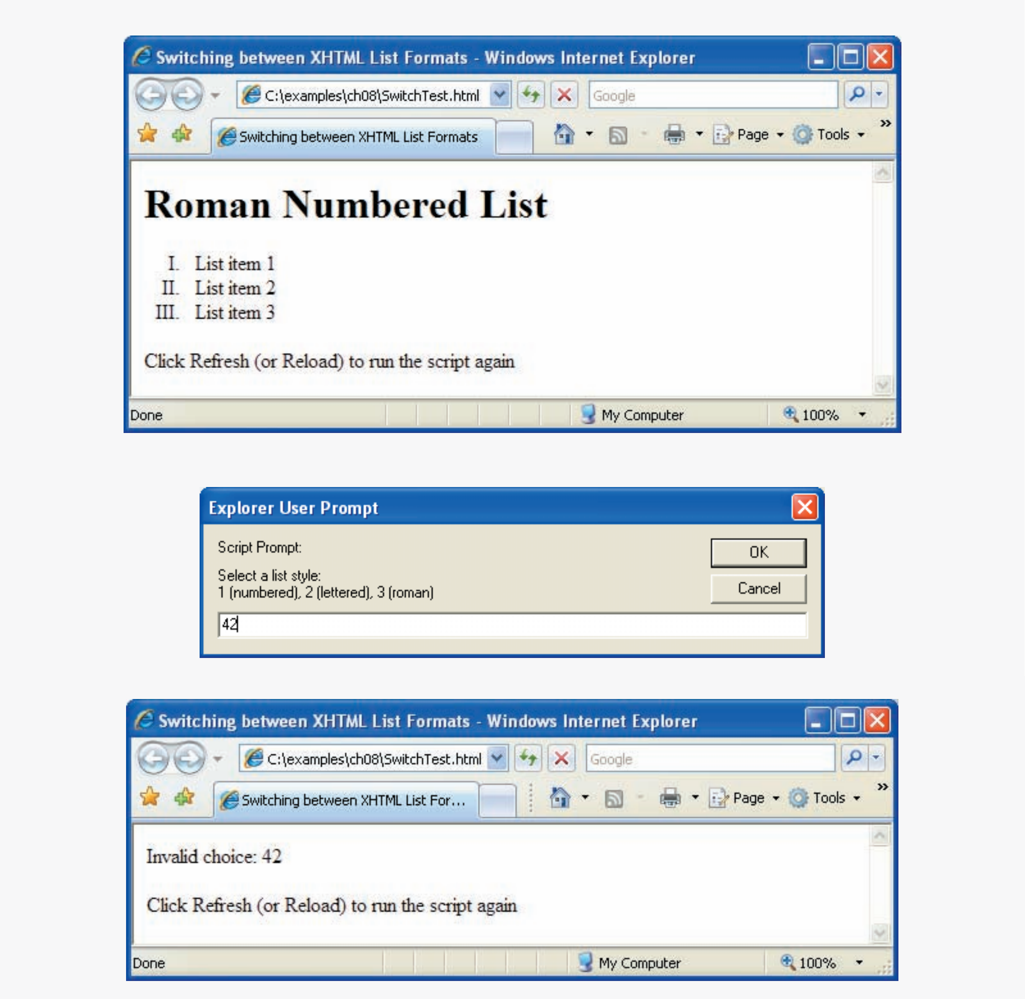
Explorer User Prompt

Script Prompt:

Select a list style:
1 (numbered), 2 (lettered), 3 (roman)

3

Fig. 8.7 | Using the `switch` multiple-selection statement. (Part 3 of 4.)


Switching between XHTML List Formats - Windows Internet Explorer
C:\examples\ch08\SwitchTest.html
Google
Switching between XHTML List Formats
Page
Tools
Done
My Computer
100%
Explorer User Prompt
OK
Cancel
Switching between XHTML List Formats - Windows Internet Explorer
C:\examples\ch08\SwitchTest.html
Google
Switching between XHTML List For...
Page
Tools
Done
My Computer
100%

Roman Numbered List

I. List item 1
II. List item 2
III. List item 3

Click Refresh (or Reload) to run the script again

Script Prompt:

Select a list style:
1 (numbered), 2 (lettered), 3 (roman)

42

Invalid choice: 42

Click Refresh (or Reload) to run the script again

Fig. 8.7 | Using the switch multiple-selection statement. (Part 4 of 4.)

by the user in a prompt dialog is returned as a string. So, the string 2 is compared to the string in each case in the switch statement. If a match occurs (case "2":), the statements for that case execute. For the string 2 (lines 29–32), we set startTag to an opening ol tag with the style property list-style-type set to upper-alpha, set endTag to "</ol>" to indicate the end of an ordered list and set listType to "<h1>Lettered List</h1>". If no match occurs between the controlling expression's value and a case label, the default case executes and sets variable validInput to false.

The break statement in line 32 causes program control to proceed with the first statement after the switch statement. The break statement is used because the cases in a switch statement would otherwise run together. If break is not used anywhere in a switch statement, then each time a match occurs in the statement, the statements for all the remaining cases execute.

Next, the flow of control continues with the `if` statement in line 42, which tests variable `validInput` to determine whether its value is `true`. If so, lines 44–49 write the `listType`, the `startTag`, three list items (`<li>`) and the `endTag`. Otherwise, the script writes text in the XHTML document indicating that an invalid choice was made (line 52).

Each `case` can have multiple actions (statements). The `switch` statement is different from others in that braces are not required around multiple actions in a `case` of a `switch`. The general `switch` statement (i.e., using a `break` in each `case`) is flowcharted in Fig. 8.8. [*Note:* As an exercise, flowchart the general `switch` statement without `break` statements.]

The flowchart makes it clear that each `break` statement at the end of a `case` causes control to exit from the `switch` statement immediately. The `break` statement is not required for the last `case` in the `switch` statement (or the `default` case, when it appears last), because program control automatically continues with the next statement after the `switch` statement.

Common Programming Error 8.6

Forgetting a `break` statement when one is needed in a `switch` statement is a logic error.

Software Engineering Observation 8.1

Provide a `default` case in `switch` statements. Cases not explicitly tested in a `switch` statement without a `default` case are ignored. Including a `default` case focuses the programmer on processing exceptional conditions. However, there are situations in which no `default` processing is needed.

Good Programming Practice 8.10

Although the `case` clauses and the `default` case clause in a `switch` statement can occur in any order, it is clearer (and more common) to place the `default` clause last.

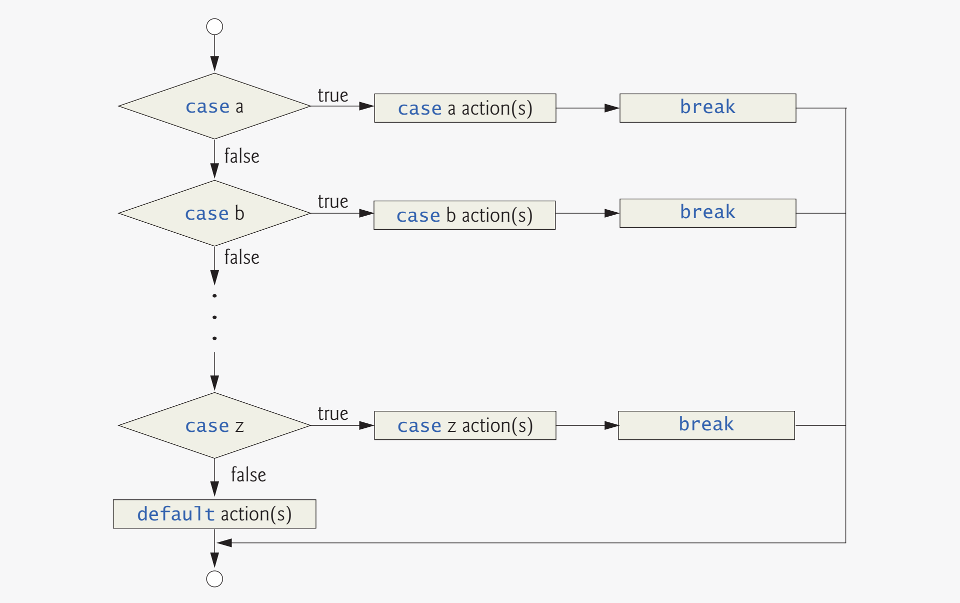


Fig. 8.8 | `switch` multiple-selection statement.

Good Programming Practice 8.11

In a `switch` statement, when the `default` clause is listed last, the `break` for that `case` statement is not required. Some programmers include this `break` for clarity and for symmetry with other `case`s.

Note that having several `case` labels listed together (e.g., `case 1: case 2:` with no statements between the cases) simply means that the same set of actions is to occur for each case. Again, note that, besides small circles and arrows, the flowchart contains only rectangle symbols and diamond symbols.

8.6 do...while Repetition Statement

The **`do...while` repetition statement** is similar to the `while` statement. In the `while` statement, the loop-continuation test occurs at the beginning of the loop, before the body of the loop executes. The `do...while` statement tests the loop-continuation condition *after* the loop body executes—therefore, *the loop body always executes at least once.* When a `do...while` terminates, execution continues with the statement after the `while` clause. Note that it is not necessary to use braces in a `do...while` statement if there is only one statement in the body. However, the braces usually are included, to avoid confusion between the `while` and `do...while` statements. For example,

```
while ( condition )
```

normally is regarded as the header to a `while` statement. A `do...while` statement with no braces around a single-statement body appears as

```
do
   statement
while ( condition );
```

which can be confusing. The last line—`while(` *condition* `);`—may be misinterpreted by the reader as a `while` statement containing an empty statement (the semicolon by itself). Thus, to avoid confusion, the `do...while` statement with a one-statement body is often written as follows:

```
do
{
   statement
} while ( condition );
```

Good Programming Practice 8.12

Some programmers always include braces in a `do...while` statement even if they are not necessary. This helps eliminate ambiguity between the `while` statement and the `do...while` statement containing a one-statement body.

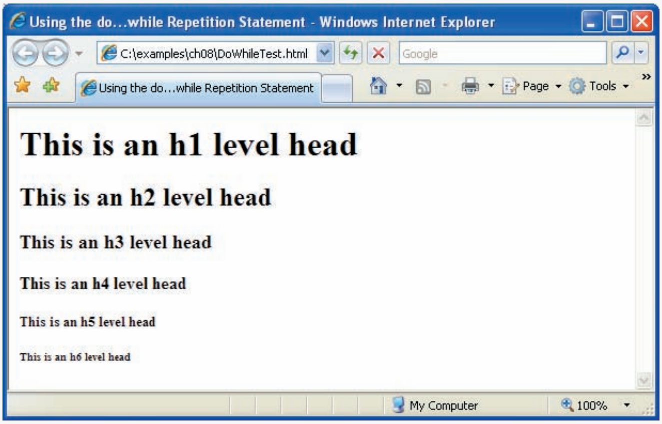
Common Programming Error 8.7

Infinite loops are caused when the loop-continuation condition never becomes `false` in a `while`, `for` or `do...while` statement. To prevent this, make sure that there is not a semicolon immediately after the header of a `while` or `for` statement. In a counter-controlled loop, make sure that the control variable is incremented (or decremented) in the body of the loop. In a sentinel-controlled loop, make sure that the sentinel value is eventually input.

The script in Fig. 8.9 uses a do…while statement to display each of the six different XHTML heading types (h1 through h6). Line 12 declares control variable counter and initializes it to 1. Upon entering the do…while statement, lines 15–17 write a line of XHTML text in the document. The value of control variable counter is used to create the starting and ending header tags (e.g., <h1> and </h1>) and to create the line of text to display (e.g., This is an h1 level head). Line 18 increments the counter before the loop-continuation test occurs at the bottom of the loop.

The do…while flowchart in Fig. 8.10 makes it clear that the loop-continuation test does not occur until the action executes at least once.

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 8.9: DoWhileTest.html -->
 6  <!-- Using the do...while repetition statement. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Using the do...while Repetition Statement</title>
10        <script type = "text/javascript">
11           <!--
12           var counter = 1;
13
14           do {
15              document.writeln( "<h" + counter + ">This is " +
16                 "an h" + counter + " level head" + "</h" +
17                 counter + ">" );
18              ++counter;
19           } while ( counter <= 6 );
20           // -->
21        </script>
22
23     </head><body></body>
24  </html>
```


Using the do...while Repetition Statement - Windows Internet Explorer
C:\examples\ch08\DoWhileTest.html
Google
Using the do...while Repetition Statement
Page
Tools
My Computer
100%

This is an h1 level head

This is an h2 level head

This is an h3 level head

This is an h4 level head

This is an h5 level head

This is an h6 level head

Fig. 8.9 | Using the do…while repetition statement.

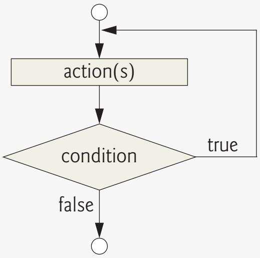


Fig. 8.10 | do...while repetition statement flowchart.

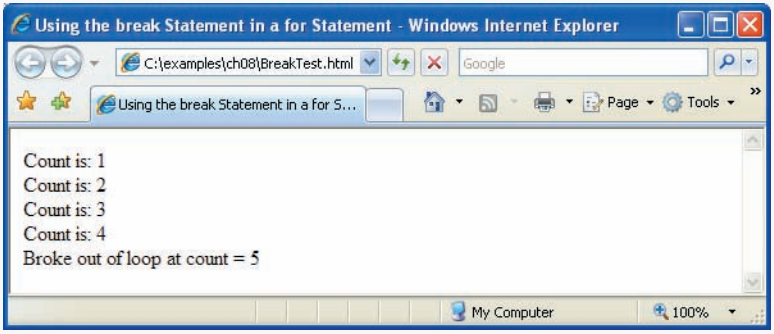
8.7 break and continue Statements

The **break** and **continue statements** alter the flow of control. The break statement, when executed in a while, for, do...while or switch statement, causes immediate exit from the statement. Execution continues with the first statement after the structure. The break statement is commonly used to escape early from a loop or to skip the remainder of a switch statement (as in Fig. 8.7). Figure 8.11 demonstrates the break statement in a for repetition statement.

During each iteration of the for statement in lines 14–20, the script writes the value of count in the XHTML document. When the if statement in line 16 detects that count

```
1   <?xml version = "1.0" encoding = "utf-8"?>
2   <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5   <!-- Fig. 8.11: BreakTest.html -->
6   <!-- Using the break statement in a for statement. -->
7   <html xmlns = "http://www.w3.org/1999/xhtml">
8      <head>
9         <title>
10           Using the break Statement in a for Statement
11        </title>
12        <script type = "text/javascript">
13           <!--
14           for ( var count = 1; count <= 10; ++count )
15           {
16              if ( count == 5 )
17                 break; // break loop only if count == 5
18
19              document.writeln( "Count is: " + count + "<br />" );
20           } //end for
21
22           document.writeln(
23              "Broke out of loop at count = " + count );
24           // -->
25        </script>
26     </head><body></body>
27  </html>
```

Fig. 8.11 | Using the break statement in a for statement. (Part 1 of 2.)


Using the break Statement in a for Statement - Windows Internet Explorer
C:\examples\ch08\BreakTest.html
Google
Using the break Statement in a for S...
Page
Tools
My Computer
100%

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Broke out of loop at count = 5
```

Fig. 8.11 | Using the `break` statement in a `for` statement. (Part 2 of 2.)

is `5`, the `break` in line 17 executes. This statement terminates the `for` statement, and the program proceeds to line 22 (the next statement in sequence immediately after the `for` statement), where the script writes the value of `count` when the loop terminated (i.e., `5`). The loop executes line 19 only four times.

The `continue` statement, when executed in a `while`, `for` or `do…while` statement, skips the remaining statements in the body of the statement and proceeds with the next iteration of the loop. In `while` and `do…while` statements, the loop-continuation test evaluates immediately after the `continue` statement executes. In `for` statements, the increment expression executes, then the loop-continuation test evaluates. This is the one case in which `for` and `while` differ. Improper placement of `continue` before the increment in a `while` may result in an infinite loop.

Figure 8.12 uses `continue` in a `for` statement to skip the `document.writeln` statement in line 20 when the `if` statement in line 17 determines that the value of `count` is `5`. When the `continue` statement executes, the script skips the remainder of the `for` statement's body. Program control continues with the increment of the `for` statement's control variable, followed by the loop-continuation test to determine whether the loop should continue executing.

Software Engineering Observation 8.2

Some programmers feel that `break` and `continue` violate structured programming. They do not use `break` and `continue`, because the effects of these statements can be achieved by structured programming techniques.

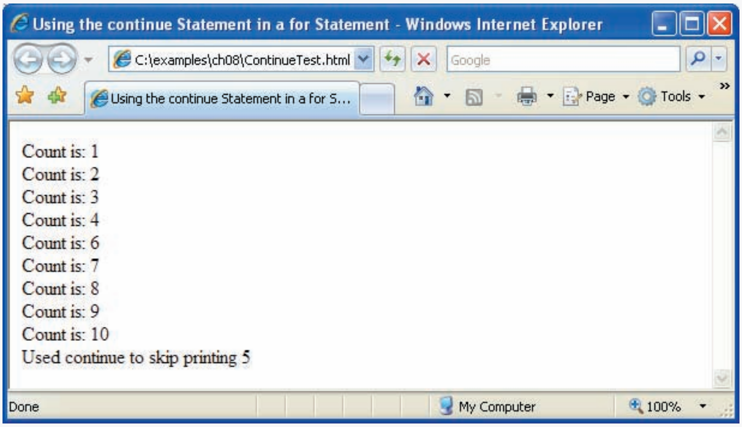
Performance Tip 8.1

The `break` and `continue` statements, when used properly, perform faster than the corresponding structured techniques.

Software Engineering Observation 8.3

There is a tension between achieving quality software engineering and achieving the best-performing software. Often, one of these goals is achieved at the expense of the other. For all but the most performance-intensive situations, the following rule of thumb should be followed: First make your code simple, readable and correct; then make it fast and small, but only if necessary.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 8.12: ContinueTest.html -->
<!-- Using the continue statement in a for statement. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>
         Using the continue Statement in a for Statement
      </title>

      <script type = "text/javascript">
         <!--
         for ( var count = 1; count <= 10; ++count )
         {
            if ( count == 5 )
               continue; // skip remaining loop code only if count == 5

            document.writeln( "Count is: " + count + "<br />" );
         } //end for

         document.writeln( "Used continue to skip printing 5" );
         // -->
      </script>

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


Using the continue Statement in a for Statement - Windows Internet Explorer
C:\examples\ch08\ContinueTest.html
Google
Using the continue Statement in a for S...
Page
Tools
Done
My Computer
100%

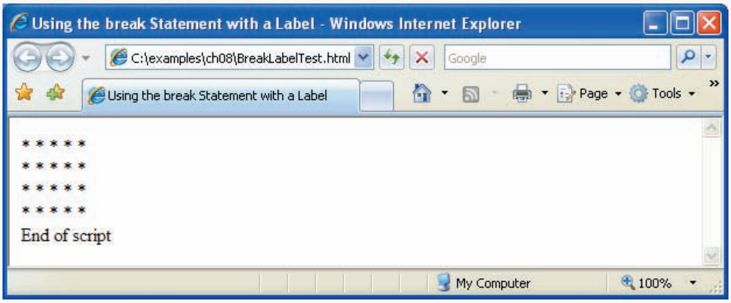
```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
Used continue to skip printing 5
```

Fig. 8.12 | Using the continue statement in a for statement.

8.8 Labeled break and continue Statements

The break statement can break out of an immediately enclosing while, for, do…while or switch statement. To break out of a nested set of structures, you can use the labeled break statement. This statement, when executed in a while, for, do…while or switch statement, causes immediate exit from that statement and any number of enclosing repetition statements; program execution resumes with the first statement after the enclosing **labeled statement** (a statement preceded by a label). The labeled statement can be a block (a set of statements enclosed in curly braces, {}). Labeled break statements commonly are used to terminate nested looping structures containing while, for, do…while or switch statements. Figure 8.13 demonstrates the labeled break statement in a nested for statement.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 8.13: BreakLabelTest.html -->
<!-- Labeled break statement in a nested for statement. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Using the break Statement with a Label</title>
      <script type = "text/javascript">
         <!--
         stop: { // labeled block
            for ( var row = 1; row <= 10; ++row )
            {
               for ( var column = 1; column <= 5 ; ++column )
               {
                  if ( row == 5 )
                     break stop; // jump to end of stop block

                  document.write( "* " );
               } //end for

               document.writeln( "<br />" );
            } //end for

            // the following line is skipped
            document.writeln( "This line should not print" );
         } // end block labeled stop

         document.writeln( "End of script" );
         // -->
      </script>
   </head><body></body>
</html>
```


Using the break Statement with a Label - Windows Internet Explorer
C:\examples\ch08\BreakLabelTest.html
Google
Using the break Statement with a Label
Page
Tools
My Computer
100%

```
* * * * *
* * * * *
* * * * *
* * * * *
End of script
```

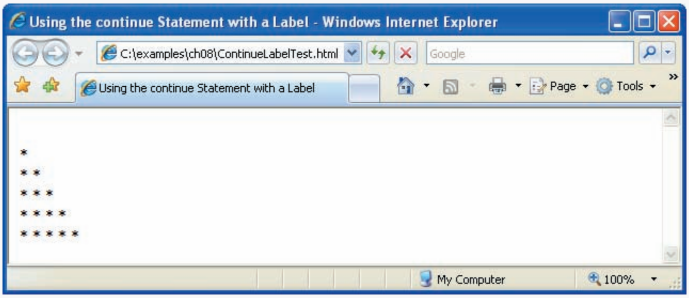
Fig. 8.13 | Labeled break statement in a nested for statement.

The labeled block (lines 12–28) begins with a label (an identifier followed by a colon). Here, we use the label stop:. The block is enclosed between the braces at the end of line 12 and in line 28, and includes both the nested for statement starting in line 13 and the document.writeln statement in line 27. When the if statement in line 17 detects that row is equal to 5, the statement in line 18 executes. This statement terminates both the for statement in line 15 and its enclosing for statement in line 13, and the program proceeds to the statement in line 30 (the first statement in sequence after the labeled block). The inner for statement executes its body only four times. Note that the document.writeln statement in line 27 never executes, because it is included in the labeled block and the outer for statement never completes.

The continue statement proceeds with the next iteration (repetition) of the immediately enclosing while, for or do…while statement. The **labeled continue statement**, when executed in a repetition statement (while, for or do…while), skips the remaining statements in the structure's body and any number of enclosing repetition statements, then proceeds with the next iteration of the enclosing **labeled repetition statement** (a repetition statement preceded by a label). In labeled while and do…while statements, the loop-continuation test evaluates immediately after the continue statement executes. In a labeled for statement, the increment expression executes, then the loop-continuation test evaluates. Figure 8.14 uses the labeled continue statement in a nested for statement to cause execution to continue with the next iteration of the outer for statement.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 8.14: ContinueLabelTest.html -->
<!-- Labeled continue statement in a nested for statement. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Using the continue Statement with a Label</title>
      <script type = "text/javascript">
         <!--
         nextRow: // target label of continue statement
            for ( var row = 1; row <= 5; ++row )
            {
               document.writeln( "<br />" );

               for ( var column = 1; column <= 10; ++column )
               {
                  if ( column > row )
                     continue nextRow; // next iteration of labeled loop

                  document.write( "* " );
               } //end for
            } //end for
         // -->
      </script>
   </head><body></body>
</html>
```

Fig. 8.14 | Labeled continue statement in a nested for statement. (Part 1 of 2.)


Using the continue Statement with a Label - Windows Internet Explorer
C:\examples\ch08\ContinueLabelTest.html
Google
Using the continue Statement with a Label
Page
Tools
My Computer
100%

```
*
**
***
****
*****
```

Fig. 8.14 | Labeled `continue` statement in a nested `for` statement. (Part 2 of 2.)

The labeled `for` statement (lines 13–24) starts with the `nextRow` label in line 12. When the `if` statement in line 19 in the inner `for` statement detects that `column` is greater than `row`, line 20 executes and program control continues with the increment of the control variable of the outer `for` statement. Even though the inner `for` statement counts from 1 to 10, the number of * characters output on a row never exceeds the value of `row`.

8.9 Logical Operators

So far, we have studied only such simple conditions as `count <= 10`, `total > 1000` and `number != sentinelValue`. These conditions were expressed in terms of the relational operators >, <, >= and <=, and in terms of the equality operators == and `!=`. Each decision tested one condition. To make a decision based on multiple conditions, we performed these tests in separate statements or in nested `if` or `if...else` statements.

JavaScript provides logical operators that can be used to form more complex conditions by combining simple conditions. The logical operators are `&&` (logical AND), `||` (logical OR) and `!` (logical NOT, also called logical negation). We consider examples of each of these operators.

Suppose that, at some point in a program, we wish to ensure that two conditions are *both* `true` before we choose a certain path of execution. In this case, we can use the logical && operator, as follows:

```
if ( gender == 1 && age >= 65 )
   ++seniorFemales;
```

This `if` statement contains two simple conditions. The condition `gender == 1` might be evaluated to determine, for example, whether a person is a female. The condition `age >= 65` is evaluated to determine whether a person is a senior citizen. The `if` statement then considers the combined condition

```
gender == 1 && age >= 65
```

This condition is `true` *if and only if* both of the simple conditions are `true`. Finally, if this combined condition is indeed `true`, the count of `seniorFemales` is incremented by `1`. If either or both of the simple conditions are `false`, the program skips the incrementing and proceeds to the statement following the `if` statement. The preceding combined condition can be made more readable by adding redundant parentheses:

```
( gender == 1 ) && ( age >= 65 )
```

The table in Fig. 8.15 summarizes the && operator. The table shows all four possible combinations of false and true values for *expression1* and *expression2*. Such tables are often called truth tables. JavaScript evaluates to false or true all expressions that include relational operators, equality operators and/or logical operators.

Now let us consider the || (logical OR) operator. Suppose we wish to ensure that either *or* both of two conditions are true before we choose a certain path of execution. In this case, we use the || operator, as in the following program segment:

```
if ( semesterAverage >= 90 || finalExam >= 90 )
   document.writeln( "Student grade is A" );
```

This statement also contains two simple conditions. The condition semesterAverage >= 90 is evaluated to determine whether the student deserves an "A" in the course because of a solid performance throughout the semester. The condition finalExam >= 90 is evaluated to determine whether the student deserves an "A" in the course because of an outstanding performance on the final exam. The if statement then considers the combined condition

```
semesterAverage >= 90 || finalExam >= 90
```

and awards the student an "A" if either or both of the simple conditions are true. Note that the message "Student grade is A" is *not* printed only when both of the simple conditions are false. Figure 8.16 is a truth table for the logical OR operator (||).

The && operator has a higher precedence than the || operator. Both operators associate from left to right. An expression containing && or || operators is evaluated only until truth or falsity is known. Thus, evaluation of the expression

```
gender == 1 && age >= 65
```

expression1	expression2	expression1 && expression2
false	false	false
false	true	false
true	false	false
true	true	true

Fig. 8.15 | Truth table for the && (logical AND) operator.

expression1	expression2	expression1 || expression2
false	false	false
false	true	true
true	false	true
true	true	true

Fig. 8.16 | Truth table for the || (logical OR) operator.

stops immediately if `gender` is not equal to `1` (i.e., the entire expression is `false`) and continues if `gender` is equal to `1` (i.e., the entire expression could still be `true` if the condition `age >= 65` is `true`). Similarly, the `||` operator immediately returns `true` if the first operand is `true`. This performance feature for evaluation of logical AND and logical OR expressions is called **short-circuit evaluation**.

JavaScript provides the `!` (logical negation) operator to enable a programmer to "reverse" the meaning of a condition (i.e., a `true` value becomes `false`, and a `false` value becomes `true`). Unlike the logical operators `&&` and `||`, which combine two conditions (i.e., they are binary operators), the logical negation operator has only a single condition as an operand (i.e., it is a unary operator). The logical negation operator is placed before a condition to choose a path of execution if the original condition (without the logical negation operator) is `false`, as in the following program segment:

```
if ( ! ( grade == sentinelValue ) )
   document.writeln( "The next grade is " + grade );
```

The parentheses around the condition `grade == sentinelValue` are needed, because the logical negation operator has a higher precedence than the equality operator. Figure 8.17 is a truth table for the logical negation operator.

In most cases, the programmer can avoid using logical negation by expressing the condition differently with an appropriate relational or equality operator. For example, the preceding statement may also be written as follows:

```
if ( grade != sentinelValue )
   document.writeln( "The next grade is " + grade );
```

The script in Fig. 8.18 demonstrates all the logical operators by producing their truth tables. The script produces an XHTML table containing the results.

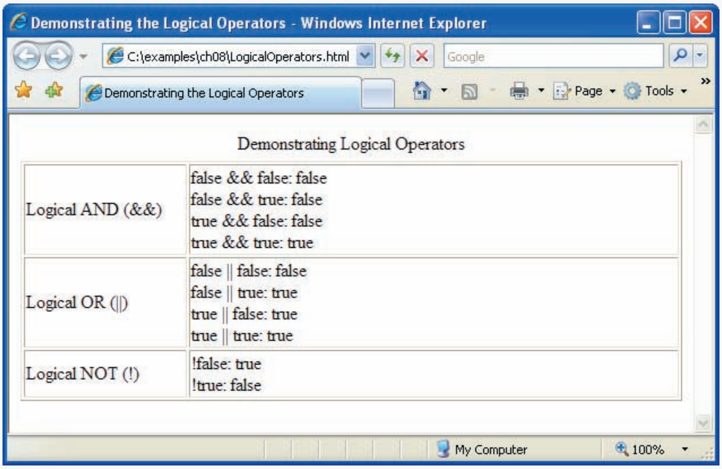
expression	!expression
`false`	`true`
`true`	`false`

Fig. 8.17 | Truth table for operator `!` (logical negation).

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 8.18: LogicalOperators.html -->
<!-- Demonstrating logical operators. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Demonstrating the Logical Operators</title>
```

Fig. 8.18 | Demonstrating logical operators. (Part 1 of 2.)

```
      <style type = "text/css">
         table   { width: 100% }
         td.left { width: 25% }
      </style>
      <script type = "text/javascript">
         <!--
         document.writeln(
            "<table border = \"1\"" );
         document.writeln(
            "<caption>Demonstrating Logical " +
               "Operators</caption>" );
         document.writeln(
            "<tr><td class = \"left\">Logical AND (&&)</td>" +
            "<td>false && false: " + ( false && false ) +
            "<br />false && true: " + ( false && true ) +
            "<br />true && false: " + ( true && false ) +
            "<br />true && true: " + ( true && true ) +
            "</td></tr>" );
         document.writeln(
            "<tr><td class = \"left\">Logical OR (||)</td>" +
            "<td>false || false: " + ( false || false ) +
            "<br />false || true: " + ( false || true ) +
            "<br />true || false: " + ( true || false ) +
            "<br />true || true: " + ( true || true ) +
            "</td></tr>" );
         document.writeln(
            "<tr><td class = \"left\">Logical NOT (!)</td>" +
            "<td>!false: " + ( !false ) +
            "<br />!true: " + ( !true ) + "</td></tr>" );
         document.writeln( "</table>" );
         // -->
      </script>
   </head><body></body>
</html>
```


Demonstrating the Logical Operators - Windows Internet Explorer
C:\examples\ch08\LogicalOperators.html
Google
Demonstrating the Logical Operators
Page
Tools
My Computer
100%

Demonstrating Logical Operators

Logical AND (&&)	false && false: false false && true: false true && false: false true && true: true
Logical OR (||)	false || false: false false || true: true true || false: true true || true: true
Logical NOT (!)	!false: true !true: false

Fig. 8.18 | Demonstrating logical operators. (Part 2 of 2.)

In the output of Fig. 8.18, the strings "false" and "true" indicate false and true for the operands in each condition. The result of the condition is shown as true or false. Note that when you use the concatenation operator with a boolean value and a string, JavaScript automatically converts the boolean value to string "false" or "true". Lines 16–39 build an XHTML table containing the results.

An interesting feature of JavaScript is that most nonboolean values can be converted to a boolean true or false value. Nonzero numeric values are considered to be true. The numeric value zero is considered to be false. Any string that contains characters is considered to be true. The empty string (i.e., the string containing no characters) is considered to be false. The value null and variables that have been declared but not initialized are considered to be false. All objects (e.g., the browser's document and window objects and JavaScript's Math object) are considered to be true.

Figure 8.19 shows the precedence and associativity of the JavaScript operators introduced up to this point. The operators are shown top to bottom in decreasing order of precedence.

Operator	Associativity	Type
`++ -- !`	right to left	unary
`* / %`	left to right	multiplicative
`+ -`	left to right	additive
`< <= > >=`	left to right	relational
`== !=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`= += -= *= /= %=`	right to left	assignment

Fig. 8.19 | Precedence and associativity of the operators discussed so far.

8.10 Summary of Structured Programming

Just as architects design buildings by employing the collective wisdom of their profession, so should programmers design programs. Our field is younger than architecture, and our collective wisdom is considerably sparser. We have learned that structured programming produces programs that are easier to understand than unstructured programs, and thus are easier to test, debug, and modify.

Flowcharts reveal the structured nature of programs or the lack thereof. Connecting individual flowchart symbols arbitrarily can lead to unstructured programs. Therefore, the programming profession has chosen to combine flowchart symbols to form a limited set of control structures and to build structured programs by properly combining control structures in two simple ways.

For simplicity, only single-entry/single-exit control structures are used—that is, there is only one way to enter and one way to exit each control structure. Connecting control structures in sequence to form structured programs is simple: The exit point of one control structure is connected to the entry point of the next control structure (i.e., the control structures are simply placed one after another in a program). We have called this process control-structure stacking. The rules for forming structured programs also allow for control structures to be nested. Figure 8.20 summarizes JavaScript's control structures. Small circles are used in the figure to indicate the single entry point and the single exit point of each structure.

Figure 8.21 shows the rules for forming properly structured programs. The rules assume that the rectangle flowchart symbol may be used to indicate any action, including input/output. [*Note:* An oval flowchart symbol indicates the beginning and end of a process.]

Applying the rules in Fig. 8.21 always results in a structured flowchart with a neat, building-block-like appearance. For example, repeatedly applying Rule 2 to the simplest

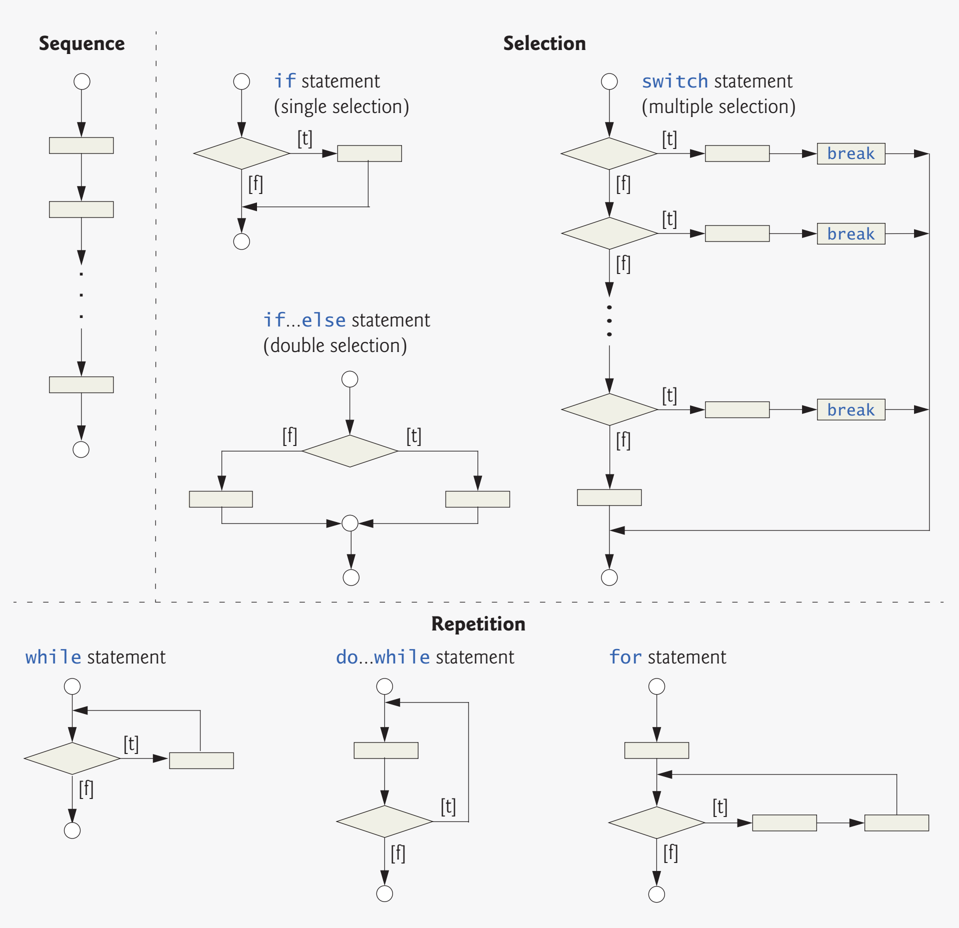


Fig. 8.20 | Single-entry/single-exit sequence, selection and repetition structures.

Rules for forming structured programs

1. Begin with the "simplest flowchart" (Fig. 8.22).
2. Any rectangle (action) can be replaced by two rectangles (actions) in sequence.
3. Any rectangle (action) can be replaced by any control structure (sequence, `if`, `if…else`, `switch`, `while`, `do…while` or `for`).
4. Rules 2 and 3 may be applied as often as necessary and in any order.

Fig. 8.21 | Forming rules for structured programs.

flowchart (Fig. 8.22) results in a structured flowchart containing many rectangles in sequence (Fig. 8.23). Note that Rule 2 generates a stack of control structures; so let us call Rule 2 the **stacking rule**.

Rule 3 is called the **nesting rule**. Repeatedly applying Rule 3 to the simplest flowchart results in a flowchart with neatly nested control structures. For example, in Fig. 8.24, the rectangle in the simplest flowchart is first replaced with a double-selection (`if…else`) structure. Then Rule 3 is applied again to both of the rectangles in the double-selection structure by replacing each of these rectangles with double-selection structures. The

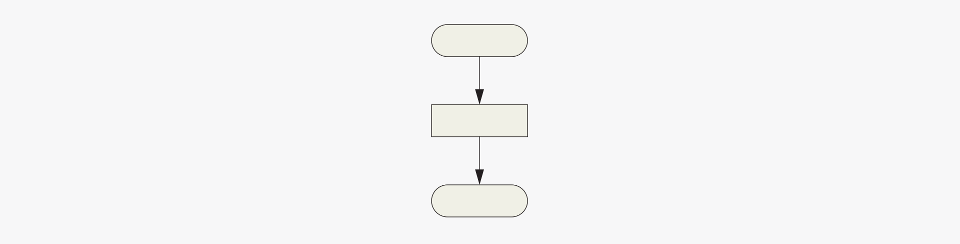


Fig. 8.22 | Simplest flowchart.

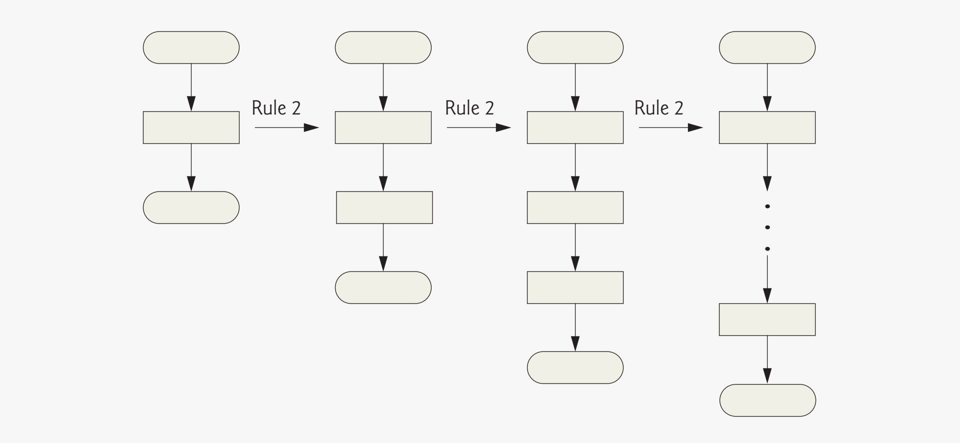


Fig. 8.23 | Repeatedly applying Rule 2 of Fig. 8.21 to the simplest flowchart.

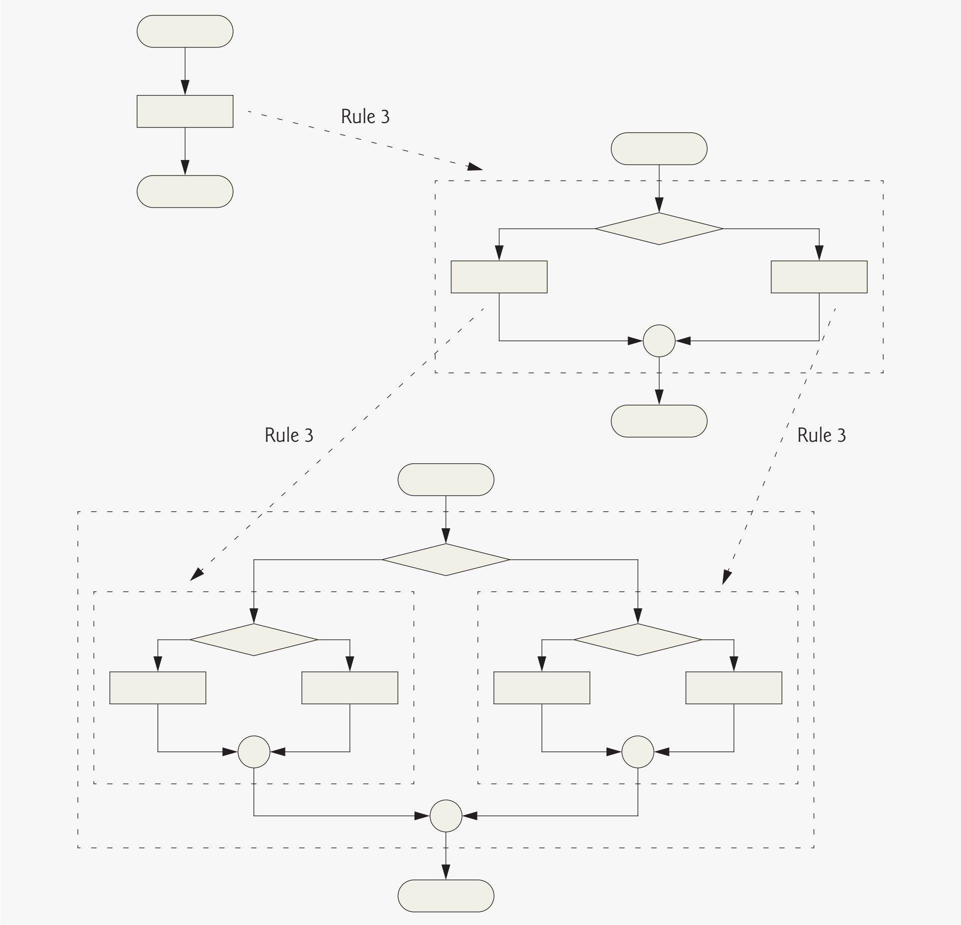


Fig. 8.24 | Applying Rule 3 of Fig. 8.21 to the simplest flowchart.

dashed box around each of the double-selection structures represents the rectangle in the original simplest flowchart that was replaced.

Rule 4 generates larger, more involved and more deeply nested structures. The flowcharts that emerge from applying the rules in Fig. 8.21 constitute the set of all possible structured flowcharts and thus the set of all possible structured programs.

The beauty of the structured approach is that we use only seven simple single-entry/single-exit pieces and assemble them in only two simple ways. Figure 8.25 shows the kinds of stacked building blocks that emerge from applying Rule 2 and the kinds of nested building blocks that emerge from applying Rule 3. The figure also shows the kind of overlapped building blocks that cannot appear in structured flowcharts (because of the elimination of the `goto` statement).

If the rules in Fig. 8.21 are followed, an unstructured flowchart (like the one in Fig. 8.26) cannot be created. If you are uncertain about whether a particular flowchart is structured, apply the rules of Fig. 8.21 in reverse to try to reduce the flowchart to the sim-

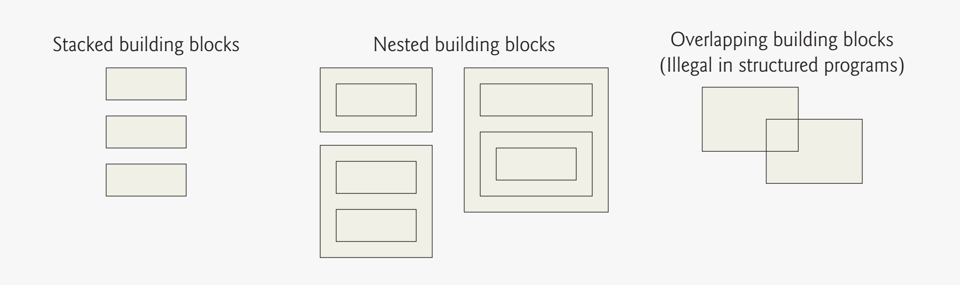


Fig. 8.25 | Stacked, nested and overlapped building blocks.

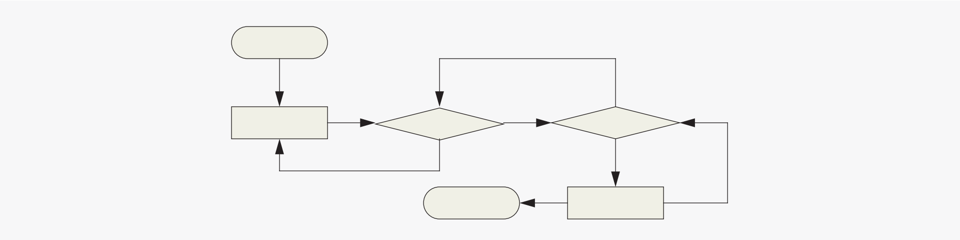


Fig. 8.26 | Unstructured flowchart.

plest flowchart. If the flowchart is reducible to the simplest flowchart, the original flowchart is structured; otherwise, it is not.

Structured programming promotes simplicity. Bohm and Jacopini have given us the result that only three forms of control are needed:

- sequence
- selection
- repetition

Sequence is trivial. Selection is implemented in one of three ways:

- `if` statement (single selection)
- `if`…`else` statement (double selection)
- `switch` statement (multiple selection)

In fact, it is straightforward to prove that the `if` statement is sufficient to provide any form of selection; everything that can be done with the `if…else` statement and the `switch` statement can be implemented by combining `if` statements (although perhaps not as smoothly).

Repetition is implemented in one of four ways:

- `while` statement
- `do`…`while` statement
- `for` statement
- `for`…`in` statement (discussed in Chapter 10)

It is straightforward to prove that the while statement is sufficient to provide any form of repetition. Everything that can be done with the do…while statement and the for statement can be done with the while statement (although perhaps not as elegantly).

Combining these results illustrates that any form of control ever needed in a JavaScript program can be expressed in terms of:

- sequence
- if statement (selection)
- while statement (repetition)

These control structures can be combined in only two ways—stacking and nesting. Indeed, structured programming promotes simplicity.

8.11 Wrap-Up

In this chapter, we discussed the composition of programs from control structures containing actions and decisions. We expanded the previous chapter's discussion on control structures, discussing the for statement, the do…while statement and the switch statement. We also saw how break and continue can be used to alter the flow of the program.

In Chapter 9, we introduce another program-structuring unit, called the function. You'll learn to compose large programs by combining functions that are composed of control structures. We will also discuss how functions promote software reusability.

8.12 Web Resources

www.deitel.com/javascript/

The Deitel JavaScript Resource Center contains links to some of the best JavaScript resources on the web. There you'll find categorized links to JavaScript tools, code generators, forums, books, libraries, frameworks and more. Also check out the tutorials for all skill levels, from introductory to advanced. Be sure to visit the related Resource Centers on XHTML (www.deitel.com/xhtml/) and CSS 2.1 (www.deitel.com/css21/).

Summary

Section 8.2 Essentials of Counter-Controlled Repetition

- Counter-controlled repetition requires: the *name* of a control variable, the *initial value* of the control variable, the *increment* (or *decrement*) by which the control variable is modified each time through the loop, and the condition that tests for the *final value* of the control variable to determine whether looping should continue.
- Because the double-quote character delimits the beginning and end of a string literal in JavaScript, it cannot be used in the contents of the string unless it is preceded by a \ to create the escape sequence \". XHTML allows either single quotes (') or double quotes (") to be placed around the value specified for an attribute. JavaScript allows single quotes to be placed in a string literal.

Section 8.3 for Repetition Statement

- The for statement "does it all"—it specifies each of the items needed for counter-controlled repetition with a control variable.

- You can use a block to put multiple statements into the body of a `for` construct.
- The for statement takes three expressions: an initialization, a condition and an expression.
- The increment expression in the `for` statement acts like a stand-alone statement at the end of the body of the `for` statement.
- Place only expressions involving the control variable in the initialization and increment sections of a `for` statement.
- The three expressions in the `for` statement are optional. The two semicolons in the `for` statement are required.
- The initialization, loop-continuation condition and increment portions of a `for` statement can contain arithmetic expressions.
- The part of a script in which a variable name can be used is known as the variable's scope.
- The "increment" of a `for` statement may be negative, in which case it is called a decrement and the loop actually counts downward.
- If the loop-continuation condition initially is `false`, the body of the `for` statement is not performed. Instead, execution proceeds with the statement following the `for` statement.

Section 8.4 Examples Using the `for` *Statement*

- JavaScript does not include an exponentiation operator. Instead, we use the `Math` object's `pow` method for this purpose. `Math.pow(x, y)` calculates the value of `x` raised to the `y`th power.
- Floating-point numbers can cause trouble as a result of rounding errors.

Section 8.5 `switch` *Multiple-Selection Statement*

- JavaScript provides the `switch` multiple-selection statement in which a variable or expression is tested separately for each of the values it may assume. Different actions are taken for each value.
- The CSS property `list-style-type` allows you to set the numbering system for the list. Possible values include `decimal` (numbers—the default), `lower-roman` (lowercase roman numerals), `upper-roman` (uppercase roman numerals), `lower-alpha` (lowercase letters), `upper-alpha` (uppercase letters), and several others.
- The `switch` statement consists of a series of case labels and an optional default case. When the flow of control reaches the `switch` statement, the script evaluates the controlling expression in the parentheses following keyword `switch`. The value of this expression is compared with the value in each of the `case` labels, starting with the first `case` label. If the comparison evaluates to `true`, the statements after the case label are executed in order until a `break` statement is reached.
- The `break` statement is used as the last statement in each `case` to exit the switch statement immediately.
- The `default` case allows you to specify a set of statements to execute if no other `case` is satisfied. This `case` is usually the last `case` in the `switch` statement.
- Each `case` can have multiple actions (statements). The `switch` statement is different from other statements in that braces are not required around multiple actions in a `case` of a `switch`.
- The `break` statement is not required for the last `case` in the `switch` statement because program control automatically continues with the next statement after the `switch` statement.
- Having several `case` labels listed together (e.g., `case 1: case 2:` with no statements between the cases) simply means that the same set of actions is to occur for each case.

Section 8.6 `do…while` *Repetition Statement*

- The `do…while` statement tests the loop-continuation condition *after* the loop body executes—therefore, *the loop body always executes at least once.*

Section 8.7 `break` *and* `continue` *Statements*

- The `break` statement, when executed in a `while`, `for`, `do…while` or `switch` statement, causes immediate exit from the statement. Execution continues with the first statement after the structure.
- The `break` statement is commonly used to escape early from a loop or to skip the remainder of a `switch` statement.
- The `continue` statement, when executed in a `while`, `for` or `do…while` statement, skips the remaining statements in the body of the statement and proceeds with the next iteration of the loop. In `while` and `do…while` statements, the loop-continuation test evaluates immediately after the `continue` statement executes. In `for` statements, the increment expression executes, then the loop-continuation test evaluates.

Section 8.8 Labeled `break` *and* `continue` *Statements*

- To break out of a nested control statement, you can use the labeled `break` statement. This statement, when executed in a `while`, `for`, `do…while` or `switch` statement, causes immediate exit from that statement and any number of enclosing repetition statements; program execution resumes with the first statement after the specified labeled statement (a statement preceded by a label).
- A labeled statement can be a block (a set of statements enclosed in curly braces, `{}`).
- Labeled `break` statements commonly are used to terminate nested looping structures containing `while`, `for`, `do…while` or `switch` statements.
- The labeled continue statement, when executed in a repetition statement (`while`, `for` or `do…while`), skips the remaining statements in the structure's body and any number of enclosing repetition statements, then proceeds with the next iteration of the specified labeled repetition statement (a repetition statement preceded by a label).
- In labeled `while` and `do…while` statements, the loop-continuation test evaluates immediately after the `continue` statement executes. In a labeled `for` statement, the increment expression executes, then the loop-continuation test evaluates.

Section 8.9 Logical Operators

- JavaScript provides logical operators that can be used to form more complex conditions by combining simple conditions. The logical operators are `&&` (logical AND), `||` (logical OR) and `!` (logical NOT, also called logical negation).
- The && operator is used to ensure that two conditions are *both* `true` before choosing a certain path of execution.
- JavaScript evaluates to `false` or `true` all expressions that include relational operators, equality operators and/or logical operators.
- The `||` (logical OR) operator is used to ensure that either *or* both of two conditions are `true` before choosing choose a certain path of execution.
- The `&&` operator has a higher precedence than the `||` operator. Both operators associate from left to right.
- An expression containing `&&` or `||` operators is evaluated only until truth or falsity is known. This is called short-circuit evaluation.
- JavaScript provides the `!` (logical negation) operator to enable a programmer to "reverse" the meaning of a condition (i.e., a `true` value becomes `false`, and a `false` value becomes `true`).
- The logical negation operator has only a single condition as an operand (i.e., it is a unary operator). The logical negation operator is placed before a condition to evaluate to `true` if the original condition (without the logical negation operator) is `false`.
- The logical negation operator has a higher precedence than the equality operator.

- Most nonboolean values can be converted to a boolean `true` or `false` value. Nonzero numeric values are considered to be `true`. The numeric value zero is considered to be `false`. Any string that contains characters is considered to be `true`. The empty string (i.e., the string containing no characters) is considered to be `false`. The value `null` and variables that have been declared but not initialized are considered to be `false`. All objects (e.g., the browser's `document` and `window` objects and JavaScript's `Math` object) are considered to be `true`.

Terminology

`!` operator
`&&` operator
`||` operator
`break`
`case` label
`continue`
counter-controlled repetition
`default` case in `switch`
`do…while` repetition statement
`for` repetition statement
`for` statement header
infinite loop
labeled `break` statement
labeled block
labeled `continue` statement
labeled repetition statement
logical AND (`&&`)
logical negation (`!`)
logical operator
logical OR (`||`)
loop-continuation condition
`Math` object
multiple selection
nested control structure
nesting rule
`Number` object
off-by-one error
`pow` method of the `Math` object
repetition structure
scope
short-circuit evaluation
simple condition
single-entry/single-exit control structure
stacked control structure
stacking rule
`switch` selection statement
`toFixed` method
truth tables
`while` repetition statement
zero-based counting

Self-Review Exercises

8.1 State whether each of the following is *true* or *false*. If *false*, explain why.

a) The `default` case is required in the `switch` selection statement.
b) The `break` statement is required in the last case of a `switch` selection statement.
c) The expression `( x > y && a < b )` is true if either `x > y` is true or `a < b` is true.
d) An expression containing the `||` operator is true if either or both of its operands is true.

8.2 Write a JavaScript statement or a set of statements to accomplish each of the following tasks:

a) Sum the odd integers between 1 and 99. Use a `for` statement. Assume that the variables `sum` and `count` have been declared.
b) Calculate the value of `2.5` raised to the power of `3`. Use the `pow` method.
c) Print the integers from 1 to 20 by using a `while` loop and the counter variable `x`. Assume that the variable `x` has been declared, but not initialized. Print only five integers per line. [*Hint:* Use the calculation `x % 5`. When the value of this expression is `0`, use `document.write( "<br />" )` to output a line break in the XHTML document.]
d) Repeat Exercise 8.2 (c), but using a `for` statement.

8.3 Find the error in each of the following code segments, and explain how to correct it:

a)
```
x = 1;
while ( x <= 10 );
   ++x;
}
```

b)
```
for ( y = .1; y != 1.0; y += .1 )
   document.write( y + " " );
```
c)
```
switch ( n )
{
   case 1:
      document.writeln( "The number is 1" );
   case 2:
      document.writeln( "The number is 2" );
      break;
   default:
      document.writeln( "The number is not 1 or 2" );
      break;
}
```
d) The following code should print the values from 1 to 10:
```
n = 1;
while ( n < 10 )
   document.writeln( n++ );
```

Answers to Self-Review Exercises

8.1 a) False. The `default` case is optional. If no default action is needed, then there is no need for a `default` case. b) False. The `break` statement is used to exit the `switch` statement. The `break` statement is not required for the last case in a `switch` statement. c) False. Both of the relational expressions must be true in order for the entire expression to be true when using the `&&` operator. d) True.

8.2 a)
```
sum = 0;
for ( count = 1; count <= 99; count += 2 )
   sum += count;
```
b)
```
Math.pow( 2.5, 3 )
```
c)
```
x = 1;
while ( x <= 20 ) {
   document.write( x + " " );
   if ( x % 5 == 0 )
      document.write( "<br />" );
   ++x;
}
```
d)
```
for ( x = 1; x <= 20; x++ ) {
    document.write( x + " " );

   if ( x % 5 == 0 )
      document.write( "<br />" );
}
```
or
```
for ( x = 1; x <= 20; x++ )

   if ( x % 5 == 0 )
      document.write( x + "<br />" );
   else
      document.write( x + " " );
```

8.3 a) Error: The semicolon after the `while` header causes an infinite loop, and there is a missing left brace.
Correction: Replace the semicolon by a `{`, or remove both the `;` and the `}`.

b) Error: Using a floating-point number to control a `for` repetition statement may not work, because floating-point numbers are represented approximately by most computers.
Correction: Use an integer, and perform the proper calculation to get the values you desire:

```
for ( y = 1; y != 10; y++ )
   document.writeln( ( y / 10 ) + " " );
```

c) Error: Missing `break` statement in the statements for the first `case`.
Correction: Add a `break` statement at the end of the statements for the first `case`. Note that this missing statement is not necessarily an error if the programmer wants the statement of `case 2:` to execute every time the `case 1:` statement executes.

d) Error: Improper relational operator used in the `while` continuation condition.
Correction: Use `<=` rather than `<`, or change `10` to `11`.

Exercises

8.4 Find the error in each of the following segments of code. [*Note:* There may be more than one error.]

a)
```
For ( x = 100, x >= 1, x++ )
   document.writeln( x );
```

b) The following code should print whether integer value is odd or even:
```
switch ( value % 2 ) {
   case 0:
      document.writeln( "Even integer" );
   case 1:
      document.writeln( "Odd integer" );
}
```

c) The following code should output the odd integers from 19 to 1:
```
for ( x = 19; x >= 1; x += 2 )
   document.writeln( x );
```

d) The following code should output the even integers from 2 to 100:
```
counter = 2;
do {
   document.writeln( counter );
   counter += 2;
} While ( counter < 100 );
```

8.5 What does the following script do?

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 8.5: ex08_05.html -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head><title>Mystery</title>
      <script type = "text/javascript">
         <!--
         document.writeln( "<table>" );

         for ( var i = 1; i <= 10; i++ )
         {
            document.writeln( "<tr>" );

            for ( var j = 1; j <= 5; j++ )
               document.writeln( "<td>(" + i + ", " + j + ")</td>" );

            document.writeln( "</tr>" );
         } // end for

         document.writeln( "</table>" );
         -->
      </script>
   </head><body />
</html>
```

8.6 Write a script that finds the smallest of several non-negative integers. Assume that the first value read specifies the number of values to be input from the user.

8.7 Write a script that calculates the product of the odd integers from 1 to 15 then outputs XHTML text that displays the results.

8.8 Modify the compound interest program in Fig. 8.6 to repeat its steps for interest rates of 5, 6, 7, 8, 9 and 10 percent. Use a `for` statement to vary the interest rate. Use a separate table for each rate.

8.9 Write a script that outputs XHTML to display the given patterns separately, one below the other. Use `for` statements to generate the patterns. All asterisks (*) should be printed by a single statement of the form `document.write( "*" );` (this causes the asterisks to print side by side). A statement of the form `document.writeln( "<br />" );` can be used to position to the next line. A statement of the form `document.write( " " );` can be used to display a space (needed for the last two patterns). There should be no other output statements in the program. [*Hint:* The last two patterns require that each line begin with an appropriate number of blanks. You may need to use the XHTML `<pre></pre>` tags.]

```
(a)              (b)              (c)              (d)
*                **********       **********                *
**               *********         *********               **
***              ********           ********              ***
****             *******             *******             ****
*****            ******               ******            *****
******           *****                 *****           ******
*******          ****                   ****          *******
********         ***                     ***         ********
*********        **                       **        *********
**********       *                         *       **********
```

8.10 One interesting application of computers is the drawing of graphs and bar charts (sometimes called histograms). Write a script that reads five numbers between 1 and 30. For each number read, output XHTML text that displays a line containing the same number of adjacent asterisks. For example, if your program reads the number 7, it should output XHTML text that displays *******.

8.11 *("The Twelve Days of Christmas" Song)* Write a script that uses repetition and a `switch` structures to print the song "The Twelve Days of Christmas." You can find the words at the site

`www.santas.net/twelvedaysofchristmas.htm`

8.12 A mail-order house sells five different products whose retail prices are as follows: product 1, $2.98; product 2, $4.50; product 3, $9.98; product 4, $4.49; and product 5, $6.87. Write a script that reads a series of pairs of numbers as follows:

a) Product number
b) Quantity sold for one day

Your program should use a `switch` statement to determine each product's retail price and should calculate and output XHTML that displays the total retail value of all the products sold last week. Use a `prompt` dialog to obtain the product number and quantity from the user. Use a sentinel-controlled loop to determine when the program should stop looping and display the final results.

8.13 Assume that `i = 1`, `j = 2`, `k = 3` and `m = 2`. What does each of the given statements print? Are the parentheses necessary in each case?

a) `document.writeln( i == 1 );`
b) `document.writeln( j == 3 );`
c) `document.writeln( i >= 1 && j < 4 );`
d) `document.writeln( m <= 99 && k < m );`
e) `document.writeln( j >= i || k == m );`
f) `document.writeln( k + m < j || 3 - j >= k );`
g) `document.writeln( !( k > m ) );`

8.14 Modify Exercise 8.9 to combine your code from the four separate triangles of asterisks into a single script that prints all four patterns side by side, making clever use of nested `for` statements.

```
*           **********  **********           *
**          *********    *********          **
***         ********      ********         ***
****        *******        *******        ****
*****       ******          ******       *****
******      *****            *****      ******
*******     ****              ****     *******
********    ***                ***    ********
*********   **                  **   *********
**********  *                    *  **********
```

8.15 *(De Morgan's Laws)* In this chapter, we have discussed the logical operators `&&`, `||` and `!`. De Morgan's Laws can sometimes make it more convenient for us to express a logical expression. These laws state that the expression `!(`*condition1* `&&` *condition2*`)` is logically equivalent to the expression `(!`*condition1* `||` `!`*condition2*`)`. Also, the expression `!(`*condition1* `||` *condition2*`)` is logically equivalent to the expression `(!`*condition1* `&&` `!`*condition2*`)`. Use De Morgan's Laws to write equivalent expressions for each of the following, then write a program to show that the original expression and the new expression are equivalent in each case:

a) `!( x < 5 ) && !( y >= 7 )`
b) `!( a == b ) || !( g != 5 )`
c) `!( ( x <= 8 ) && ( y > 4 ) )`
d) `!( ( i > 4 ) || ( j <= 6 ) )`

8.16 Write a script that prints the following diamond shape:

```
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

You may use output statements that print a single asterisk (*), a single space or a single newline character. Maximize your use of repetition (with nested for statements), and minimize the number of output statements.

8.17 Modify the program you wrote in Exercise 8.16 to read an odd number in the range 1 to 19. This number specifies the number of rows in the diamond. Your program should then display a diamond of the appropriate size.

8.18 A criticism of the break statement and the continue statement is that each is unstructured. Actually, break statements and continue statements can always be replaced by structured statements, although coding the replacement can be awkward. Describe in general how you would remove any break statement from a loop in a program and replace it with some structured equivalent. [*Hint:* The break statement "jumps out of" a loop from the body of that loop. The other way to leave is by failing the loop-continuation test. Consider using in the loop-continuation test a second test that indicates "early exit because of a 'break' condition."] Use the technique you develop here to remove the break statement from the program in Fig. 8.11.

8.19 What does the following script do?

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 8.19: ex08_19.html -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head><title>Mystery</title>
      <script type = "text/javascript">
         <!--
         for ( var i = 1; i <= 5; i++ )
         {
            for ( var j = 1; j <= 3; j++ )
            {
               for ( var k = 1; k <= 4; k++ )
                  document.write( "*" );
               document.writeln( "<br />" );
            } // end for
            document.writeln( "<br />" );
         } // end for
         // -->
      </script>
   </head><body></body>
</html>
```

8.20 Describe in general how you would remove any continue statement from a loop in a program and replace it with some structured equivalent. Use the technique you develop to remove the continue statement from the program in Fig. 8.12.

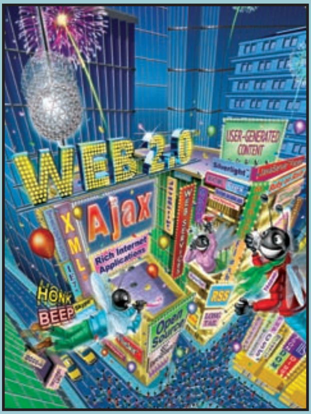
8.21 Given the following switch statement:

```
switch ( k )
{
   case 1:
      break;
   case 2:
   case 3:
      ++k;
      break;
   case 4:
      --k;
      break;
   default:
      k *= 3;
} //end switch

x = k;
```

What values are assigned to x when k has values of 1, 2, 3, 4 and 10?

9

JavaScript: Functions



Form ever follows function.
—Louis Sullivan

E pluribus unum.
(One composed of many.)
—Virgil

O! call back yesterday, bid time return.
—William Shakespeare

Call me Ishmael.
—Herman Melville

When you call me that, smile.
—Owen Wister

OBJECTIVES

In this chapter you will learn:

- To construct programs modularly from small pieces called functions.
- To create new functions.
- How to pass information between functions.
- Simulation techniques that use random number generation.
- How the visibility of identifiers is limited to specific regions of programs.

Outline

9.1 Introduction

Most computer programs that solve real-world problems are much larger than the programs presented in the first few chapters of this book. Experience has shown that the best way to develop and maintain a large program is to construct it from small, simple pieces, or **modules**. This technique is called **divide and conquer**. This chapter describes many key features of JavaScript that facilitate the design, implementation, operation and maintenance of large scripts.

9.2 Program Modules in JavaScript

Modules in JavaScript are called **functions**. JavaScript programs are written by combining new functions that the programmer writes with "prepackaged" functions and objects available in JavaScript. The prepackaged functions that belong to JavaScript objects (such as `Math.pow` and `Math.round`, introduced previously) are called **methods**. The term method implies that the function belongs to a particular object. We refer to functions that belong to a particular JavaScript object as methods; all others are referred to as functions.

JavaScript provides several objects that have a rich collection of methods for performing common mathematical calculations, string manipulations, date and time manipulations, and manipulations of collections of data called arrays. These objects make your job easier, because they provide many of the capabilities programmers frequently need. Some common predefined objects of JavaScript and their methods are discussed in Chapter 10, JavaScript: Arrays, and Chapter 11, JavaScript: Objects.

Good Programming Practice 9.1

Familiarize yourself with the rich collection of objects and methods provided by JavaScript.

Software Engineering Observation 9.1

Avoid reinventing the wheel. Use existing JavaScript objects, methods and functions instead of writing new ones. This reduces script-development time and helps avoid introducing errors.

Portability Tip 9.1

Using the methods built into JavaScript objects helps make scripts more portable.

You can write functions to define specific tasks that may be used at many points in a script. These functions are referred to as **programmer-defined functions**. The actual statements defining the function are written only once and are hidden from other functions.

A function is **invoked** (i.e., made to perform its designated task) by a **function call**. The function call specifies the function name and provides information (as **arguments**) that the called function needs to perform its task. A common analogy for this structure is the hierarchical form of management. A boss (the **calling function**, or **caller**) asks a worker (the **called function**) to perform a task and **return** (i.e., report back) the results when the task is done. The boss function does not know how the worker function performs its designated tasks. The worker may call other worker functions—the boss will be unaware of this. We'll soon see how this "hiding" of implementation details promotes good software engineering. Figure 9.1 shows the `boss` function communicating with several worker functions in a hierarchical manner. Note that `worker1` acts as a "boss" function to `worker4` and `worker5`, and `worker4` and `worker5` report back to `worker1`.

Functions are invoked by writing the name of the function, followed by a left parenthesis, followed by a comma-separated list of zero or more arguments, followed by a right parenthesis. For example, a programmer desiring to convert a string stored in variable `inputValue` to a floating-point number and add it to variable `total` might write

```
total += parseFloat( inputValue );
```

When this statement executes, JavaScript function **`parseFloat`** converts the string in the `inputValue` variable to a floating-point value and adds that value to `total`. Variable `inputValue` is function `parseFloat`'s argument. Function `parseFloat` takes a string representation of a floating-point number as an argument and returns the corresponding floating-point numeric value. Function arguments may be constants, variables or expressions.

Methods are called in the same way, but require the name of the object to which the method belongs and a dot preceding the method name. For example, we've already seen the syntax `document.writeln("Hi there.");`. This statement calls the `document` object's `writeln` method to output the text.

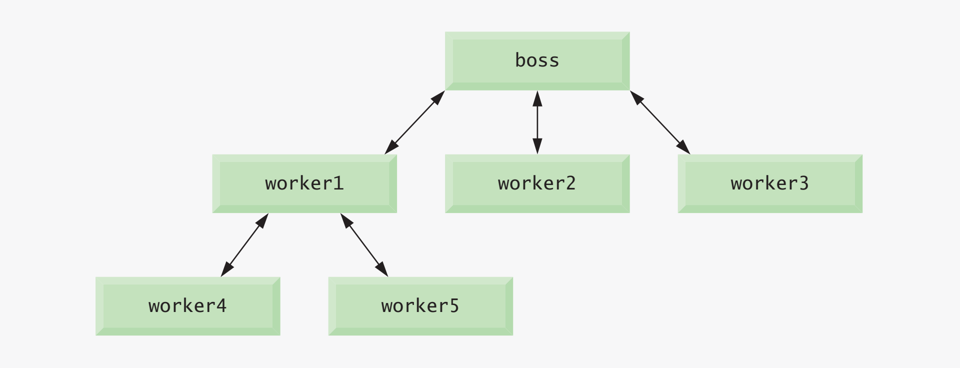


Fig. 9.1 | Hierarchical boss-function/worker-function relationship.

9.3 Programmer-Defined Functions

Functions allow you to modularize a program. All variables declared in function definitions are local variables—this means that they can be accessed only in the function in which they are defined. Most functions have a list of parameters that provide the means for communicating information between functions via function calls. A function's parameters are also considered to be local variables. When a function is called, the arguments in the function call are assigned to the corresponding parameters in the function definition.

There are several reasons for modularizing a program with functions. The divide-and-conquer approach makes program development more manageable. Another reason is software reusability (i.e., using existing functions as building blocks to create new programs). With good function naming and definition, programs can be created from standardized functions rather than built by using customized code. For example, we did not have to define how to convert strings to integers and floating-point numbers—JavaScript already provides function `parseInt` to convert a string to an integer and function `parseFloat` to convert a string to a floating-point number. A third reason is to avoid repeating code in a program. Code that is packaged as a function can be executed from several locations in a program by calling the function.

Software Engineering Observation 9.2

If a function's task cannot be expressed concisely, perhaps the function is performing too many different tasks. It is usually best to break such a function into several smaller functions.

9.4 Function Definitions

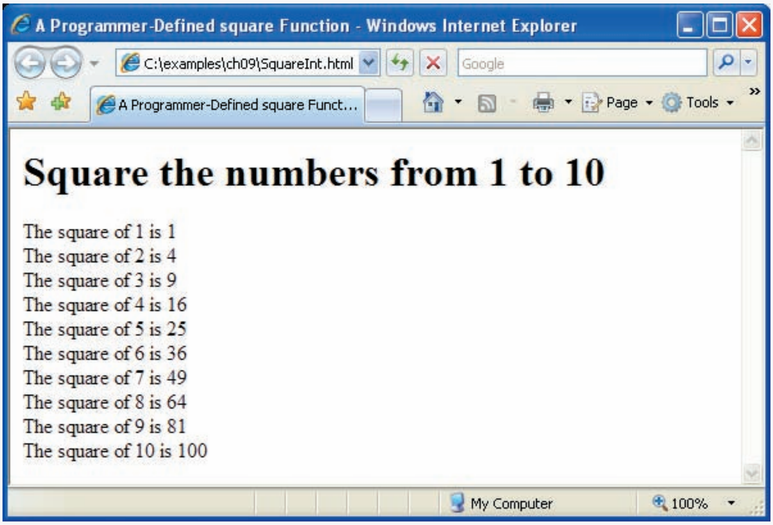
Each script we have presented thus far in the text has consisted of a series of statements and control structures in sequence. These scripts have been executed as the browser loads the web page and evaluates the `<head>` section of the page. We now consider how you can write your own customized functions and call them in a script.

Programmer-Defined Function `square`

Consider a script (Fig. 9.2) that uses a function `square` to calculate the squares of the integers from 1 to 10. [*Note:* We continue to show many examples in which the `body` element of the XHTML document is empty and the document is created directly by JavaScript. In later chapters, we show many examples in which JavaScripts interact with the elements in the `body` of a document.]

The `for` statement in lines 15–17 outputs XHTML that displays the results of squaring the integers from 1 to 10. Each iteration of the loop calculates the `square` of the current value of control variable `x` and outputs the result by writing a line in the XHTML document. Function `square` is invoked, or called, in line 17 with the expression `square(x)`. When program control reaches this expression, the program calls function `square` (defined in lines 23–26). The parentheses `()` represent the function-call operator, which has high precedence. At this point, the program makes a copy of the value of `x` (the argument) and program control transfers to the first line of function `square`. Function `square` receives the copy of the value of `x` and stores it in the parameter `y`. Then `square` calculates `y * y`. The result is passed back (returned) to the point in line 17 where `square` was invoked. Lines 16–17 concatenate `"The square of "`, the value of `x`, the string `" is "`,

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 9.2: SquareInt.html -->
<!-- Programmer-defined function square. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>A Programmer-Defined square Function</title>
      <script type = "text/javascript">
         <!--
         document.writeln( "<h1>Square the numbers from 1 to 10</h1>" );

         // square the numbers from 1 to 10
         for ( var x = 1; x <= 10; x++ )
            document.writeln( "The square of " + x + " is " +
               square( x ) + "<br />" );

         // The following square function definition is executed
         // only when the function is explicitly called.

         // square function definition
         function square( y )
         {
            return y * y;
         } // end function square
         // -->
      </script>
   </head><body></body>
</html>
```


A Programmer-Defined square Function - Windows Internet Explorer
C:\examples\ch09\SquareInt.html
Google
A Programmer-Defined square Funct...
Page
Tools
My Computer
100%

Square the numbers from 1 to 10

```
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
The square of 6 is 36
The square of 7 is 49
The square of 8 is 64
The square of 9 is 81
The square of 10 is 100
```

Fig. 9.2 | Programmer-defined function square.

the value returned by function `square` and a `<br />` tag and write that line of text in the XHTML document. This process is repeated 10 times.

The definition of function `square` (lines 23–26) shows that `square` expects a single parameter `y`. Function `square` uses this name in its body to manipulate the value passed to `square` from line 17. The **return statement** in `square` passes the result of the calculation `y * y` back to the calling function. Note that JavaScript keyword `var` is not used to declare variables in the parameter list of a function.

Common Programming Error 9.1

Using the JavaScript `var` *keyword to declare a variable in a function parameter list results in a JavaScript runtime error.*

In this example, function `square` follows the rest of the script. When the `for` statement terminates, program control does *not* flow sequentially into function `square`. A function must be called explicitly for the code in its body to execute. Thus, when the `for` statement terminates in this example, the script terminates.

Good Programming Practice 9.2

Place a blank line between function definitions to separate the functions and enhance program readability.

Software Engineering Observation 9.3

Statements that are enclosed in the body of a function definition are not executed by the JavaScript interpreter unless the function is invoked explicitly.

The format of a function definition is

```
function function-name( parameter-list )
{
    declarations and statements
}
```

The *function-name* is any valid identifier. The *parameter-list* is a comma-separated list containing the names of the parameters received by the function when it is called (remember that the arguments in the function call are assigned to the corresponding parameter in the function definition). There should be one argument in the function call for each parameter in the function definition. If a function does not receive any values, the *parameter-list* is empty (i.e., the function name is followed by an empty set of parentheses). The *declarations* and *statements* in braces form the **function body**.

Common Programming Error 9.2

Forgetting to return a value from a function that is supposed to return a value is a logic error.

Common Programming Error 9.3

Placing a semicolon after the right parenthesis enclosing the parameter list of a function definition results in a JavaScript runtime error.

Common Programming Error 9.4

Redefining a function parameter as a local variable in the function is a logic error.

Common Programming Error 9.5

Passing to a function an argument that is not compatible with the corresponding parameter's expected type is a logic error and may result in a JavaScript runtime error.

Good Programming Practice 9.3

Although it is not incorrect to do so, do not use the same name for an argument passed to a function and the corresponding parameter in the function definition. Using different names avoids ambiguity.

Software Engineering Observation 9.4

To promote software reusability, every function should be limited to performing a single, well-defined task, and the name of the function should express that task effectively. Such functions make programs easier to write, debug, maintain and modify.

Error-Prevention Tip 9.1

A small function that performs one task is easier to test and debug than a larger function that performs many tasks.

There are three ways to return control to the point at which a function was invoked. If the function does not return a result, control returns when the program reaches the function-ending right brace or by executing the statement

```
return;
```

If the function does return a result, the statement

```
return expression;
```

returns the value of *expression* to the caller. When a `return` statement is executed, control returns immediately to the point at which the function was invoked.

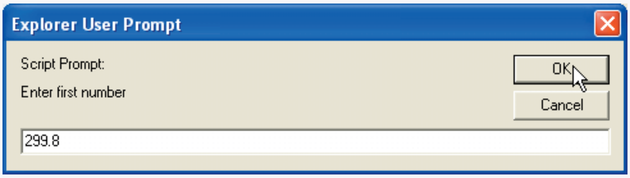
Programmer-Defined Function `maximum`

The script in our next example (Fig. 9.3) uses a programmer-defined function called `maximum` to determine and return the largest of three floating-point values.

The three floating-point values are input by the user via `prompt` dialogs (lines 12–14). Lines 16–18 use function `parseFloat` to convert the strings entered by the user to floating-point values. The statement in line 20 passes the three floating-point values to function `maximum` (defined in lines 28–31), which determines the largest floating-point value. This value is returned to line 20 by the `return` statement in function `maximum`. The value returned is assigned to variable `maxValue`. Lines 22–25 display the three floating-point values input by the user and the calculated `maxValue`.

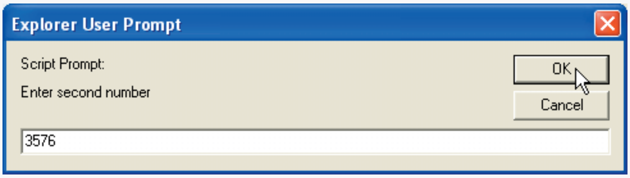
Note the implementation of the function `maximum` (lines 28–31). The first line indicates that the function's name is `maximum` and that the function takes three parameters (`x`, `y` and `z`) to accomplish its task. Also, the body of the function contains the statement which returns the largest of the three floating-point values, using two calls to the `Math` object's `max` method. First, method `Math.max` is invoked with the values of variables `y` and `z` to determine the larger of the two values. Next, the value of variable `x` and the result of the first call to `Math.max` are passed to method `Math.max`. Finally, the result of the second call to `Math.max` is returned to the point at which `maximum` was invoked (i.e., line 20). Note

```
1  <?xml version = "1.0" encoding = "utf-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4  
5  <!-- Fig. 9.3: maximum.html -->
6  <!-- Programmer-Defined maximum function. -->
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8     <head>
9        <title>Finding the Maximum of Three Values</title>
10       <script type = "text/javascript">
11          <!--
12          var input1 = window.prompt( "Enter first number", "0" );
13          var input2 = window.prompt( "Enter second number", "0" );
14          var input3 = window.prompt( "Enter third number", "0" );
15 
16          var value1 = parseFloat( input1 );
17          var value2 = parseFloat( input2 );
18          var value3 = parseFloat( input3 );
19 
20          var maxValue = maximum( value1, value2, value3 );
21 
22          document.writeln( "First number: " + value1 +
23             "<br />Second number: " + value2 +
24             "<br />Third number: " + value3 +
25             "<br />Maximum is: " + maxValue );
26 
27          // maximum function definition (called from line 20)
28          function maximum( x, y, z )
29          {
30             return Math.max( x, Math.max( y, z ) );
31          } // end function maximum
32          // -->
33       </script>
34    </head>
35    <body>
36       <p>Click Refresh (or Reload) to run the script again</p>
37    </body>
38 </html>
```



Script Prompt:
Enter first number

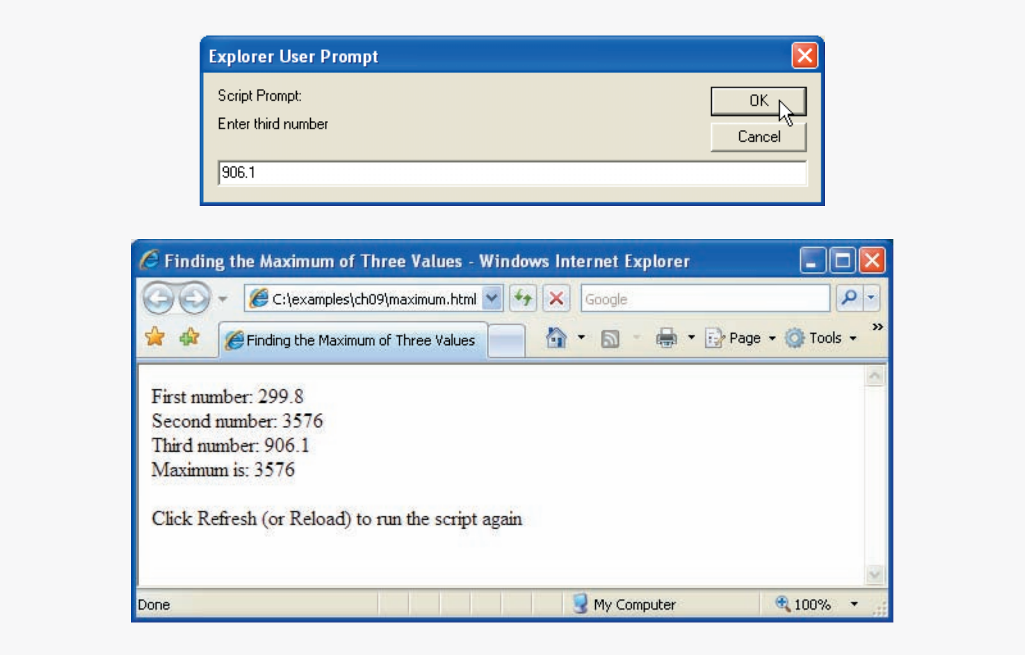
299.8



Script Prompt:
Enter second number

3576

Fig. 9.3 | Programmer-defined `maximum` function. (Part 1 of 2.)


Explorer User Prompt
OK
Cancel
Finding the Maximum of Three Values - Windows Internet Explorer
C:\examples\ch09\maximum.html
Google
Finding the Maximum of Three Values
Page
Tools
Done
My Computer
100%

Script Prompt:
Enter third number

906.1

```
First number: 299.8
Second number: 3576
Third number: 906.1
Maximum is: 3576

Click Refresh (or Reload) to run the script again
```

Fig. 9.3 | Programmer-defined `maximum` function. (Part 2 of 2.)

once again that the script terminates before sequentially reaching the definition of function `maximum`. The statement in the body of function `maximum` executes only when the function is invoked from line 20.

9.5 Random Number Generation

We now take a brief and, it is hoped, entertaining diversion into a popular programming application, namely simulation and game playing. In this section and the next, we develop a nicely structured game-playing program that includes multiple functions. The program uses most of the control structures we have studied.

There is something in the air of a gambling casino that invigorates people, from the high rollers at the plush mahogany-and-felt craps tables to the quarter poppers at the one-armed bandits. It is the element of chance, the possibility that luck will convert a pocketful of money into a mountain of wealth. The element of chance can be introduced through the `Math` object's **`random` method**. (Remember, we are calling `random` a method because it belongs to the `Math` object.)

Consider the following statement:

```
var randomValue = Math.random();
```

Method `random` generates a floating-point value from 0.0 up to, but not including, 1.0. If `random` truly produces values at random, then every value from 0.0 up to, but not including, 1.0 has an equal chance (or probability) of being chosen each time `random` is called.

The range of values produced directly by `random` is often different than what is needed in a specific application. For example, a program that simulates coin tossing might require only 0 for heads and 1 for tails. A program that simulates rolling a six-sided die would require random integers in the range from 1 to 6. A program that randomly predicts the next type of spaceship, out of four possibilities, that will fly across the horizon in a video game might require random integers in the range 0–3 or 1–4.

To demonstrate method `random`, let us develop a program (Fig. 9.4) that simulates 20 rolls of a six-sided die and displays the value of each roll. We use the multiplication operator (*) with `random` as follows:

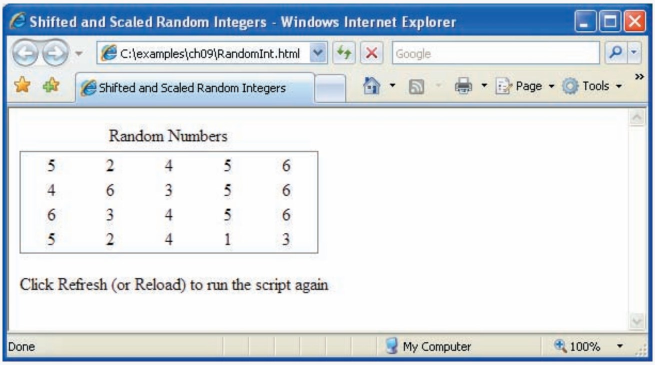
```
Math.floor( 1 + Math.random() * 6 )
```

First, the preceding expression multiplies the result of a call to `Math.random()` by 6 to produce a number in the range 0.0 up to, but not including, 6.0. This is called scaling the range of the random numbers. Next, we add 1 to the result to shift the range of numbers to produce a number in the range 1.0 up to, but not including, 7.0. Finally, we use method **`Math.floor`** to *round* the result down to the closest integer not greater than the argument's value—for example, 1.75 is rounded to 1. Figure 9.4 confirms that the results are in the range 1 to 6.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 9.4: RandomInt.html -->
<!-- Random integers, shifting and scaling. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Shifted and Scaled Random Integers</title>
      <style type = "text/css">
         table { width: 50%;
                 border: 1px solid gray;
                 text-align: center }
      </style>
      <script type = "text/javascript">
         <!--
         var value;

         document.writeln( "<table>" );
         document.writeln( "<caption>Random Numbers</caption><tr>" );

         for ( var i = 1; i <= 20; i++ )
         {
            value = Math.floor( 1 + Math.random() * 6 );
            document.writeln( "<td>" + value + "</td>" );

            // start a new table row every 5 entries
            if ( i % 5 == 0 && i != 20 )
               document.writeln( "</tr><tr>" );
         } // end for

         document.writeln( "</tr></table>" );
```

Fig. 9.4 | Random integers, shifting and scaling. (Part 1 of 2.)

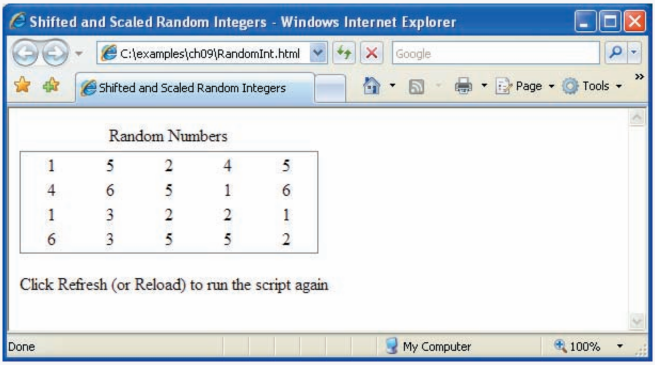
```
         // -->
      </script>
   </head>
   <body>
      <p>Click Refresh (or Reload) to run the script again</p>
   </body>
</html>
```


Shifted and Scaled Random Integers - Windows Internet Explorer
C:\examples\ch09\RandomInt.html
Google
Shifted and Scaled Random Integers
Page
Tools

Random Numbers

5	2	4	5	6
4	6	3	5	6
6	3	4	5	6
5	2	4	1	3

Click Refresh (or Reload) to run the script again


Shifted and Scaled Random Integers - Windows Internet Explorer
C:\examples\ch09\RandomInt.html
Google
Shifted and Scaled Random Integers
Page
Tools

Random Numbers

1	5	2	4	5
4	6	5	1	6
1	3	2	2	1
6	3	5	5	2

Click Refresh (or Reload) to run the script again

Fig. 9.4 | Random integers, shifting and scaling. (Part 2 of 2.)

To show that these numbers occur with approximately equal likelihood, let us simulate 6000 rolls of a die with the program in Fig. 9.5. Each integer from 1 to 6 should appear approximately 1000 times. Use your browser's **Refresh** (or **Reload**) button to execute the script again.

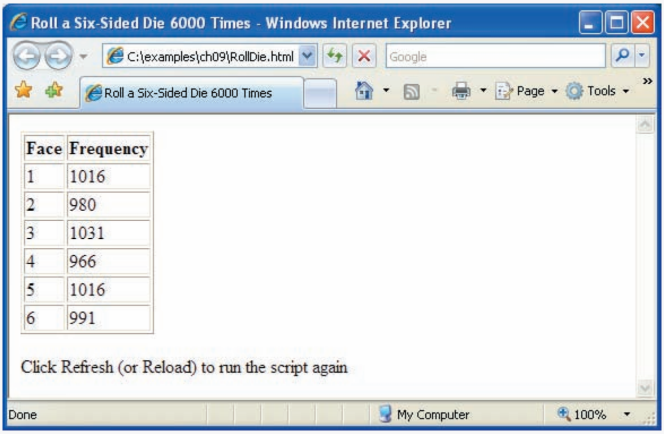
```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 9.5: RollDie.html -->
<!-- Rolling a Six-Sided Die 6000 times. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
```

Fig. 9.5 | Rolling a six-sided die 6000 times. (Part 1 of 3.)

```
   <head>
      <title>Roll a Six-Sided Die 6000 Times</title>
      <script type = "text/javascript">
         <!--
         var frequency1 = 0;
         var frequency2 = 0;
         var frequency3 = 0;
         var frequency4 = 0;
         var frequency5 = 0;
         var frequency6 = 0;
         var face;

         // roll die 6000 times and accumulate results
         for ( var roll = 1; roll <= 6000; roll++ )
         {
            face = Math.floor( 1 + Math.random() * 6 );

            switch ( face )
            {
               case 1:
                  ++frequency1;
                  break;
               case 2:
                  ++frequency2;
                  break;
               case 3:
                  ++frequency3;
                  break;
               case 4:
                  ++frequency4;
                  break;
               case 5:
                  ++frequency5;
                  break;
               case 6:
                  ++frequency6;
                  break;
            } // end switch
         } // end for

         document.writeln( "<table border = \"1\">" );
         document.writeln( "<thead><th>Face</th>" +
            "<th>Frequency</th></thead>" );
         document.writeln( "<tbody><tr><td>1</td><td>" +
            frequency1 + "</td></tr>" );
         document.writeln( "<tr><td>2</td><td>" + frequency2 +
            "</td></tr>" );
         document.writeln( "<tr><td>3</td><td>" + frequency3 +
            "</td></tr>" );
         document.writeln( "<tr><td>4</td><td>" + frequency4 +
            "</td></tr>" );
         document.writeln( "<tr><td>5</td><td>" + frequency5 +
            "</td></tr>" );
```

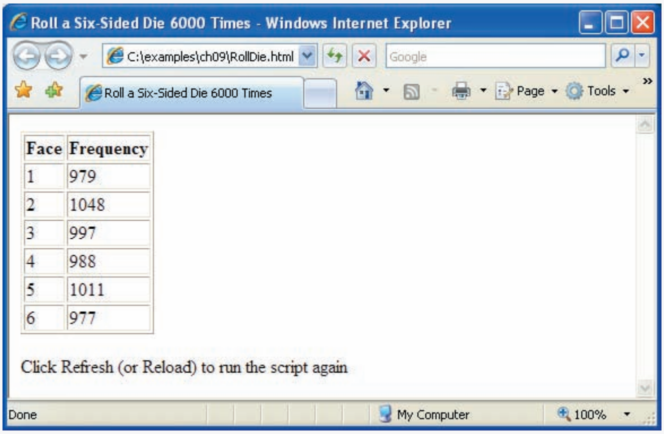
Fig. 9.5 | Rolling a six-sided die 6000 times. (Part 2 of 3.)

```
61            document.writeln( "<tr><td>6</td><td>" + frequency6 +
62               "</td></tr></tbody></table>" );
63            // -->
64         </script>
65      </head>
66      <body>
67         <p>Click Refresh (or Reload) to run the script again</p>
68      </body>
69   </html>
```


Roll a Six-Sided Die 6000 Times - Windows Internet Explorer
C:\examples\ch09\RollDie.html
Google
Roll a Six-Sided Die 6000 Times
Page
Tools
Done
My Computer
100%

Face	Frequency
1	1016
2	980
3	1031
4	966
5	1016
6	991

Click Refresh (or Reload) to run the script again


Roll a Six-Sided Die 6000 Times - Windows Internet Explorer
C:\examples\ch09\RollDie.html
Google
Roll a Six-Sided Die 6000 Times
Page
Tools
Done
My Computer
100%

Face	Frequency
1	979
2	1048
3	997
4	988
5	1011
6	977

Click Refresh (or Reload) to run the script again

Fig. 9.5 | Rolling a six-sided die 6000 times. (Part 3 of 3.)

As the output of the program shows, we used `Math` method `random` and the scaling and shifting techniques of the previous example to simulate the rolling of a six-sided die. Note that we used nested control structures to determine the number of times each side of the six-sided die occurred. Lines 12–17 declare and initialize counter variables to keep track of the number of times each of the six die values appears. Line 18 declares a variable to store the face value of the die. The `for` statement in lines 21–46 iterates 6000 times. During each iteration of the loop, line 23 produces a value from 1 to 6, which is stored in face. The nested switch statement in lines 25–45 uses the face value that was randomly chosen as its controlling expression. Based on the value of face, the program increments one of the six counter variables during each iteration of the loop. Note that no default case is provided in this switch statement, because the statement in line 23 produces only the values 1, 2, 3, 4, 5 and 6. In this example, the default case would never execute. After we study Arrays in Chapter 10, we discuss a way to replace the entire switch statement in this program with a single-line statement.

Run the program several times, and observe the results. Note that the program produces different random numbers each time the script executes, so the results should vary.

The values returned by random are always in the range

```
0.0 ≤ Math.random() < 1.0
```

Previously, we demonstrated the statement

```
face = Math.floor( 1 + Math.random() * 6 );
```

which simulates the rolling of a six-sided die. This statement always assigns an integer (at random) to variable face, in the range 1 ≤ face ≤ 6. Note that the width of this range (i.e., the number of consecutive integers in the range) is 6, and the starting number in the range is 1. Referring to the preceding statement, we see that the width of the range is determined by the number used to scale random with the multiplication operator (6 in the preceding statement) and that the starting number of the range is equal to the number (1 in the preceding statement) added to Math.random() * 6. We can generalize this result as

```
face = Math.floor( a + Math.random() * b );
```

where a is the **shifting value** (which is equal to the first number in the desired range of consecutive integers) and b is the **scaling factor** (which is equal to the width of the desired range of consecutive integers). In this chapter's exercises, you'll see that it's possible to choose integers at random from sets of values other than ranges of consecutive integers.

9.6 Example: Game of Chance

One of the most popular games of chance is a dice game known as craps, which is played in casinos and back alleys throughout the world. The rules of the game are straightforward:

> *A player rolls two dice. Each die has six faces. These faces contain one, two, three, four, five and six spots, respectively. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, the player wins. If the sum is 2, 3 or 12 on the first throw (called "craps"), the player loses (i.e., the "house" wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first throw, that sum becomes the player's "point." To win, you must continue rolling the dice until you "make your point" (i.e., roll your point value). You lose by rolling a 7 before making the point.*

The script in Fig. 9.6 simulates the game of craps.

Note that the player must roll two dice on the first and all subsequent rolls. When you execute the script, click the **Roll Dice** button to play the game. A message below the **Roll Dice** button displays the status of the game after each roll.

Until now, all user interactions with scripts have been through either a prompt dialog (in which the user types an input value for the program) or an alert dialog (in which a message is displayed to the user, and the user can click **OK** to dismiss the dialog). Although these dialogs are valid ways to receive input from a user and to display messages, they are fairly limited in their capabilities. A `prompt` dialog can obtain only one value at a time from the user, and a message dialog can display only one message.

More frequently, multiple inputs are received from the user at once via an XHTML form (such as one in which the user enters name and address information) or to display many pieces of data at once (e.g., the values of the dice, the sum of the dice and the point in this example). To begin our introduction to more elaborate user interfaces, this program uses an XHTML form (discussed in Chapter 4) and a new graphical user interface concept—GUI **event handling**. This is our first example in which the JavaScript executes in response to the user's interaction with a GUI component in an XHTML form. This interaction causes an event. Scripts are often used to respond to events.

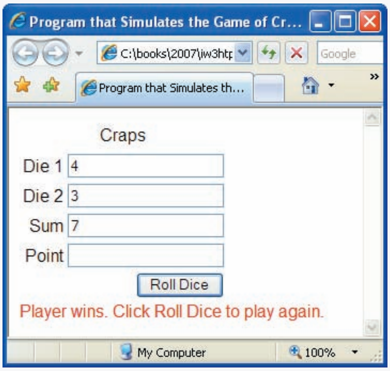
```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 9.6: Craps.html -->
<!-- Craps game simulation. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Program that Simulates the Game of Craps</title>
      <style type = "text/css">
         table   { text-align: right }
         body    { font-family: arial, sans-serif }
         div.red { color: red }
      </style>
      <script type = "text/javascript">
         <!--
         // variables used to test the state of the game
         var WON = 0;
         var LOST = 1;
         var CONTINUE_ROLLING = 2;

         // other variables used in program
         var firstRoll = true; // true if current roll is first
         var sumOfDice = 0; // sum of the dice
         var myPoint = 0; // point if no win/loss on first roll
         var gameStatus = CONTINUE_ROLLING; // game not over yet

         // process one roll of the dice
         function play()
         {
            // get the point field on the page
            var point = document.getElementById( "pointfield" );

            // get the status div on the page
            var statusDiv = document.getElementById( "status" );
            if ( firstRoll ) // first roll of the dice
            {
```

Fig. 9.6 | Craps game simulation. (Part 1 of 4.)

```
               sumOfDice = rollDice();

               switch ( sumOfDice )
               {
                  case 7: case 11: // win on first roll
                     gameStatus = WON;
                     // clear point field
                     point.value = "";
                     break;
                  case 2: case 3: case 12: // lose on first roll
                     gameStatus = LOST;
                     // clear point field
                     point.value = "";
                     break;
                  default:                   // remember point
                     gameStatus = CONTINUE_ROLLING;
                     myPoint = sumOfDice;
                     point.value = myPoint;
                     firstRoll = false;
               } // end switch
            } // end if
            else
            {
               sumOfDice = rollDice();

               if ( sumOfDice == myPoint ) // win by making point
                  gameStatus = WON;
               else
                  if ( sumOfDice == 7 )    // lose by rolling 7
                     gameStatus = LOST;
            } // end else

            if ( gameStatus == CONTINUE_ROLLING )
               statusDiv.innerHTML = "Roll again";
            else
            {
               if ( gameStatus == WON )
                  statusDiv.innerHTML = "Player wins. " +
                     "Click Roll Dice to play again.";
               else
                  statusDiv.innerHTML = "Player loses. " +
                     "Click Roll Dice to play again.";

               firstRoll = true;
            } // end else
         } // end function play

         // roll the dice
         function rollDice()
         {
            var die1;
            var die2;
            var workSum;
```

Fig. 9.6 | Craps game simulation. (Part 2 of 4.)

```

            die1 = Math.floor( 1 + Math.random() * 6 );
            die2 = Math.floor( 1 + Math.random() * 6 );
            workSum = die1 + die2;

            document.getElementById( "die1field" ).value = die1;
            document.getElementById( "die2field" ).value = die2;
            document.getElementById( "sumfield" ).value = workSum;

            return workSum;
         } // end function rollDice
         // -->
      </script>
   </head>
   <body>
      <form action = "">
         <table>
         <caption>Craps</caption>
         <tr><td>Die 1</td>
            <td><input id = "die1field" type = "text" />
            </td></tr>
         <tr><td>Die 2</td>
            <td><input id = "die2field" type = "text" />
            </td></tr>
         <tr><td>Sum</td>
            <td><input id = "sumfield" type = "text" />
            </td></tr>
         <tr><td>Point</td>
            <td><input id = "pointfield" type = "text" />
            </td></tr>
         <tr><td /><td><input type = "button" value = "Roll Dice"
           onclick = "play()" /></td></tr>
         </table>
         <div id = "status" class = "red">
            Click the Roll Dice button to play</div>
      </form>
   </body>
</html>
```


Program that Simulates the Game of Cr...
C:\books\2007\iw3htp
Google
Program that Simulates th...
My Computer
100%

Craps

Die 1 4

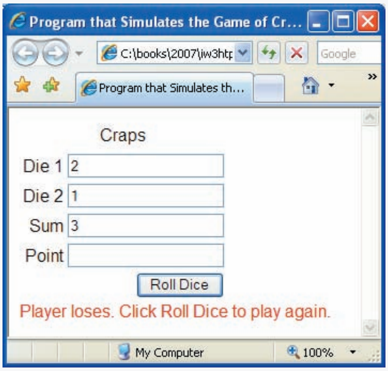
Die 2 3

Sum 7

Point

Roll Dice

Player wins. Click Roll Dice to play again.


Program that Simulates the Game of Cr...
C:\books\2007\iw3htp
Google
Program that Simulates th...
My Computer
100%

Craps

Die 1 2

Die 2 1

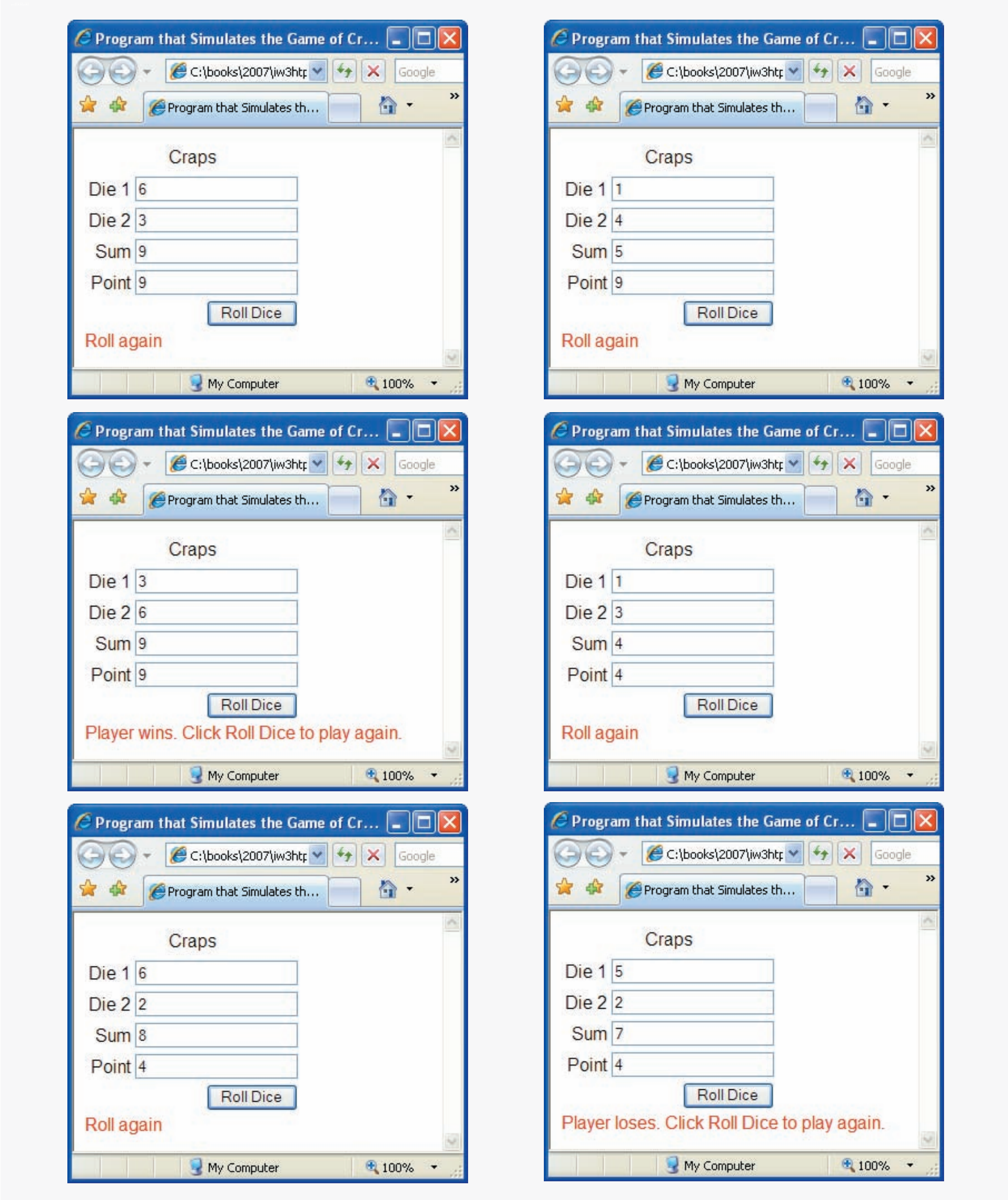
Sum 3

Point

Roll Dice

Player loses. Click Roll Dice to play again.

Fig. 9.6 | Craps game simulation. (Part 3 of 4.)



Program that Simulates the Game of Cr...
C:\books\2007\iw3htp
Google
Program that Simulates th...
My Computer
100%
Roll Dice

Craps

Die 1 6
Die 2 3
Sum 9
Point 9

Roll again

Craps

Die 1 1
Die 2 4
Sum 5
Point 9

Roll again

Craps

Die 1 3
Die 2 6
Sum 9
Point 9

Player wins. Click Roll Dice to play again.

Craps

Die 1 1
Die 2 3
Sum 4
Point 4

Roll again

Craps

Die 1 6
Die 2 2
Sum 8
Point 4

Roll again

Craps

Die 1 5
Die 2 2
Sum 7
Point 4

Player loses. Click Roll Dice to play again.

Fig. 9.6 | Craps game simulation. (Part 4 of 4.)

Before we discuss the script code, we discuss the `body` element (lines 105–126) of the XHTML document. The GUI components in this section are used extensively in the script.

Line 106 begins the definition of an XHTML `form` element. The XHTML standard requires that every `form` contain an `action` attribute, but because this form does not post its information to a web server, the empty string (`""`) is used.

In this example, we have decided to place the form's GUI components in an XHTML `table` element, so line 107 begins the definition of the XHTML table. Lines 109–120 create four table rows. Each row contains a left cell with a text label and an `input` element in the right cell.

Four `input` fields (lines 110, 113, 116 and 119) are created to display the value of the first die, the second die, the sum of the dice and the current point value, if any. Their `id` attributes are set to `die1field`, `die2field`, `sumfield`, and `pointfield`, respectively. The `id` attribute can be used to apply CSS styles and to enable script code to refer to an element in an XHTML document. Because the `id` attribute, if specified, must have a unique value, JavaScript can reliably refer to any single element via its `id` attribute. We see how this is done in a moment.

Lines 121–122 create a fifth row with an empty cell in the left column before the **Roll Dice** button. The button's **`onclick` attribute** indicates the action to take when the user of the XHTML document clicks the **Roll Dice** button. In this example, clicking the button causes a call to function `play`.

This style of programming is known as **event-driven programming**—the user interacts with a GUI component, the script is notified of the event and the script processes the event. The user's interaction with the GUI "drives" the program. The button click is known as the **event**. The function that is called when an event occurs is known as an **event-handling function** or **event handler**. When a GUI event occurs in a form, the browser calls the specified event-handling function. Before any event can be processed, each GUI component must know which event-handling function will be called when a particular event occurs. Most XHTML GUI components have several different event types. The event model is discussed in detail in Chapter 13, JavaScript: Events. By specifying `onclick = "play()"` for the **Roll Dice** button, we instruct the browser to **listen for events** (button-click events in particular). This **registers the event handler** for the GUI component, causing the browser to begin listening for the click event on the component. If no event handler is specified for the **Roll Dice** button, the script will not respond when the user presses the button.

Lines 123–125 end the `table` and `form` elements, respectively. After the table, a `div` element is created with an `id` attribute of `"status"`. This element will be updated by the script to display the result of each roll to the user. A style declaration in line 13 colors the text contained in this `div` red.

The game is reasonably involved. The player may win or lose on the first roll, or may win or lose on any subsequent roll. Lines 18–20 create variables that define the three game states—game won, game lost and continue rolling the dice. Unlike many other programming languages, JavaScript does not provide a mechanism to define a **constant** (i.e., a variable whose value cannot be modified). For this reason, we use all capital letters for these variable names, to indicate that we do not intend to modify their values and to make them stand out in the code—a common industry practice for genuine constants.

Good Programming Practice 9.4

Use only uppercase letters (with underscores between words) in the names of variables that should be used as constants. This format makes such variables stand out in a program.

Good Programming Practice 9.5

Use meaningfully named variables rather than literal values (such as 2) to make programs more readable.

Lines 23–26 declare several variables that are used throughout the script. Variable firstRoll indicates whether the next roll of the dice is the first roll in the current game. Variable sumOfDice maintains the sum of the dice from the last roll. Variable myPoint stores the point if the player does not win or lose on the first roll. Variable gameStatus keeps track of the current state of the game (WON, LOST or CONTINUE_ROLLING).

We define a function rollDice (lines 86–101) to roll the dice and to compute and display their sum. Function rollDice is defined once, but is called from two places in the program (lines 38 and 61). Function rollDice takes no arguments, so it has an empty parameter list. Function rollDice returns the sum of the two dice.

The user clicks the **Roll Dice** button to roll the dice. This action invokes function play (lines 29–83) of the script. Lines 32 and 35 create two new variables with objects representing elements in the XHTML document using the document object's **getElementById method**. The getElementById method, given an id as an argument, finds the XHTML element with a matching id attribute and returns a JavaScript object representing the element. Line 32 stores an object representing the pointfield input element (line 119) in the variable point. Line 35 gets an object representing the status div from line 124. In a moment, we show how you can use these objects to manipulate the XHTML document.

Function play checks the variable firstRoll (line 36) to determine whether it is true or false. If true, the roll is the first roll of the game. Line 38 calls rollDice, which picks two random values from 1 to 6, displays the value of the first die, the value of the second die and the sum of the dice in the first three text fields and returns the sum of the dice. (We discuss function rollDice in detail shortly.) After the first roll (if firstRoll is false), the nested switch statement in lines 40–57 determines whether the game is won or lost, or whether it should continue with another roll. After the first roll, if the game is not over, sumOfDice is saved in myPoint and displayed in the text field point in the XHTML form.

Note how the text field's value is changed in lines 45, 50 and 55. The object stored in the variable point allows access to the pointfield text field's contents. The expression point.value accesses the **value property** of the text field referred to by point. The value property specifies the text to display in the text field. To access this property, we specify the object representing the text field (point), followed by a **dot** (**.**) and the name of the property to access (value). This technique for accessing properties of an object (also used to access methods as in Math.pow) is called **dot notation**. We discuss using scripts to access elements in an XHTML page in more detail in Chapter 13.

The program proceeds to the nested if…else statement in lines 70–82, which uses the statusDiv variable to update the div that displays the game status. Using the object's **innerHTML property**, we set the text inside the div to reflect the most recent status. In lines 71, 75–76 and 78–79, we set the div's innerHTML to

```
Roll again.
```

if gameStatus is equal to CONTINUE_ROLLING, to

```
Player wins. Click Roll Dice to play again.
```

if gameStatus is equal to WON and to

```
Player loses. Click Roll Dice to play again.
```

if gameStatus is equal to LOST. If the game is won or lost, line 81 sets firstRoll to true to indicate that the next roll of the dice begins the next game.

The program then waits for the user to click the button **Roll Dice** again. Each time the user clicks **Roll Dice**, the program calls function `play`, which, in turn, calls the `rollDice` function to produce a new value for `sumOfDice`. If `sumOfDice` matches `myPoint`, `gameStatus` is set to `WON`, the `if…else` statement in lines 70–82 executes and the game is complete. If `sum` is equal to `7`, `gameStatus` is set to `LOST`, the `if…else` statement in lines 70–82 executes and the game is complete. Clicking the **Roll Dice** button starts a new game. The program updates the four text fields in the XHTML form with the new values of the dice and the sum on each roll, and updates the text field `point` each time a new game begins.

Function `rollDice` (lines 86–101) defines its own local variables `die1`, `die2` and `workSum` (lines 88–90). Because they are defined inside the `rollDice` function, these variables are accessible only inside that function. Lines 92–93 pick two random values in the range 1 to 6 and assign them to variables `die1` and `die2`, respectively. Lines 96–98 once again use the `document`'s `getElementById` method to find and update the correct `input` elements with the values of `die1`, `die2` and `workSum`. Note that the integer values are converted automatically to strings when they are assigned to each text field's `value` property. Line 100 returns the value of `workSum` for use in function `play`.

Software Engineering Observation 9.5

Variables that are declared inside the body of a function are known only in that function. If the same variable names are used elsewhere in the program, they will be entirely separate variables in memory.

Note the use of the various program-control mechanisms. The craps program uses two functions—`play` and `rollDice`—and the `switch`, `if…else` and nested `if` statements. Note also the use of multiple `case` labels in the `switch` statement to execute the same statements (lines 42 and 47). In the exercises at the end of this chapter, we investigate various interesting characteristics of the game of craps.

Error-Prevention Tip 9.2

Initializing variables when they are declared in functions helps avoid incorrect results and interpreter messages warning of uninitialized data.

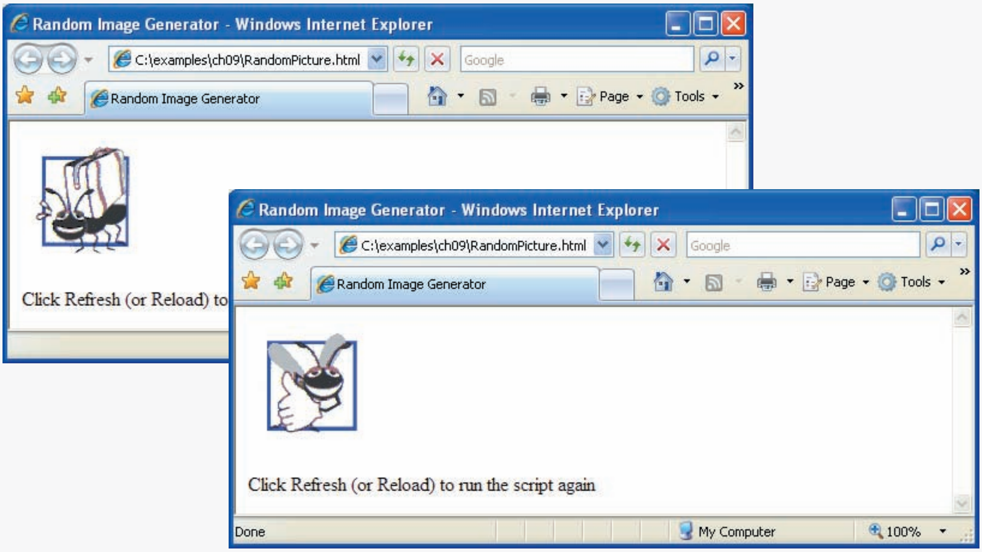
9.7 Another Example: Random Image Generator

Web content that varies randomly adds dynamic, interesting effects to a page. In the next example, we build a **random image generator**, a script that displays a randomly selected image every time the page that contains the script is loaded.

For the script in Fig. 9.7 to function properly, the directory containing the file `RandomPicture.html` must also contain seven images with integer filenames (i.e., `1.gif`, `2.gif`, …, `7.gif`). The web page containing this script displays one of these seven images, selected at random, each time the page loads.

Lines 12–13 randomly select an image to display on a web page. This `document.write` statement creates an image tag in the web page with the `src` attribute set to a random integer from 1 to 7, concatenated with `".gif"`. Thus, the script dynamically sets the source of the image tag to the name of one of the image files in the current directory.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 9.7: RandomPicture.html -->
<!-- Random image generation using Math.random. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Random Image Generator</title>
      <script type = "text/javascript">
         <!--
         document.write ( "<img src = \"" +
            Math.floor( 1 + Math.random() * 7 ) +  ".gif\" />" );
         // -->
      </script>
   </head>
   <body>
      <p>Click Refresh (or Reload) to run the script again</p>
   </body>
</html>
```



Random Image Generator - Windows Internet Explorer
C:\examples\ch09\RandomPicture.html
Google
Random Image Generator
Page
Tools
Done
My Computer
100%

Click Refresh (or Reload) to

Click Refresh (or Reload) to run the script again

Fig. 9.7 | Random image generation using `Math.random`.

9.8 Scope Rules

Chapters 6–8 used identifiers for variable names. The attributes of variables include name, value and data type (e.g., string, number or boolean). We also use identifiers as names for user-defined functions. Each identifier in a program also has a scope.

The scope of an identifier for a variable or function is the portion of the program in which the identifier can be referenced. Global variables or script-level variables that are declared in the `head` element are accessible in any part of a script and are said to have global scope. Thus every function in the script can potentially use the variables.

Identifiers declared inside a function have **function** (or **local**) **scope** and can be used only in that function. Function scope begins with the opening left brace ({) of the function in which the identifier is declared and ends at the terminating right brace (}) of the function. Local variables of a function and function parameters have function scope. If a local variable in a function has the same name as a global variable, the global variable is “hidden” from the body of the function.

Good Programming Practice 9.6

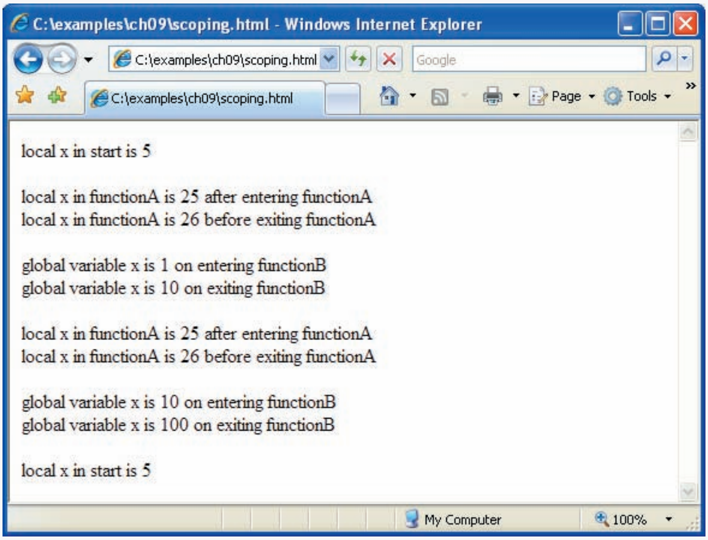
Avoid local-variable names that hide global-variable names. This can be accomplished by simply avoiding the use of duplicate identifiers in a script.

The script in Fig. 9.8 demonstrates the **scope rules** that resolve conflicts between global variables and local variables of the same name. This example also demonstrates the **`onload` event** (line 52), which calls an event handler (`start`) when the `<body>` of the XHTML document is completely loaded into the browser window.

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 9.8: scoping.html -->
 6  <!-- Scoping example. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>A Scoping Example</title>
10        <script type = "text/javascript">
11           <!--
12           var x = 1; // global variable
13
14           function start()
15           {
16              var x = 5; // variable local to function start
17
18              document.writeln( "local x in start is " + x );
19
20              functionA(); // functionA has local x
21              functionB(); // functionB uses global variable x
22              functionA(); // functionA reinitializes local x
23              functionB(); // global variable x retains its value
24
25              document.writeln(
26                 "<p>local x in start is " + x + "</p>" );
27           } // end function start
28
29           function functionA()
30           {
31              var x = 25; // initialized each time
32                          // functionA is called
33
34              document.writeln( "<p>local x in functionA is " +
35                                x + " after entering functionA" );
```

Fig. 9.8 | Scoping example. (Part 1 of 2.)

```
36                ++x;
37                document.writeln( "<br />local x in functionA is " +
38                   x + " before exiting functionA" + "</p>" );
39             } // end functionA
40
41             function functionB()
42             {
43                document.writeln( "<p>global variable x is " + x +
44                   " on entering functionB" );
45                x *= 10;
46                document.writeln( "<br />global variable x is " +
47                   x + " on exiting functionB"  + "</p>" );
48             } // end functionB
49             // -->
50          </script>
51       </head>
52       <body onload = "start()"></body>
53    </html>
```


C:\examples\ch09\scoping.html - Windows Internet Explorer
C:\examples\ch09\scoping.html
Google
C:\examples\ch09\scoping.html
Page
Tools
My Computer
100%

```
local x in start is 5

local x in functionA is 25 after entering functionA
local x in functionA is 26 before exiting functionA

global variable x is 1 on entering functionB
global variable x is 10 on exiting functionB

local x in functionA is 25 after entering functionA
local x in functionA is 26 before exiting functionA

global variable x is 10 on entering functionB
global variable x is 100 on exiting functionB

local x in start is 5
```

Fig. 9.8 | Scoping example. (Part 2 of 2.)

Global variable x (line 12) is declared and initialized to 1. This global variable is hidden in any block (or function) that declares a variable named x. Function start (line 14–27) declares a local variable x (line 16) and initializes it to 5. This variable is output in a line of XHTML text to show that the global variable x is hidden in start. The script defines two other functions—functionA and functionB—that each take no arguments and return nothing. Each function is called twice from function start.

Function functionA defines local variable x (line 31) and initializes it to 25. When functionA is called, the variable is output in a line of XHTML text to show that the global variable x is hidden in functionA; then the variable is incremented and output in a line of XHTML text again before the function is exited. Each time this function is called, local variable x is re-created and initialized to 25.

Function functionB does not declare any variables. Therefore, when it refers to variable x, the global variable x is used. When functionB is called, the global variable is output in a line of XHTML text, multiplied by 10 and output in a line of XHTML text again before the function is exited. The next time function functionB is called, the global variable has its modified value, 10, which again gets multiplied by 10, and 100 is output. Finally, the program outputs local variable x in start in a line of XHTML text again, to show that none of the function calls modified the value of x in start, because the functions all referred to variables in other scopes.

9.9 JavaScript Global Functions

JavaScript provides seven global functions. We have already used two of these functions—parseInt and parseFloat. The global functions are summarized in Fig. 9.9.

Actually, the global functions in Fig. 9.9 are all part of JavaScript's **Global object**. The Global object contains all the global variables in the script, all the user-defined functions in the script and all the functions listed in Fig. 9.9. Because global functions and user-defined functions are part of the Global object, some JavaScript programmers refer to these functions as methods. We use the term method only when referring to a function that is called for a particular object (e.g., Math.random()). As a JavaScript programmer, you do not need to use the Global object directly; JavaScript references it for you.

Global function	Description
escape	Takes a string argument and returns a string in which all spaces, punctuation, accent characters and any other character that is not in the ASCII character set (see Appendix D, ASCII Character Set) are encoded in a hexadecimal format (see Appendix E, Number Systems) that can be represented on all platforms.
eval	Takes a string argument representing JavaScript code to execute. The JavaScript interpreter evaluates the code and executes it when the eval function is called. This function allows JavaScript code to be stored as strings and executed dynamically. [*Note:* It is considered a serious security risk to use eval to process any data entered by a user because a malicious user could exploit this to run dangerous code.]
isFinite	Takes a numeric argument and returns true if the value of the argument is not NaN, Number.POSITIVE_INFINITY or Number.NEGATIVE_INFINITY (values that are not numbers or numbers outside the range that JavaScript supports)—otherwise, the function returns false.
isNaN	Takes a numeric argument and returns true if the value of the argument is not a number; otherwise, it returns false. The function is commonly used with the return value of parseInt or parseFloat to determine whether the result is a proper numeric value.

Fig. 9.9 | JavaScript global functions. (Part 1 of 2.)

Global function	Description
`parseFloat`	Takes a string argument and attempts to convert the beginning of the string into a floating-point value. If the conversion is unsuccessful, the function returns `NaN`; otherwise, it returns the converted value (e.g., `parseFloat( "abc123.45" )` returns `NaN`, and `parseFloat( "123.45abc" )` returns the value `123.45`).
`parseInt`	Takes a string argument and attempts to convert the beginning of the string into an integer value. If the conversion is unsuccessful, the function returns `NaN`; otherwise, it returns the converted value (e.g., `parseInt( "abc123" )` returns `NaN`, and `parseInt( "123abc" )` returns the integer value `123`). This function takes an optional second argument, from 2 to 36, specifying the **radix** (or **base**) of the number. Base 2 indicates that the first argument string is in **binary** format, base 8 indicates that the first argument string is in **octal** format and base 16 indicates that the first argument string is in **hexadecimal** format. See Appendix E, Number Systems, for more information on binary, octal and hexadecimal numbers.
`unescape`	Takes a string as its argument and returns a string in which all characters previously encoded with `escape` are decoded.

Fig. 9.9 | JavaScript global functions. (Part 2 of 2.)

9.10 Recursion

The programs we have discussed thus far are generally structured as functions that call one another in a disciplined, hierarchical manner. A **recursive function** is a function that calls *itself*, either directly, or indirectly through another function. **Recursion** is an important topic discussed at length in computer science courses. In this section, we present a simple example of recursion.

We consider recursion conceptually first; then we examine several programs containing recursive functions. Recursive problem-solving approaches have a number of elements in common. A recursive function is called to solve a problem. The function actually knows how to solve only the simplest case(s), or **base case(s)**. If the function is called with a base case, the function returns a result. If the function is called with a more complex problem, it divides the problem into two conceptual pieces—a piece that the function knows how to process (the base case) and a piece that the function does not know how to process. To make recursion feasible, the latter piece must resemble the original problem, but be a simpler or smaller version it. Because this new problem looks like the original problem, the function invokes (calls) a fresh copy of itself to go to work on the smaller problem; this invocation is referred to as a **recursive call**, or the **recursion step**. The recursion step also normally includes the keyword `return`, because its result will be combined with the portion of the problem the function knew how to solve to form a result that will be passed back to the original caller.

The recursion step executes while the original call to the function is still open (i.e., it has not finished executing). The recursion step can result in many more recursive calls as the function divides each new subproblem into two conceptual pieces. For the recursion eventually to terminate, each time the function calls itself with a simpler version of the original problem, the sequence of smaller and smaller problems must converge on the base case. At that point, the function recognizes the base case, returns a result to the previous copy of the function, and a sequence of returns ensues up the line until the original function call eventually returns the final result to the caller. This process sounds exotic when compared with the conventional problem solving we have performed to this point.

As an example of these concepts at work, let us write a recursive program to perform a popular mathematical calculation. The factorial of a nonnegative integer *n,* written *n*! (and pronounced "*n* factorial"), is the product

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

where 1! is equal to 1 and 0! is defined as 1. For example, 5! is the product 5 · 4 · 3 · 2 · 1, which is equal to 120.

The factorial of an integer (`number` in the following example) greater than or equal to zero can be calculated **iteratively** (nonrecursively) using a `for` statement, as follows:

```
var factorial = 1;

for ( var counter = number; counter >= 1; --counter )
   factorial *= counter;
```

A recursive definition of the factorial function is arrived at by observing the following relationship:

$$n! = n \cdot (n-1)!$$

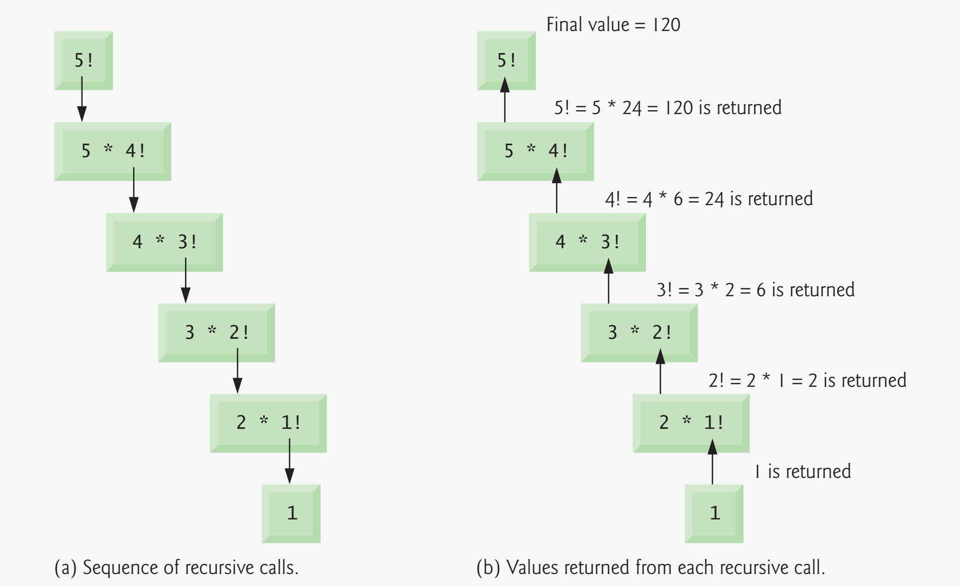
For example, *5!* is clearly equal to 5 * 4!, as is shown by the following equations:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

The evaluation of 5! would proceed as shown in Fig. 9.10. Figure 9.10 (a) shows how the succession of recursive calls proceeds until 1! is evaluated to be 1, which terminates the recursion. Figure 9.10 (b) shows the values returned from each recursive call to its caller until the final value is calculated and returned.

Figure 9.11 uses recursion to calculate and print the factorials of the integers 0 to 10. The recursive function `factorial` first tests (line 24) whether a terminating condition is `true`, i.e., whether `number` is less than or equal to 1. If so, `factorial` returns 1, no further recursion is necessary and the function returns. If `number` is greater than 1, line 27 expresses the problem as the product of `number` and the value returned by a recursive call to `factorial` evaluating the factorial of `number - 1`. Note that `factorial( number - 1 )` is a simpler problem than the original calculation, `factorial( number )`.

Function `factorial` (lines 22–28) receives as its argument the value for which to calculate the factorial. As can be seen in the screen capture in Fig. 9.11, factorial values become large quickly.


5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
Final value = 120
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1 is returned
1

(a) Sequence of recursive calls.

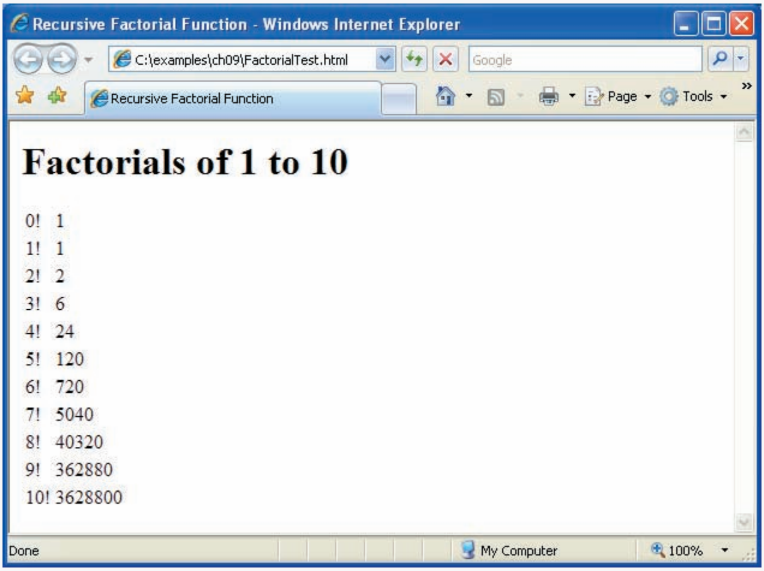
(b) Values returned from each recursive call.

Fig. 9.10 | Recursive evaluation of 5!.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 9.11: FactorialTest.html -->
<!-- Factorial calculation with a recursive function. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Recursive Factorial Function</title>
      <script type = "text/javascript">
      <!--
         document.writeln( "<h1>Factorials of 1 to 10</h1>" );
         document.writeln( "<table>" );

         for ( var i = 0; i <= 10; i++ )
            document.writeln( "<tr><td>" + i + "!</td><td>" +
               factorial( i ) + "</td></tr>" );

         document.writeln( "</table>" );

         // Recursive definition of function factorial
         function factorial( number )
         {
            if ( number <= 1 )  // base case
               return 1;
            else
               return number * factorial( number - 1 );
         } // end function factorial
```

Fig. 9.11 | Factorial calculation with a recursive function. (Part 1 of 2.)

```
         // -->
      </script>
   </head><body></body>
</html>
```


Recursive Factorial Function - Windows Internet Explorer
C:\examples\ch09\FactorialTest.html
Google
Recursive Factorial Function
Page
Tools
Done
My Computer
100%

Factorials of 1 to 10

```
0!  1
1!  1
2!  2
3!  6
4!  24
5!  120
6!  720
7!  5040
8!  40320
9!  362880
10! 3628800
```

Fig. 9.11 | Factorial calculation with a recursive function. (Part 2 of 2.)

Common Programming Error 9.6

Forgetting to return a value from a recursive function when one is needed results in a logic error.

Common Programming Error 9.7

Omitting the base case and writing the recursion step incorrectly so that it does not converge on the base case are both errors that cause infinite recursion, eventually exhausting memory. This situation is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.

Error-Prevention Tip 9.3

Internet Explorer displays an error message when a script seems to be going into infinite recursion. Firefox simply terminates the script after detecting the problem. This allows the user of the web page to recover from a script that contains an infinite loop or infinite recursion.

9.11 Recursion vs. Iteration

In the preceding section, we studied a function that can easily be implemented either recursively or iteratively. In this section, we compare the two approaches and discuss why you might choose one approach over the other in a particular situation.

Both iteration and recursion are based on a control statement: Iteration uses a repetition statement (e.g., `for`, `while` or `do…while`); recursion uses a selection statement (e.g., `if`, `if…else` or `switch`). Both iteration and recursion involve repetition: Iteration explicitly uses a repetition statement; recursion achieves repetition through repeated function calls. Iteration and recursion each involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized. Iteration both with counter-controlled repetition and with recursion gradually approaches termination: Iteration keeps modifying a counter until the counter assumes a value that makes the loop-continuation condition fail; recursion keeps producing simpler versions of the original problem until the base case is reached. Both iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem each time via a sequence that converges on the base case or if the base case is incorrect.

One negative aspect of recursion is that function calls require a certain amount of time and memory space not directly spent on executing program instructions. This is known as function-call overhead. Because recursion uses repeated function calls, this overhead greatly affects the performance of the operation. In many cases, using repetition statements in place of recursion is more efficient. However, some problems can be solved more elegantly (and more easily) with recursion.

Software Engineering Observation 9.6

Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution may not be apparent.

Performance Tip 9.1

Avoid using recursion in performance-oriented situations. Recursive calls take time and consume additional memory.

Common Programming Error 9.8

Accidentally having a nonrecursive function call itself, either directly, or indirectly through another function, can cause infinite recursion.

In addition to the Factorial function examnple (Fig. 9.11), we also provide several recursion exercises—raising an integer to an integer power (Exercise 9.34), visualizing recursion (Exercise 9.35) and sum of two integers (Exercise 9.36). Also, Fig. 14.26 uses recursion to traverse an XML document tree.

9.12 Wrap-Up

This chapter introduced JavaScript functions, which allow you to modularize your programs. We showed how to call functions and methods and how to define your own functions that accomplish tasks. We also showed how parameters are used to pass data into a function, while return values are used to pass a result back to the caller. We discussed how to get a range of random numbers and built a craps game and a random image generator using these concepts. Finally, we introduced recursion, which provides an alternative method for solving problems that involve repetitive calculations. In the next chapter, we introduce arrays, which allow you to store lists of data in a single variable.

9.13 Web Resources

`www.deitel.com/javascript/`

The Deitel JavaScript Resource Center contains links to some of the best JavaScript resources on the web. There you'll find categorized links to JavaScript tools, code generators, forums, books, libraries, frameworks and more. Also check out the tutorials for all skill levels, from introductory to advanced. Be sure to visit the related Resource Centers on XHTML (`www.deitel.com/xhtml/`) and CSS 2.1 (`www.deitel.com/css21/`).

Summary

Section 9.1 Introduction

- The best way to develop and maintain a large program is to construct it from small, simple pieces, or modules. This technique is called divide and conquer.

Section 9.2 Program Modules in JavaScript

- JavaScript programs are written by combining new functions that the programmer writes with "prepackaged" functions and objects available in JavaScript.
- The term method implies that the function belongs to a particular object. We refer to functions that belong to a particular JavaScript object as methods; all others are referred to as functions.
- JavaScript provides several objects that have a rich collection of methods for performing common mathematical calculations, string manipulations, date and time manipulations, and manipulations of collections of data called arrays. These objects make your job easier, because they provide many of the capabilities programmers frequently need.
- Whenever possible, use existing JavaScript objects, methods and functions instead of writing new ones. This reduces script-development time and helps avoid introducing errors.
- You can define functions that perform specific tasks and use them at many points in a script. These functions are referred to as programmer-defined functions. The actual statements defining the function are written only once and are hidden from other functions.
- Functions are invoked by writing the name of the function, followed by a left parenthesis, followed by a comma-separated list of zero or more arguments, followed by a right parenthesis.
- Methods are called in the same way as functions, but require the name of the object to which the method belongs and a dot preceding the method name.
- Function (and method) arguments may be constants, variables or expressions.

Section 9.3 Programmer-Defined Functions

- All variables declared in function definitions are local variables—this means that they can be accessed only in the function in which they are defined.
- A function's parameters are considered to be local variables. When a function is called, the arguments in the call are assigned to the corresponding parameters in the function definition.
- Code that is packaged as a function can be executed from several locations in a program by calling the function.
- Each function should perform a single, well-defined task, and the name of the function should express that task effectively. This promotes software reusability.

Section 9.4 Function Definitions

- The return statement passes information from inside a function back to the point in the program where it was called.

- A function must be called explicitly for the code in its body to execute.
- The format of a function definition is

```
function function-name( parameter-list )
{
    declarations and statements
}
```

- There are three ways to return control to the point at which a function was invoked. If the function does not return a result, control returns when the program reaches the function-ending right brace or by executing the statement `return;`. If the function does return a result, the statement `return` *expression*; returns the value of *expression* to the caller.

Section 9.5 Random Number Generation

- Method `random` generates a floating-point value from 0.0 up to, but not including, 1.0.
- Random integers in a certain range can be generated by scaling and shifting the values returned by `random`, then using Math.floor to convert them to integers. The scaling factor determines the size of the range (i.e. a scaling factor of 4 means four possible integers). The shift number is added to the result to determine where the range begins (i.e. shifting the numbers by 3 would give numbers between 3 and 7.)

Section 9.6 Example: Game of Chance

- JavaScript can execute actions in response to the user's interaction with a GUI component in an XHTML form. This is referred to as GUI event handling
- An XHTML element's `onclick` attribute indicates the action to take when the user of the XHTML document clicks on the element.
- In event-driven programming, the user interacts with a GUI component, the script is notified of the event and the script processes the event. The user's interaction with the GUI "drives" the program. The function that is called when an event occurs is known as an event-handling function or event handler.
- The `getElementById` method, given an `id` as an argument, finds the XHTML element with a matching `id` attribute and returns a JavaScript object representing the element.
- The `value` property of a JavaScript object representing an XHTML text input element specifies the text to display in the text field.
- Using an XHTML container (e.g. `div`, `span`, `p`) object's `innerHTML` property, we can use a script to set the contents of the element.

Section 9.7 Another Example: Random Image Generator

- We can use random number generation to randomly select from a number of images in order to display a random image each time a page loads.

Section 9.8 Scope Rules

- Each identifier in a program has a scope. The scope of an identifier for a variable or function is the portion of the program in which the identifier can be referenced.
- Global variables or script-level variables (i.e., variables declared in the head element of the XHTML document) are accessible in any part of a script and are said to have global scope. Thus every function in the script can potentially use the variables.
- Identifiers declared inside a function have function (or local) scope and can be used only in that function. Function scope begins with the opening left brace (`{`) of the function in which the iden-

tifier is declared and ends at the terminating right brace (}) of the function. Local variables of a function and function parameters have function scope.

- If a local variable in a function has the same name as a global variable, the global variable is "hidden" from the body of the function.
- The `onload` property of the body element calls an event handler when the `body` of the XHTML document is completely loaded into the browser window.

Section 9.9 JavaScript Global Functions

- JavaScript provides seven global functions as part of a `Global` object. This object contains all the global variables in the script, all the user-defined functions in the script and all the built-in global functions functions listed in Fig. 9.9.
- You do not need to use the `Global` object directly; JavaScript uses it for you.

Section 9.10 Recursion

- A recursive function calls itself, either directly, or indirectly through another function.
- A recursive function knows how to solve only the simplest case, or base case. If the function is called with a base case, it returns a result. If the function is called with a more complex problem, it knows how to divide the problem into two conceptual pieces—a piece that the function knows how to process (the base case) and a simpler or smaller version of the original problem.
- The function invokes (calls) a fresh copy of itself to go to work on the smaller problem; this invocation is referred to as a recursive call, or the recursion step.
- The recursion step executes while the original call to the function is still open (i.e., it has not finished executing).
- For recursion eventually to terminate, each time the function calls itself with a simpler version of the original problem, the sequence of smaller and smaller problems must converge on the base case. At that point, the function recognizes the base case, returns a result to the previous copy of the function, and a sequence of returns ensues up the line until the original function call eventually returns the final result to the caller.

Section 9.11 Recursion vs. Iteration

- Both iteration and recursion involve repetition: Iteration explicitly uses a repetition statement; recursion achieves repetition through repeated function calls.
- Iteration and recursion each involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized.
- Iteration with counter-controlled repetition and recursion both gradually approach termination: Iteration keeps modifying a counter until the counter assumes a value that makes the loop-continuation condition fail; recursion keeps producing simpler versions of the original problem until the base case is reached.
- Recursion repeatedly invokes the mechanism and, consequently, the overhead of function calls. This effect can be expensive in terms of processor time and memory space.
- Some problems can be understood or solved more easily with recursion than with iteration.

Terminology

argument in a function call
base case
binary format
block
called function
caller
calling function
computer-assisted instruction (CAI)

constant
converge on the base case
copy of a value
divide and conquer
dot (.)
dot notation
element of chance
escape function
eval function
element of chance
event
event handler
event-handling function
event-driven programming
floor method of the Math object
function
function (local) scope
function argument
function body
function call
function definition
function name
function parameter
function-call operator ()
getElementById method of the document object
Global object
global scope
global variable
hexadecimal
innerHTML property
invoke a function
isFinite function
isNaN function
iterative solution
listen for events
local scope
local variable
max method of the Math object
method
modularize a program
module
object
octal
onclick event
onload event
parameter in a function definition
parseFloat function
parseInt function
programmer-defined function
probability
programmer-defined function
radix
random method of the Math object
random-number generation
recursion
recursive function
recursive step
registering an event handler
respond to an event
return statement
scaling
scaling factor
scope
script-level variable
shifting value
simulation
software engineering
software reusability
unescape function
value property of an XHTML text field

Self-Review Exercises

9.1 Fill in the blanks in each of the following statements:

a) Program modules in JavaScript are called ________.
b) A function is invoked using a(n) ________.
c) A variable known only inside the function in which it is defined is called a(n) ________.
d) The ________ statement in a called function can be used to pass the value of an expression back to the calling function.
e) The keyword ________ indicates the beginning of a function definition.

9.2 For the given program, state the scope (either global scope or function scope) of each of the following elements:

a) The variable x.
b) The variable y.
c) The function cube.
d) The function output.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Exercise 9.2: cube.html -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Scoping</title>
      <script type = "text/javascript">
         <!--
         var x;

         function output()
         {
            for ( x = 1; x <= 10; x++ )
               document.writeln( cube( x ) + "<br />" );
         } // end function output

         function cube( y )
         {
            return y * y * y;
         } // end function cube
         // -->
   </script>
   </head><body onload = "output()"></body>
</html>
```

9.3 Fill in the blanks in each of the following statements:

a) Programmer-defined functions, global variables and JavaScript's global functions are all part of the ________ object.

b) Function ________ determines if its argument is or is not a number.

c) Function ________ takes a string argument and returns a string in which all spaces, punctuation, accent characters and any other character that is not in the ASCII character set are encoded in a hexadecimal format.

d) Function ________ takes a string argument representing JavaScript code to execute.

e) Function ________ takes a string as its argument and returns a string in which all characters that were previously encoded with `escape` are decoded.

9.4 Fill in the blanks in each of the following statements:

a) An identifier's ________ is the portion of the program in which the it can be used.

b) The three ways to return control from a called function to a caller are ________, ________ and ________.

c) The ________ function is used to produce random numbers.

d) Variables declared in a block or in a function's parameter list are of ________ scope.

9.5 Locate the error in each of the following program segments and explain how to correct it:

a)
```
method g()
{
   document.writeln( "Inside method g" );
}
```

b)
```
// This function should return the sum of its arguments
function sum( x, y )
{
   var result;
   result = x + y;
}
```

c)
```
function f( a );
{
   document.writeln( a );
}
```

9.6 Write a complete JavaScript program to prompt the user for the radius of a sphere, and call function `sphereVolume` to calculate and display the volume of the sphere. Use the statement

```
volume = ( 4.0 / 3.0 ) * Math.PI * Math.pow( radius, 3 );
```

to calculate the volume. The user should input the radius through an XHTML text field in a `<form>` and click an XHTML button to initiate the calculation.

Answers to Self-Review Exercises

9.1 a) functions. b) function call. c) local variable. d) `return`. e) `function`.

9.2 a) Global scope. b) Function scope. c) Global scope. d) Global scope.

9.3 a) `Global`. b) `isNaN`. c) `escape`. d) `eval`. e) `unescape`.

9.4 a) scope. b) `return`; or `return` *expression*; or encountering the closing right brace of a function. c) `Math.random`. d) local.

9.5 a) Error: `method` is not the keyword used to begin a function definition.
Correction: Change `method` to `function`.

b) Error: The function is supposed to return a value, but does not.
Correction: Either delete variable `result` and place the statement

```
return x + y;
```

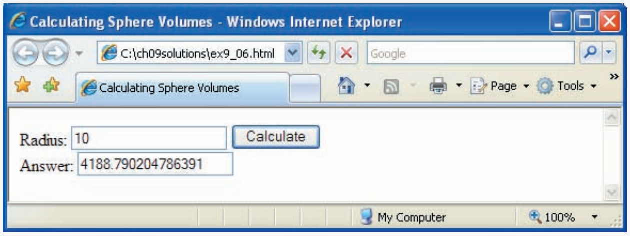
in the function or add the following statement at the end of the function body:

```
return result;
```

c) Error: The semicolon after the right parenthesis that encloses the parameter list.
Correction: Delete the semicolon after the right parenthesis of the parameter list.

9.6 The solution below calculates the volume of a sphere using the radius entered by the user.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Exercise 9.6: volume.html -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Calculating Sphere Volumes</title>
      <script type = "text/javascript">
         <!--
         function displayVolume()
         {
            var inputField = document.getElementById( "radiusField" );
            var radius = parseFloat( inputField.value );
            var answerField = document.getElementById( "answer" );
            answerField.value = sphereVolume( radius );
         } // end function displayVolume

         function sphereVolume( radius )
         {
            return ( 4.0 / 3.0 ) * Math.PI * Math.pow( radius, 3 );
         } // end function sphereVolume
         // -->
      </script>
   </head>
   <body>
      <form action = "">
         <div>
            <label>Radius:
               <input id = "radiusField" type = "text" /></label>
               <input type = "button" value = "Calculate"
                  onclick = "displayVolume()" />
            <br />
            <label>Answer:
               <input id = "answer" type = "text" /></label>
         </div>
      </form>
   </body>
</html>
```


Calculating Sphere Volumes - Windows Internet Explorer
C:\ch09solutions\ex9_06.html
Google
Calculating Sphere Volumes
Page
Tools
Calculate
My Computer
100%

Radius: 10

Answer: 4188.790204786391

Exercises

9.7 Write a script that prompts the user for the radius of a circle, uses a function `circleArea` to calculate the area of the circle, and prints the area of the circle.

9.8 A parking garage charges a $2.00 minimum fee to park for up to three hours. The garage charges an additional $0.50 per hour for each hour *or part thereof* in excess of three hours. The maximum charge for any given 24-hour period is $10.00. Assume that no car parks for longer than 24 hours at a time. Write a script that calculates and displays the parking charges for each customer who parked a car in this garage yesterday. You should input from the user the hours parked for each customer. The program should display the charge for the current customer and should calculate and display the running total of yesterday's receipts. The program should use the function `calculateCharges` to determine the charge for each customer. Use a text input field to obtain the input from the user.

9.9 Write function `distance` that calculates the distance between two points ($x1$, $y1$) and ($x2$, $y2$). All numbers and return values should be floating-point values. Incorporate this function into a script that enables the user to enter the coordinates of the points through an XHTML form.

9.10 Answer each of the following questions:

a) What does it mean to choose numbers “at random”?

b) Why is the `Math.random` function useful for simulating games of chance?

c) Why is it often necessary to scale and/or shift the values produced by `Math.random`?

d) Why is computerized simulation of real-world situations a useful technique?

9.11 Write statements that assign random integers to the variable n in the following ranges:

a) $1 \le n \le 2$

b) $1 \le n \le 100$

c) $0 \le n \le 9$

d) $1000 \le n \le 1112$

e) $-1 \le n \le 1$
f) $-3 \le n \le 11$

9.12 For each of the following sets of integers, write a single statement that will print a number at random from the set:

a) 2, 4, 6, 8, 10.
b) 3, 5, 7, 9, 11.
c) 6, 10, 14, 18, 22.

9.13 Write a function `integerPower( base, exponent )` that returns the value of

$$base^{exponent}$$

For example, `integerPower( 3, 4 ) = 3 * 3 * 3 * 3`. Assume that `exponent` and `base` are integers. Function `integerPower` should use a `for` or `while` statement to control the calculation. Incorporate this function into a script that reads integer values from an XHTML form for `base` and `exponent` and performs the calculation with the `integerPower` function. The XHTML form should consist of two text fields and a button to initiate the calculation. The user should interact with the program by typing numbers in both text fields then clicking the button.

9.14 Write a function `multiple` that determines, for a pair of integers, whether the second integer is a multiple of the first. The function should take two integer arguments and return `true` if the second is a multiple of the first, and `false` otherwise. Incorporate this function into a script that inputs a series of pairs of integers (one pair at a time). The XHTML form should consist of two text fields and a button to initiate the calculation. The user should interact with the program by typing numbers in both text fields, then clicking the button.

9.15 Write a script that inputs integers (one at a time) and passes them one at a time to function `isEven`, which uses the modulus operator to determine whether an integer is even. The function should take an integer argument and return `true` if the integer is even and `false` otherwise. Use sentinel-controlled looping and a `prompt` dialog.

9.16 Write a function `squareOfAsterisks` that displays a solid square of asterisks whose side is specified in integer parameter `side`. For example, if `side` is `4`, the function displays

```
****
****
****
****
```

Incorporate this function into a script that reads an integer value for `side` from the user at the keyboard and performs the drawing with the `squareOfAsterisks` function.

9.17 Modify the script created in Exercise 9.16 to also prompt the user for a character which will be used to create the square. Thus, if `side` is `5` and `fillCharacter` is `#`, the function should print

```
#####
#####
#####
#####
#####
```

9.18 Write program segments that accomplish each of the following tasks:

a) Calculate the integer part of the quotient when integer `a` is divided by integer `b`.
b) Calculate the integer remainder when integer `a` is divided by integer `b`.
c) Use the program pieces developed in parts (a) and (b) to write a function `displayDigits` that receives an integer between `1` and `99999` and prints it as a series of digits, each pair of which is separated by two spaces. For example, the integer `4562` should be printed as

```
4  5  6  2
```

d) Incorporate the function developed in part (c) into a script that inputs an integer from a `prompt` dialog and invokes `displayDigits` by passing to the function the integer entered.

9.19 Implement the following functions:

a) Function `celsius` returns the Celsius equivalent of a Fahrenheit temperature, using the calculation

```
C = 5.0 / 9.0 * ( F - 32 );
```

b) Function `fahrenheit` returns the Fahrenheit equivalent of a Celsius temperature, using the calculation

```
F = 9.0 / 5.0 * C + 32;
```

c) Use these functions to write a script that enables the user to enter either a Fahrenheit or a Celsius temperature and displays the Celsius or Fahrenheit equivalent.

Your XHTML document should contain two buttons—one to initiate the conversion from Fahrenheit to Celsius and one to initiate the conversion from Celsius to Fahrenheit.

9.20 Write a function `minimum3` that returns the smallest of three floating-point numbers. Use the `Math.min` function to implement `minimum3`. Incorporate the function into a script that reads three values from the user and determines the smallest value.

9.21 An integer number is said to be a **perfect number** if its factors, including 1 (but not the number itself), sum to the number. For example, 6 is a perfect number, because 6 = 1 + 2 + 3. Write a function `perfect` that determines whether parameter `number` is a perfect number. Use this function in a script that determines and displays all the perfect numbers between 1 and 1000. Print the factors of each perfect number to confirm that the number is indeed perfect. Challenge the computing power of your computer by testing numbers much larger than 1000. Display the results in a `<textarea>`.

9.22 An integer is said to be **prime** if it is greater than 1 and divisible only by 1 and itself. For example, 2, 3, 5 and 7 are prime, but 4, 6, 8 and 9 are not.

a) Write a function that determines whether a number is prime.

b) Use this function in a script that determines and prints all the prime numbers between 1 and 10,000. How many of these 10,000 numbers do you really have to test before being sure that you have found all the primes? Display the results in a `<textarea>`.

c) Initially, you might think that *n*/2 is the upper limit for which you must test to see whether a number is prime, but you only need go as high as the square root of *n*. Why? Rewrite the program using the `Math.sqrt` method to calculate the square root, and run it both ways. Estimate the performance improvement.

9.23 Write a function that takes an integer value and returns the number with its digits reversed. For example, given the number 7631, the function should return 1367. Incorporate the function into a script that reads a value from the user. Display the result of the function in the status bar.

9.24 The **greatest common divisor** (GCD) of two integers is the largest integer that evenly divides each of the two numbers. Write a function `gcd` that returns the greatest common divisor of two integers. Incorporate the function into a script that reads two values from the user.

9.25 Write a function `qualityPoints` that inputs a student’s average and returns 4 if the student's average is 90–100, 3 if the average is 80–89, 2 if the average is 70–79, 1 if the average is 60–69 and 0 if the average is lower than 60. Incorporate the function into a script that reads a value from the user.

9.26 Write a script that simulates coin tossing. Let the program toss the coin each time the user clicks the **Toss** button. Count the number of times each side of the coin appears. Display the results.

The program should call a separate function `flip` that takes no arguments and returns `false` for tails and `true` for heads. [*Note:* If the program realistically simulates the coin tossing, each side of the coin should appear approximately half the time.]

9.27 Computers are playing an increasing role in education. Write a program that will help an elementary-school student learn multiplication. Use `Math.random` to produce two positive one-digit integers. It should then display a question such as

```
How much is 6 times 7?
```

The student then types the answer into a text field. Your program checks the student's answer. If it is correct, display the string `"Very good!"` and generate a new question. If the answer is wrong, display the string `"No. Please try again."` and let the student try the same question again repeatedly until the student finally gets it right. A separate function should be used to generate each new question. This function should be called once when the script begins execution and each time the user answers the question correctly.

9.28 The use of computers in education is referred to as computer-assisted instruction (CAI). One problem that develops in CAI environments is student fatigue. This problem can be eliminated by varying the computer's dialogue to hold the student's attention. Modify the program in Exercise 9.27 to print one of a variety of comments for each correct answer and each incorrect answer. The set of responses for correct answers is as follows:

```
Very good!
Excellent!
Nice work!
Keep up the good work!
```

The set of responses for incorrect answers is as follows:

```
No. Please try again.
Wrong. Try once more.
Don't give up!
No. Keep trying.
```

Use random number generation to choose a number from 1 to 4 that will be used to select an appropriate response to each answer. Use a `switch` statement to issue the responses.

9.29 More sophisticated computer-assisted instruction systems monitor the student's performance over a period of time. The decision to begin a new topic is often based on the student's success with previous topics. Modify the program in Exercise 9.28 to count the number of correct and incorrect responses typed by the student. After the student answers 10 questions, your program should calculate the percentage of correct responses. If the percentage is lower than 75 percent, print `Please ask your instructor for extra help`, and reset the program so another student can try it.

9.30 Write a script that plays a "guess the number" game as follows: Your program chooses the number to be guessed by selecting a random integer in the range 1 to 1000. The script displays the prompt `Guess a number between 1 and 1000` next to a text field. The player types a first guess into the text field and clicks a button to submit the guess to the script. If the player's guess is incorrect, your program should display `Too high. Try again.` or `Too low. Try again.` to help the player "zero in" on the correct answer and should clear the text field so the user can enter the next guess. When the user enters the correct answer, display `Congratulations. You guessed the number!` and clear the text field so the user can play again. [*Note:* The guessing technique employed in this problem is similar to a binary search, which we discuss in Chapter 10, JavaScript: Arrays.]

9.31 Modify the program of Exercise 9.30 to count the number of guesses the player makes. If the number is 10 or fewer, display `Either you know the secret or you got lucky!` If the player guesses the number in 10 tries, display `Ahah! You know the secret!` If the player makes more than 10 guesses, display `You should be able to do better!` Why should it take no more than 10 guesses? Well, with each good guess, the player should be able to eliminate half of the numbers. Now show why any number 1 to 1000 can be guessed in 10 or fewer tries.

9.32 Exercises 9.27 through 9.29 developed a computer-assisted instruction program to teach an elementary-school student multiplication. This exercise suggests enhancements to that program.

a) Modify the program to allow the user to enter a grade-level capability. A grade level of 1 means to use only single-digit numbers in the problems, a grade level of 2 means to use numbers as large as two digits, and so on.

b) Modify the program to allow the user to pick the type of arithmetic problems he or she wishes to study. An option of 1 means addition problems only, 2 means subtraction problems only, 3 means multiplication problems only, 4 means division problems only and 5 means to intermix randomly problems of all these types.

9.33 Modify the craps program in Fig. 9.6 to allow wagering. Initialize variable `bankBalance` to 1000 dollars. Prompt the player to enter a `wager`. Check that the `wager` is less than or equal to `bankBalance` and, if not, have the user reenter `wager` until a valid `wager` is entered. After a valid `wager` is entered, run one game of craps. If the player wins, increase `bankBalance` by `wager`, and print the new `bankBalance`. If the player loses, decrease `bankBalance` by `wager`, print the new `bankBalance`, check whether `bankBalance` has become zero and, if so, print the message `Sorry. You busted!` As the game progresses, print various messages to create some chatter, such as `Oh, you're going for broke, huh?` or `Aw c'mon, take a chance!` or `You're up big. Now's the time to cash in your chips!`. Implement the chatter as a separate function that randomly chooses the string to display.

9.34 Write a recursive function `power( base, exponent )` that, when invoked, returns

$$base^{exponent}$$

for example, `power( 3, 4 ) = 3 * 3 * 3 * 3`. Assume that `exponent` is an integer greater than or equal to 1. The recursion step would use the relationship

$$base^{exponent} = base \cdot base^{exponent-1}$$

and the terminating condition occurs when `exponent` is equal to `1`, because

$$base^{1} = base$$

Incorporate this function into a script that enables the user to enter the `base` and `exponent`.

9.35 *(Visualizing Recursion)* It is interesting to watch recursion in action. Modify the factorial function in Fig. 9.11 to display its local variable and recursive-call parameter. For each recursive call, display the outputs on a separate line and add a level of indentation. Do your utmost to make the outputs clear, interesting and meaningful. Your goal here is to design and implement an output format that helps a person understand recursion better. You may want to add such display capabilities to the many other recursion examples and exercises throughout the text.

9.36 What does the following function do?

```
// Parameter b must be a positive
// integer to prevent infinite recursion
function mystery( a, b )
{
   if ( b == 1 )
      return a;
   else
      return a + mystery( a, b - 1 );
}
```

10

JavaScript: Arrays

With sobs and tears he sorted out
Those of the largest size . . .
—Lewis Carroll

Attempt the end, and never stand to doubt;
Nothing's so hard, but search will find it out.
—Robert Herrick

Now go, write it before them in a table,
and note it in a book.
—Isaiah 30:8

'Tis in my memory lock'd,
And you yourself shall keep the key of it.
—William Shakespeare

OBJECTIVES

In this chapter you will learn:

- To use arrays to store lists and tables of values.
- To declare an array, initialize an array and refer to individual elements of an array.
- To pass arrays to functions.
- To search and sort an array.
- To declare and manipulate multidimensional arrays.

Outline

10.1 Introduction

This chapter serves as an introduction to the important topic of data structures. **Arrays** are data structures consisting of related data items (sometimes called **collections** of data items). JavaScript arrays are “dynamic” entities in that they can change size after they are created. Many of the techniques demonstrated in this chapter are used frequently in Chapters 12–13 as we introduce the collections that allow a script programmer to manipulate every element of an XHTML document dynamically.

10.2 Arrays

An array is a group of memory locations that all have the same name and normally are of the same type (although this attribute is not required in JavaScript). To refer to a particular location or element in the array, we specify the name of the array and the **position number** of the particular element in the array.

Figure 10.1 shows an array of integer values named c. This array contains 12 **elements**. Any one of these elements may be referred to by giving the name of the array followed by the position number of the element in square brackets ([]). The first element in every array is the **zeroth element**. Thus, the first element of array c is referred to as c[0], the second element of array c is referred to as c[1], the seventh element of array c is referred to as c[6] and, in general, the *i*th element of array c is referred to as c[i-1]. Array names follow the same conventions as other identifiers.

The position number in square brackets is called a **subscript** (or an **index**). A subscript must be an integer or an integer expression. If a program uses an expression as a subscript, then the expression is evaluated to determine the value of the subscript. For example, if we assume that variable a is equal to 5 and that variable b is equal to 6, then the statement

```
c[ a + b ] += 2;
```

adds 2 to array element c[11]. Note that a subscripted array name is a left-hand-side expression—it can be used on the left side of an assignment to place a new value into an array

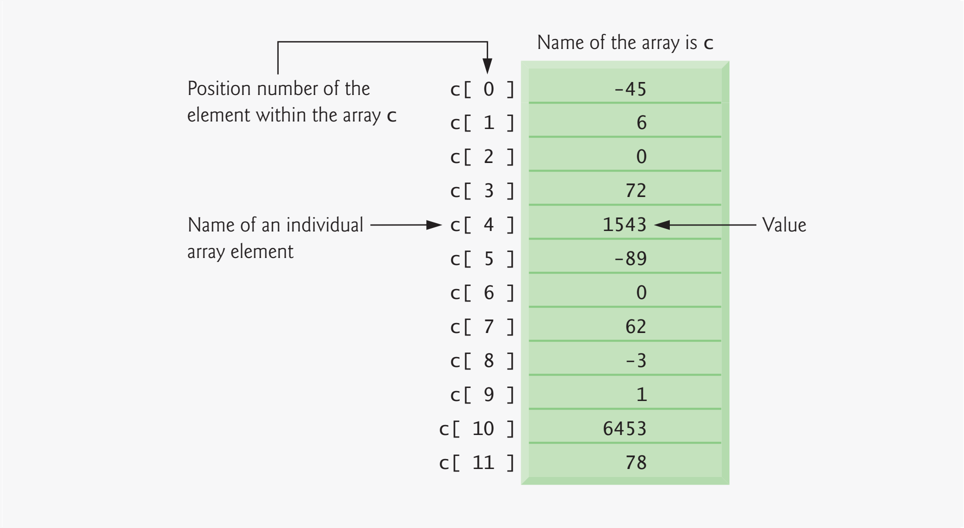


Fig. 10.1 | Array with 12 elements.

element. It can also be used on the right side of an assignment to assign its value to another left-hand side expression.

Let us examine array c in Fig. 10.1 more closely. The array's **name** is c. The **length** of array c is 12 and can be found using by the following expression:

```
c.length
```

Every array in JavaScript knows its own length. The array's 12 elements are referred to as c[0], c[1], c[2], …, c[11]. The **value** of c[0] is -45, the value of c[1] is 6, the value of c[2] is 0, the value of c[7] is 62 and the value of c[11] is 78. To calculate the sum of the values contained in the first three elements of array c and store the result in variable sum, we would write

```
sum = c[ 0 ] + c[ 1 ] + c[ 2 ];
```

To divide the value of the seventh element of array c by 2 and assign the result to the variable x, we would write

```
x = c[ 6 ] / 2;
```

Common Programming Error 10.1

It is important to note the difference between the "seventh element of the array" and "array element seven." Because array subscripts begin at 0, the seventh element of the array has a subscript of 6, while array element seven has a subscript of 7 and is actually the eighth element of the array. This confusion is a source of "off-by-one" errors.

The brackets that enclose the array subscript are a JavaScript operator. Brackets have the same level of precedence as parentheses. The chart in Fig. 10.2 shows the precedence and associativity of the operators introduced so far in the text. They are shown from top to bottom in decreasing order of precedence, alongside their associativity and type.

Operators	Associativity	Type
() [] .	left to right	highest
++ -- !	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment

Fig. 10.2 | Precedence and associativity of the operators discussed so far.

10.3 Declaring and Allocating Arrays

Arrays occupy space in memory. Actually, an array in JavaScript is an **`Array` object**. The programmer uses **operator `new`** to allocate dynamically (request memory for) the number of elements required by each array. Operator `new` creates an object as the program executes by obtaining enough memory to store an object of the type specified to the right of `new`. The process of creating new objects is also known as **creating an instance** or **instantiating an object**, and operator `new` is known as the **dynamic memory allocation operator**. Arrays are allocated with `new` because arrays are considered to be objects, and all objects must be created with `new`. To allocate 12 elements for integer array `c`, use the statement

```
var c = new Array( 12 );
```

The preceding statement can also be performed in two steps, as follows:

```
var c; // declares the array
c = new Array( 12 ); // allocates the array
```

When arrays are allocated, the elements are not initialized—they have the value `undefined`.

Common Programming Error 10.2

Assuming that the elements of an array are initialized when the array is allocated may result in logic errors.

10.4 Examples Using Arrays

This section presents several examples of creating and manipulating arrays.

Creating and Initializing Arrays

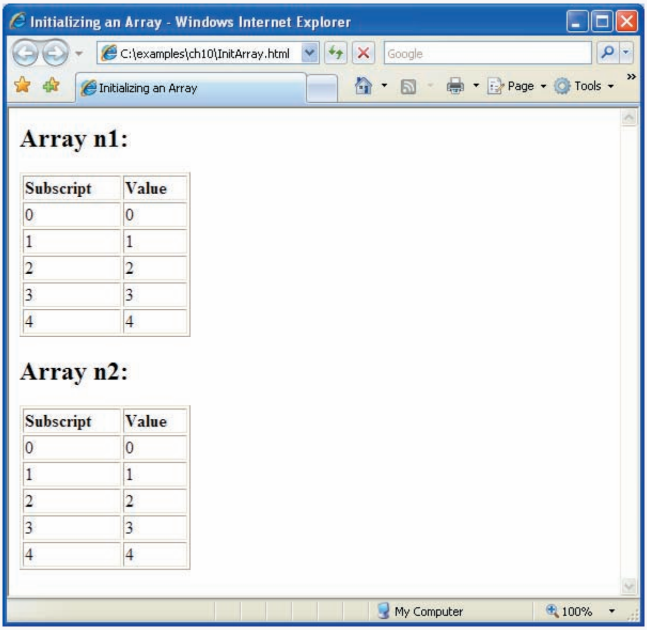
The script in Fig. 10.3 uses operator `new` to allocate an `Array` of five elements and an empty array. The script demonstrates initializing an `Array` of existing elements and also shows that an Array can grow dynamically to accommodate new elements. The Array's values are displayed in XHTML tables.

Line 17 creates Array n1 as an array of five elements. Line 18 creates Array n2 as an empty array. Lines 21–22 use a for statement to initialize the elements of n1 to their sub-

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 10.3: InitArray.html -->
 6  <!-- Initializing the elements of an array. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Initializing an Array</title>
10        <style type = "text/css">
11           table { width: 10em }
12           th    { text-align: left }
13        </style>
14        <script type = "text/javascript">
15           <!--
16           // create (declare) two new arrays
17           var n1 = new Array( 5 ); // allocate five-element Array
18           var n2 = new Array(); // allocate empty Array
19
20           // assign values to each element of Array n1
21           for ( var i = 0; i < n1.length; ++i )
22              n1[ i ] = i;
23
24           // create and initialize five elements in Array n2
25           for ( i = 0; i < 5; ++i )
26              n2[ i ] = i;
27
28           outputArray( "Array n1:", n1 );
29           outputArray( "Array n2:", n2 );
30
31           // output the heading followed by a two-column table
32           // containing subscripts and elements of "theArray"
33           function outputArray( heading, theArray )
34           {
35              document.writeln( "<h2>" + heading + "</h2>" );
36              document.writeln( "<table border = \"1\"" );
37              document.writeln( "<thead><th>Subscript</th>" +
38                 "<th>Value</th></thead><tbody>" );
39
40              // output the subscript and value of each array element
41              for ( var i = 0; i < theArray.length; i++ )
42                 document.writeln( "<tr><td>" + i + "</td><td>" +
43                    theArray[ i ] + "</td></tr>" );
44
45              document.writeln( "</tbody></table>" );
46           } // end function outputArray
47           // -->
```

Fig. 10.3 | Initializing the elements of an array. (Part 1 of 2.)

```
48          </script>
49       </head><body></body>
50    </html>
```


Initializing an Array - Windows Internet Explorer
C:\examples\ch10\InitArray.html
Google
Initializing an Array
Page
Tools
My Computer
100%

Array n1:

Subscript	Value
0	0
1	1
2	2
3	3
4	4

Array n2:

Subscript	Value
0	0
1	1
2	2
3	3
4	4

Fig. 10.3 | Initializing the elements of an array. (Part 2 of 2.)

script numbers (0 to 4). Note also the use of zero-based counting (remember, array subscripts start at 0) so that the loop can access every element of the array. Note too the use of the expression `n1.length` in the condition for the `for` statement to determine the length of the array. In this example, the length of the array is 5, so the loop continues executing as long as the value of control variable `i` is less than `5`. For a five-element array, the subscript values are 0 through 4, so using the less than operator, <, guarantees that the loop does not attempt to access an element beyond the end of the array. Zero-based counting is usually used to iterate through arrays.

Lines 25–26 use a `for` statement to add five elements to the `Array n2` and initialize each element to its subscript number (0 to 4). Note that `Array n2` grows dynamically to accommodate the values assigned to each element of the array.

Software Engineering Observation 10.1

JavaScript automatically reallocates an `Array` when a value is assigned to an element that is outside the bounds of the original `Array`. Elements between the last element of the original `Array` and the new element have undefined values.

Lines 28–29 invoke function `outputArray` (defined in lines 33–46) to display the contents of each array in an XHTML table. Function `outputArray` receives two arguments—a string to be output before the XHTML table that displays the contents of the array and the array to output. Lines 41–43 use a `for` statement to output XHTML text that defines each row of the table. Once again, note the use of zero-based counting so that the loop can access every element of the array.

Common Programming Error 10.3

Referring to an element outside the `Array` bounds is normally a logic error.

Error-Prevention Tip 10.1

When using subscripts to loop through an `Array`, the subscript should never go below 0 and should always be less than the number of elements in the `Array` (i.e., one less than the size of the `Array`). Make sure that the loop-terminating condition prevents the access of elements outside this range.

If the values of an `Array`'s elements are known in advance, the elements can be allocated and initialized in the declaration of the array. There are two ways in which the initial values can be specified. The statement

```
var n = [ 10, 20, 30, 40, 50 ];
```

uses a comma-separated **initializer list** enclosed in square brackets (`[` and `]`) to create a five-element `Array` with subscripts of `0`, `1`, `2`, `3` and `4`. The array size is determined by the number of values in the initializer list. Note that the preceding declaration does not require the `new` operator to create the `Array` object—this functionality is provided by the interpreter when it encounters an array declaration that includes an initializer list. The statement

```
var n = new Array( 10, 20, 30, 40, 50 );
```

also creates a five-element array with subscripts of `0`, `1`, `2`, `3` and `4`. In this case, the initial values of the array elements are specified as arguments in the parentheses following `new Array`. The size of the array is determined by the number of values in parentheses. It is also possible to reserve a space in an `Array` for a value to be specified later by using a comma as a **place holder** in the initializer list. For example, the statement

```
var n = [ 10, 20, , 40, 50 ];
```

creates a five-element array with no value specified for the third element (`n[ 2 ]`).

Initializing Arrays with Initializer Lists

The script in Fig. 10.4 creates three `Array` objects to demonstrate initializing arrays with initializer lists (lines 18–20) and displays each array in an XHTML table using the same function `outputArray` discussed in Fig. 10.3. Note that when `Array integers2` is displayed in the web page, the elements with subscripts 1 and 2 (the second and third elements of the array) appear in the web page as `undefined`. These are the two elements for which we did not supply values in the declaration in line 20 in the script.

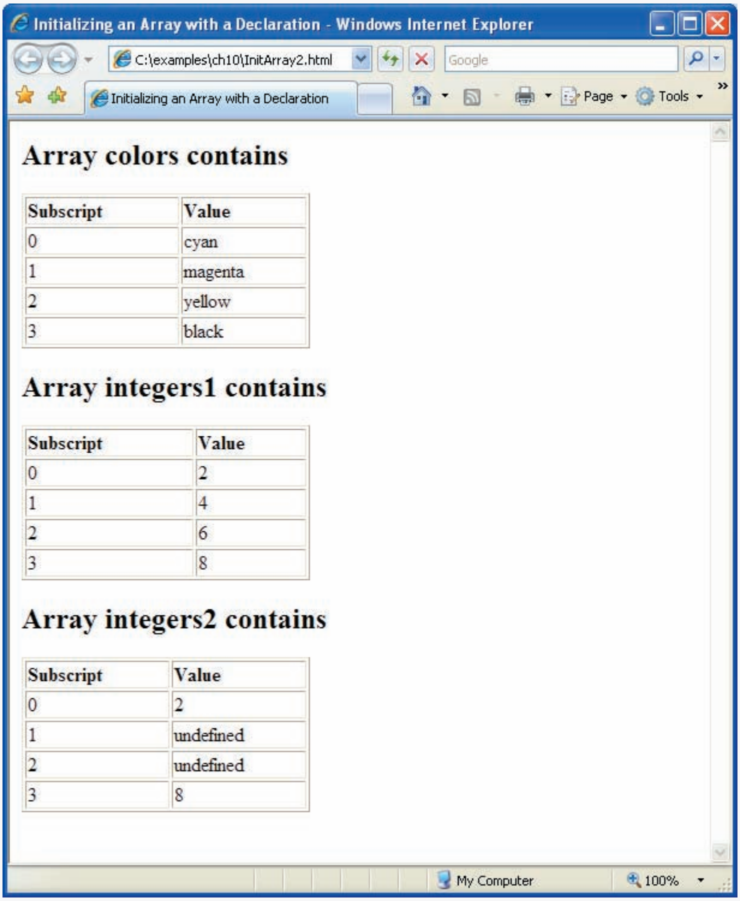
Summing the Elements of an Array with `for` and `for…in`

The script in Fig. 10.5 sums the values contained in `theArray`, the 10-element integer array declared, allocated and initialized in line 13. The statement in line 19 in the body of the first `for` statement does the totaling. Note that the values supplied as initializers for array `theArray` could be read into the program using an XHTML form.

In this example, we introduce JavaScript's **for…in statement**, which enables a script to perform a task for each element in an array (or, as we will see in Chapters 12–13, for each element in a collection). This process is also known as **iterating over the elements of an array**. Lines 25-26 show the syntax of a `for…in` statement. Inside the parentheses, we declare the `element` variable used to select each element in the object to the right of key-

```
1  <?xml version = "1.0" encoding = "utf-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 10.4: InitArray2.html -->
6  <!-- Declaring and initializing arrays. -->
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8     <head>
9        <title>Initializing an Array with a Declaration</title>
10       <style type = "text/css">
11          table { width: 15em }
12          th    { text-align: left }
13       </style>
14       <script type = "text/javascript">
15          <!--
16          // Initializer list specifies the number of elements and
17          // a value for each element.
18          var colors = new Array( "cyan", "magenta","yellow", "black" );
19          var integers1 = [ 2, 4, 6, 8 ];
20          var integers2 = [ 2, , , 8 ];
21
22          outputArray( "Array colors contains", colors );
23          outputArray( "Array integers1 contains", integers1 );
24          outputArray( "Array integers2 contains", integers2 );
25
26          // output the heading followed by a two-column table
27          // containing the subscripts and elements of theArray
28          function outputArray( heading, theArray )
29          {
30             document.writeln( "<h2>" + heading + "</h2>" );
31             document.writeln( "<table border = \"1\"" );
32             document.writeln( "<thead><th>Subscript</th>" +
33                "<th>Value</th></thead><tbody>" );
34
35             // output the subscript and value of each array element
36             for ( var i = 0; i < theArray.length; i++ )
37                document.writeln( "<tr><td>" + i + "</td><td>" +
38                   theArray[ i ] + "</td></tr>" );
39
40             document.writeln( "</tbody></table>" );
41          } // end function outputArray
42          // -->
43       </script>
44    </head><body></body>
45 </html>
```

Fig. 10.4 | Declaring and initializing arrays. (Part 1 of 2.)


Initializing an Array with a Declaration - Windows Internet Explorer
C:\examples\ch10\InitArray2.html
Google
Initializing an Array with a Declaration
Page
Tools
My Computer
100%

Array colors contains

Subscript	Value
0	cyan
1	magenta
2	yellow
3	black

Array integers1 contains

Subscript	Value
0	2
1	4
2	6
3	8

Array integers2 contains

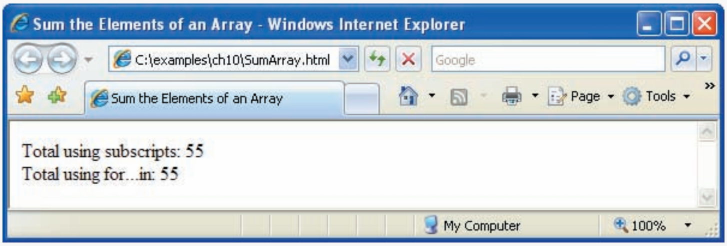
Subscript	Value
0	2
1	undefined
2	undefined
3	8

Fig. 10.4 | Declaring and initializing arrays. (Part 2 of 2.)

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 10.5: SumArray.html -->
<!-- Summing elements of an array. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Sum the Elements of an Array</title>

      <script type = "text/javascript">
         <!--
         var theArray = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ];
         var total1 = 0, total2 = 0;

```

Fig. 10.5 | Summing elements of an array. (Part 1 of 2.)

```
         // iterates through the elements of the array in order and adds
         // each element's value to total1
         for ( var i = 0; i < theArray.length; i++ )
            total1 += theArray[ i ];

         document.writeln( "Total using subscripts: " + total1 );

         // iterates through the elements of the array using a for... in
         // statement to add each element's value to total2
         for ( var element in theArray )
            total2 += theArray[ element ];

         document.writeln( "<br />Total using for...in: " + total2 );
         // -->
      </script>
   </head><body></body>
</html>
```


Sum the Elements of an Array - Windows Internet Explorer
C:\examples\ch10\SumArray.html
Google
Sum the Elements of an Array
Page
Tools
My Computer
100%

```
Total using subscripts: 55
Total using for...in: 55
```

Fig. 10.5 | Summing elements of an array. (Part 2 of 2.)

word `in` (`theArray` in this case). When using `for…in`, JavaScript automatically determines the number of elements in the array. As the JavaScript interpreter iterates over `theArray`'s elements, variable `element` is assigned a value that can be used as a subscript for `theArray`. In the case of an `Array`, the value assigned is a subscript in the range from `0` up to, but not including, `theArray.length`. Each value is added to `total2` to produce the sum of the elements in the array.

Error-Prevention Tip 10.2

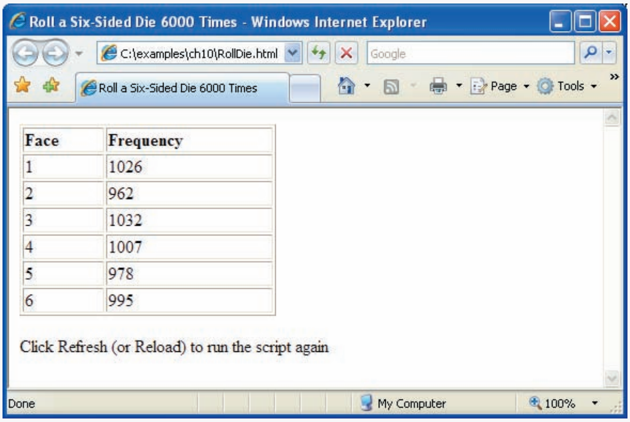
When iterating over all the elements of an `Array`, use a `for…in` statement to ensure that you manipulate only the existing elements of the `Array`. Note that a `for…in` statement skips any undefined elements in the array.

Using the Elements of an Array as Counters

In Chapter 9, we indicated that there is a more elegant way to implement the dice-rolling program in Fig. 9.5. The program rolled a single six-sided die 6000 times and used a `switch` statement to total the number of times each value was rolled. An array version of this script is shown in Fig. 10.6. The `switch` statement in Fig. 9.5 is replaced by line 24 of this program. This line uses the random `face` value as the subscript for the array `frequency` to determine which element to increment during each iteration of the loop. Because the random number calculation in line 23 produces numbers from 1 to 6 (the values for a six-sided die), the `frequency` array must be large enough to allow subscript values of 1 to 6. The smallest number of elements required for an array to have these subscript values is seven elements (subscript values from 0 to 6). In this program, we ignore element 0 of array `frequency` and use only the elements that correspond to values on the sides of a die. Also, lines 32–34 of this program use a loop to generate the table that was written one line at a time in Fig. 9.5. Because we can loop through array `frequency` to help produce the output, we do not have to enumerate each XHTML table row as we did in Fig. 9.5.

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 10.6: RollDie.html -->
 6  <!-- Dice-rolling program using an array instead of a switch. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Roll a Six-Sided Die 6000 Times</title>
10        <style type = "text/css">
11           table { width: 15em }
12           th    { text-align: left }
13        </style>
14        <script type = "text/javascript">
15           <!--
16           var face;
17           var frequency = [ , 0, 0, 0, 0, 0, 0 ]; // leave frequency[0]
18                                                   // uninitialized
19
20           // summarize results
21           for ( var roll = 1; roll <= 6000; ++roll )
22           {
23              face = Math.floor( 1 + Math.random() * 6 );
24              ++frequency[ face ];
25           } // end for
26
27           document.writeln( "<table border = \"1\"><thead>" );
28           document.writeln( "<th>Face</th>" +
29              "<th>Frequency</th></thead><tbody>" );
30
31           // generate entire table of frequencies for each face
32           for ( face = 1; face < frequency.length; ++face )
33              document.writeln( "<tr><td>" + face + "</td><td>" +
34                 frequency[ face ] + "</td></tr>" );
35
36           document.writeln( "</tbody></table>" );
37           // -->
38        </script>
39     </head>
40     <body>
41        <p>Click Refresh (or Reload) to run the script again</p>
42     </body>
43  </html>
```

Fig. 10.6 | Dice-rolling program using an array instead of a `switch`. (Part 1 of 2.)


Roll a Six-Sided Die 6000 Times - Windows Internet Explorer
C:\examples\ch10\RollDie.html
Google
Roll a Six-Sided Die 6000 Times
Page
Tools
Done
My Computer
100%

Face	Frequency
1	1026
2	962
3	1032
4	1007
5	978
6	995

Click Refresh (or Reload) to run the script again

Fig. 10.6 | Dice-rolling program using an array instead of a `switch`. (Part 2 of 2.)

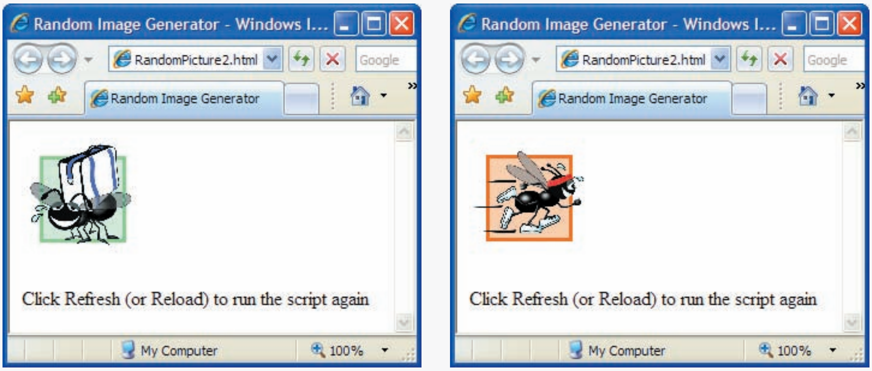
10.5 Random Image Generator Using Arrays

In Chapter 9, we created a random image generator that required image files to be named `1.gif`, `2.gif`, …, `7.gif`. In this example (Fig. 10.7), we create a more elegant random image generator that does not require the image filenames to be integers. This version of the random image generator uses an array `pictures` to store the names of the image files as strings. The script generates a random integer and uses it as a subscript into the `pictures` array. The script outputs an XHTML `img` element whose `src` attribute contains the image filename located in the randomly selected position in the `pictures` array.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 10.7: RandomPicture2.html -->
<!-- Random image generation using arrays. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Random Image Generator</title>
      <style type = "text/css">
         table { width: 15em }
         th    { text-align: left }
      </style>
      <script type = "text/javascript">
         <!--
         var pictures =
            [ "CPE", "EPT", "GPP", "GUI", "PERF", "PORT", "SEO" ];

         // pick a random image from the pictures array and displays by
         // creating an img tag and appending the src attribute to the
         // filename
```

Fig. 10.7 | Random image generation using arrays. (Part 1 of 2.)

```
22            document.write ( "<img src = \"" +
23               pictures[ Math.floor( Math.random() * 7 ) ] + ".gif\" />" );
24            // -->
25         </script>
26      </head>
27      <body>
28         <p>Click Refresh (or Reload) to run the script again</p>
29      </body>
30   </html>
```



Click Refresh (or Reload) to run the script again

Click Refresh (or Reload) to run the script again

Fig. 10.7 | Random image generation using arrays. (Part 2 of 2.)

The script declares the array `pictures` in lines 16–17 and initializes it with the names of seven image files. Lines 22–23 create the `img` tag that displays the random image on the web page. Line 22 opens the `img` tag and begins the `src` attribute. Line 23 generates a random integer from 0 to 6 as an index into the `pictures` array, the result of which is a randomly selected image filename. The expression

```
pictures[ Math.floor( Math.random() * 7 ) ]
```

evaluates to a string from the `pictures` array, which then is written to the document (line 23). Line 23 completes the `img` tag with the extension of the image file (`.gif`).

10.6 References and Reference Parameters

Two ways to pass arguments to functions (or methods) in many programming languages are pass-by-value and pass-by-reference. When an argument is passed to a function by value, a *copy* of the argument's value is made and is passed to the called function. In JavaScript, numbers, boolean values and strings are passed to functions by value.

With pass-by-reference, the caller gives the called function direct access to the caller's data and allows it to modify the data if it so chooses. This procedure is accomplished by passing to the called function the actual location in memory (also called the address) where the data resides. Pass-by-reference can improve performance because it can eliminate the overhead of copying large amounts of data, but it can weaken security because the called function can access the caller's data. In JavaScript, all objects (and thus all `Arrays`) are passed to functions by reference.

Error-Prevention Tip 10.3

With pass-by-value, changes to the copy of the called function do not affect the original variable's value in the calling function. This prevents the accidental side effects that so greatly hinder the development of correct and reliable software systems.

Software Engineering Observation 10.2

Unlike some other languages, JavaScript does not allow the programmer to choose whether to pass each argument by value or by reference. Numbers, boolean values and strings are passed by value. Objects are passed to functions by reference. When a function receives a reference to an object, the function can manipulate the object directly.

Software Engineering Observation 10.3

When returning information from a function via a `return` *statement, numbers and boolean values are always returned by value (i.e., a copy is returned), and objects are always returned by reference (i.e., a reference to the object is returned). Note that, in the pass-by-reference case, it is not necessary to return the new value, since the object is already modified.*

To pass a reference to an object into a function, simply specify the reference name in the function call. Normally, the reference name is the identifier that the program uses to manipulate the object. Mentioning the reference by its parameter name in the body of the called function actually refers to the original object in memory, and the original object can be accessed directly by the called function.

`Array`s are objects in JavaScript, so `Array`s are passed to a function by reference—a called function can access the elements of the caller's original `Array`s. The name of an array actually is a reference to an object that contains the array elements and the `length` variable, which indicates the number of elements in the array. In the next section, we demonstrate pass-by-value and pass-by-reference, using arrays.

10.7 Passing Arrays to Functions

To pass an array argument to a function, specify the name of the array (a reference to the array) without brackets. For example, if array `hourlyTemperatures` has been declared as

```
var hourlyTemperatures = new Array( 24 );
```

then the function call

```
modifyArray( hourlyTemperatures );
```

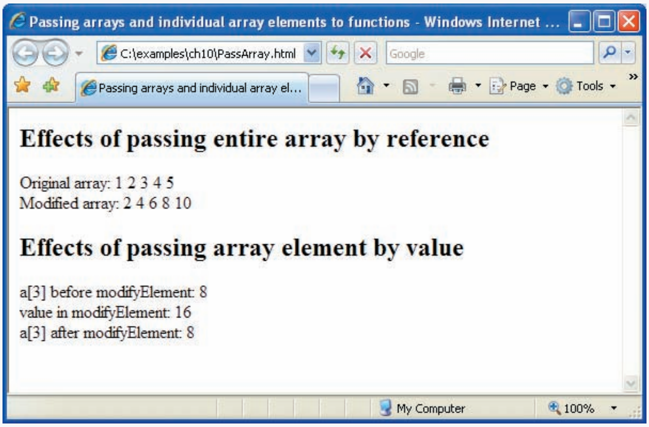
passes array `hourlyTemperatures` to function `modifyArray`. As stated in Section 10.2, every array object in JavaScript knows its own size (via the `length` attribute). Thus, when we pass an array object into a function, we do not pass the size of the array separately as an argument. Figure 10.3 illustrated this concept when we passed `Array`s `n1` and `n2` to function `outputArray` to display each `Array`'s contents.

Although entire arrays are passed by reference, *individual numeric and boolean array elements* are passed *by value* exactly as simple numeric and boolean variables are passed (the objects referred to by individual elements of an `Array` of objects are still passed by reference). Such simple single pieces of data are called scalars, or scalar quantities. To pass an array element to a function, use the subscripted name of the element as an argument in the function call.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 10.8: PassArray.html -->
<!-- Passing arrays and individual array elements to functions. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Passing arrays and individual array
         elements to functions</title>
      <script type = "text/javascript">
         <!--
         var a = [ 1, 2, 3, 4, 5 ];

         document.writeln( "<h2>Effects of passing entire " +
            "array by reference</h2>" );
         outputArray( "Original array: ", a );

         modifyArray( a );  // array a passed by reference

         outputArray( "Modified array: ", a );

         document.writeln( "<h2>Effects of passing array " +
            "element by value</h2>" +
            "a[3] before modifyElement: " + a[ 3 ] );

         modifyElement( a[ 3 ] ); // array element a[3] passed by value

         document.writeln( "<br />a[3] after modifyElement: " + a[ 3 ] );

         // outputs heading followed by the contents of "theArray"
         function outputArray( heading, theArray )
         {
            document.writeln(
               heading + theArray.join( " " ) + "<br />" );
         } // end function outputArray

         // function that modifies the elements of an array
         function modifyArray( theArray )
         {
            for ( var j in theArray )
               theArray[ j ] *= 2;
         } // end function modifyArray

         // function that modifies the value passed
         function modifyElement( e )
         {
            e *= 2; // scales element e only for the duration of the
                    // function
            document.writeln( "<br />value in modifyElement: " + e );
         } // end function modifyElement
         // -->
      </script>
```

Fig. 10.8 | Passing arrays and individual array elements to functions. (Part 1 of 2.)

```
54      </head><body></body>
55   </html>
```


Passing arrays and individual array elements to functions - Windows Internet ...
C:\examples\ch10\PassArray.html
Google
Passing arrays and individual array el...
Page
Tools
My Computer
100%

Effects of passing entire array by reference

```
Original array: 1 2 3 4 5
Modified array: 2 4 6 8 10
```

Effects of passing array element by value

```
a[3] before modifyElement: 8
value in modifyElement: 16
a[3] after modifyElement: 8
```

Fig. 10.8 | Passing arrays and individual array elements to functions. (Part 2 of 2.)

For a function to receive an `Array` through a function call, the function's parameter list must specify a parameter that will refer to the `Array` in the body of the function. Unlike other programming languages, JavaScript does not provide a special syntax for this purpose. JavaScript simply requires that the identifier for the `Array` be specified in the parameter list. For example, the function header for function `modifyArray` might be written as

```
function modifyArray( b )
```

indicating that `modifyArray` expects to receive a parameter named `b` (the argument supplied in the calling function must be an `Array`). Arrays are passed by reference, and therefore when the called function uses the array name `b`, it refers to the actual array in the caller (array `hourlyTemperatures` in the preceding call). The script in Fig. 10.8 demonstrates the difference between passing an entire array and passing an array element.

Software Engineering Observation 10.4

JavaScript does not check the number of arguments or types of arguments that are passed to a function. It is possible to pass any number of values to a function. JavaScript will attempt to perform conversions when the values are used.

The statement in line 17 invokes function `outputArray` to display the contents of array `a` before it is modified. Function `outputArray` (defined in lines 32–36) receives a string to output and the array to output. The statement in lines 34–35 uses `Array` method **`join`** to create a string containing all the elements in `theArray`. Method `join` takes as its argument a string containing the **separator** that should be used to separate the elements of the array in the string that is returned. If the argument is not specified, the empty string is used as the separator.

Line 19 invokes function `modifyArray` (lines 39–43) and passes it array `a`. The `modifyArray` function multiplies each element by 2. To illustrate that array `a`'s elements were modified, the statement in line 21 invokes function `outputArray` again to display the contents of array a after it is modified. As the screen capture shows, the elements of a are indeed modified by modifyArray.

To show the value of a[3] before the call to modifyElement, line 25 outputs the value of a[3]. Line 27 invokes modifyElement (lines 46–51) and passes a[3] as the argument. Remember that a[3] actually is one integer value in the array a. Also remember that numeric values and boolean values are always passed to functions by value. Therefore, a copy of a[3] is passed. Function modifyElement multiplies its argument by 2 and stores the result in its parameter e. The parameter of function modifyElement is a local variable in that function, so when the function terminates, the local variable is no longer accessible. Thus, when control is returned to the main script, the unmodified value of a[3] is displayed by the statement in line 29.

10.8 Sorting Arrays

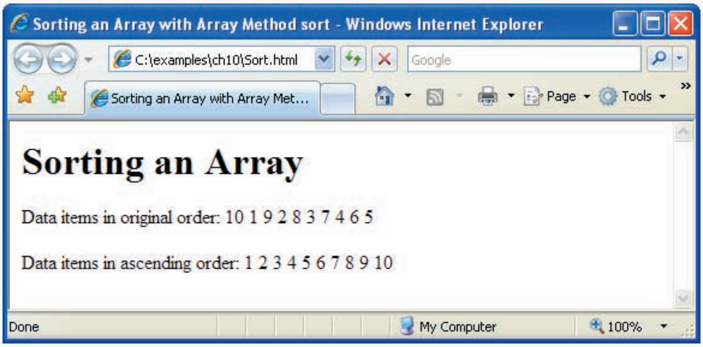
Sorting data (putting data in a particular order, such as ascending or descending) is one of the most important computing functions. A bank sorts all checks by account number so that it can prepare individual bank statements at the end of each month. Telephone companies sort their lists of accounts by last name and, within that, by first name, to make it easy to find phone numbers. Virtually every organization must sort some data—in many cases, massive amounts of data. Sorting data is an intriguing problem that has attracted some of the most intense research efforts in the field of computer science.

The Array object in JavaScript has a built-in method **sort** for sorting arrays. Figure 10.9 demonstrates the Array object's sort method.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 10.9: Sort.html -->
<!-- Sorting an array with sort. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Sorting an Array with Array Method sort</title>
      <script type = "text/javascript">
         <!--
         var a = [ 10, 1, 9, 2, 8, 3, 7, 4, 6, 5 ];

         document.writeln( "<h1>Sorting an Array</h1>" );
         outputArray( "Data items in original order: ", a );
         a.sort( compareIntegers );  // sort the array
         outputArray( "Data items in ascending order: ", a );

         // output the heading followed by the contents of theArray
         function outputArray( heading, theArray )
         {
            document.writeln( "<p>" + heading +
               theArray.join( " " ) + "</p>" );
         } // end function outputArray
```

Fig. 10.9 | Sorting an array with sort. (Part 1 of 2.)

```
25
26             // comparison function for use with sort
27             function compareIntegers( value1, value2 )
28             {
29                return parseInt( value1 ) - parseInt( value2 );
30             } // end function compareIntegers
31             // -->
32          </script>
33       </head><body></body>
34    </html>
```


Sorting an Array with Array Method sort - Windows Internet Explorer
C:\examples\ch10\Sort.html
Google
Sorting an Array with Array Met...
Page
Tools
Done
My Computer
100%

Sorting an Array

Data items in original order: 10 1 9 2 8 3 7 4 6 5

Data items in ascending order: 1 2 3 4 5 6 7 8 9 10

Fig. 10.9 | Sorting an array with `sort`. (Part 2 of 2.)

By default, `Array` method `sort` (with no arguments) uses string comparisons to determine the sorting order of the `Array` elements. The strings are compared by the ASCII values of their characters. [*Note:* String comparison is discussed in more detail in Chapter 11, JavaScript: Objects.] In this example, we'd like to sort an array of integers.

Method `sort` takes as its optional argument the name of a function (called the **comparator function**) that compares its two arguments and returns one of the following:

- a negative value if the first argument is less than the second argument
- zero if the arguments are equal, or
- a positive value if the first argument is greater than the second argument

This example uses function `compareIntegers` (defined in lines 27–30) as the comparator function for method `sort`. It calculates the difference between the integer values of its two arguments (function `parseInt` ensures that the arguments are handled properly as integers). If the first argument is less than the second argument, the difference will be a negative value. If the arguments are equal, the difference will be zero. If the first argument is greater than the second argument, the difference will be a positive value.

Line 16 invokes `Array` object `a`'s `sort` method and passes function `compareIntegers` as an argument. Method `sort` receives function `compareIntegers` as an argument, then uses the function to compare elements of the `Array` `a` to determine their sorting order.

Software Engineering Observation 10.5

Functions in JavaScript are considered to be data. Therefore, functions can be assigned to variables, stored in `Arrays` *and passed to functions just like other data types.*

10.9 Searching Arrays: Linear Search and Binary Search

Often, a programmer will be working with large amounts of data stored in arrays. It may be necessary to determine whether an array contains a value that matches a certain key value. The process of locating a particular element value in an array is called searching. In this section we discuss two searching techniques—the simple linear search technique (Fig. 10.10) and the more efficient binary search technique (Fig. 10.11).

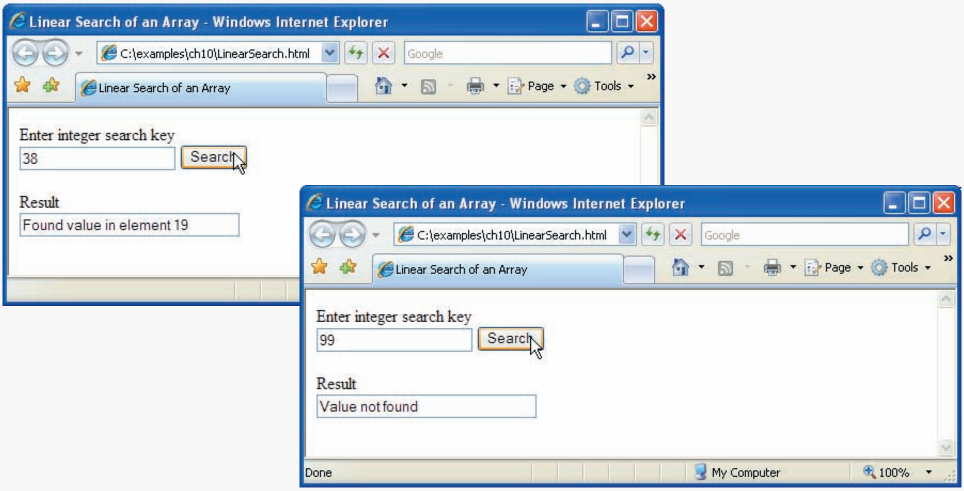
Searching an Array with Linear Search

The script in Fig. 10.10 performs a linear search on an array. Function `linearSearch` (defined in lines 42–50) uses a `for` statement containing an `if` statement to compare each element of an array with a search key (lines 45–47). If the search key is found, the function returns the subscript value (line 47) of the element to indicate the exact position of the search key in the array. [*Note:* The loop (lines 45–47) in the `linearSearch` function terminates, and the function returns control to the caller as soon as the `return` statement in its body executes.] If the search key is not found, the function returns a value of `-1`. The function returns the value `-1` because it is not a valid subscript number.

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 10.10: LinearSearch.html -->
 6  <!-- Linear search of an array. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Linear Search of an Array</title>
10        <script type = "text/javascript">
11           <!--
12           var a = new Array( 100 );  // create an Array
13
14           // fill Array with even integer values from 0 to 198
15           for ( var i = 0; i < a.length; ++i )
16              a[ i ] = 2 * i;
17
18           // function called when "Search" button is pressed
19           function buttonPressed()
20           {
21              // get the input text field
22              var inputVal = document.getElementById( "inputVal" );
23
24              // get the result text field
25              var result = document.getElementById( "result" );
26
27              // get the search key from the input text field
28              var searchKey = inputVal.value;
29
30              // Array a is passed to linearSearch even though it
31              // is a global variable. Normally an array will
32              // be passed to a method for searching.
```

Fig. 10.10 | Linear search of an array. (Part 1 of 2.)

```
            var element = linearSearch( a, parseInt( searchKey ) );

            if ( element != -1 )
               result.value = "Found value in element " + element;
            else
               result.value = "Value not found";
         } // end function buttonPressed

         // Search "theArray" for the specified "key" value
         function linearSearch( theArray, key )
         {
            // iterates through each element of the array in order
            for ( var n = 0; n < theArray.length; ++n )
               if ( theArray[ n ] == key )
                  return n;

            return -1;
         } // end function linearSearch
         // -->
      </script>
   </head>

   <body>
      <form action  = "">
         <p>Enter integer search key<br />
         <input id = "inputVal" type = "text" />
         <input type = "button" value = "Search"
                onclick = "buttonPressed()" /><br /></p>
         <p>Result<br />
         <input id = "result" type = "text" size = "30" /></p>
      </form>
   </body>
</html>
```


Linear Search of an Array - Windows Internet Explorer
C:\examples\ch10\LinearSearch.html
Google
Linear Search of an Array
Page
Tools
Search
Done
My Computer
100%

Enter integer search key
38
Result
Found value in element 19

Enter integer search key
99
Result
Value not found

Fig. 10.10 | Linear search of an array. (Part 2 of 2.)

If the array being searched is not in any particular order, the value is just as likely to be found in the first element as the last. On average, therefore, the program will have to compare the search key with half the elements of the array.

The program contains a 100-element array (defined in line 12) filled with the even integers from 0 to 198. The user types the search key in a text field (defined in the XHTML form in lines 56–63) and clicks the **Search** button to start the search. [*Note:* The array is passed to `linearSearch` even though the array is a global script variable. We do this because arrays are normally passed to functions for searching.]

Searching an Array with Binary Search

The linear search method works well for small arrays or for unsorted arrays. However, for large arrays, linear searching is inefficient. The **binary search algorithm** is more efficient, but it requires that the array be sorted. The first iteration of this algorithm tests the middle element in the array. If this matches the search key, the algorithm ends. Assuming the array is sorted in ascending order, then if the search key is less than the middle element, it cannot match any element in the second half of the array and the algorithm continues with only the first half of the array (i.e., the first element up to, but not including, the middle element). If the search key is greater than the middle element, it cannot match any element in the first half of the array and the algorithm continues with only the second half of the array (i.e., the element after the middle element through the last element). Each iteration tests the middle value of the remaining portion of the array. If the search key does not match the element, the algorithm eliminates half of the remaining elements. The algorithm ends by either finding an element that matches the search key or reducing the subarray to zero size.

As an example consider the sorted 15-element array

```
2   3   5   10   27   30   34   51   56   65   77   81   82   93   99
```

and a search key of 65. A program implementing the binary search algorithm would first check whether 51 is the search key (because 51 is the middle element of the array). The search key (65) is larger than 51, so 51 is discarded along with the first half of the array (all elements smaller than 51.) Next, the algorithm checks whether 81 (the middle element of the remainder of the array) matches the search key. The search key (65) is smaller than 81, so 81 is discarded along with the elements larger than 81. After just two tests, the algorithm has narrowed the number of values to check to three (56, 65 and 77). The algorithm then checks 65 (which indeed matches the search key), and returns the index of the array element containing 65. This algorithm required just three comparisons to determine whether the search key matched an element of the array. Using a linear search algorithm would have required 10 comparisons. [*Note:* In this example, we have chosen to use an array with 15 elements so that there will always be an obvious middle element in the array. With an even number of elements, the middle of the array lies between two elements. We implement the algorithm to choose the lower of those two elements.]

Figure 10.11 presents the iterative version of function `binarySearch` (lines 40–64). Function `binarySearch` is called from line 31 of function `buttonPressed` (lines 18–37)—the event handler for the **Search** button in the XHTML form. Function `binarySearch` receives two arguments—an array called `theArray` (the array to search) and `key` (the search key). The array is passed to `binarySearch` even though the array is a global variable. Once again, we do this because an array is normally passed to a function for searching. If key matches the middle element of a subarray (line 55), middle (the subscript of the current element) is returned, to indicate that the value was found and the search is complete. If key does not match the middle element of a subarray, the low subscript or the high subscript (both declared in the function) is adjusted, so that a smaller subarray can be searched. If key is less than the middle element (line 57), the high subscript is set to middle - 1 and the search is continued on the elements from low to middle - 1. If key is greater than the middle element (line 59), the low subscript is set to middle + 1 and the search is continued on the elements from middle + 1 to high. These comparisons are performed by the nested if…else statement in lines 55–60.

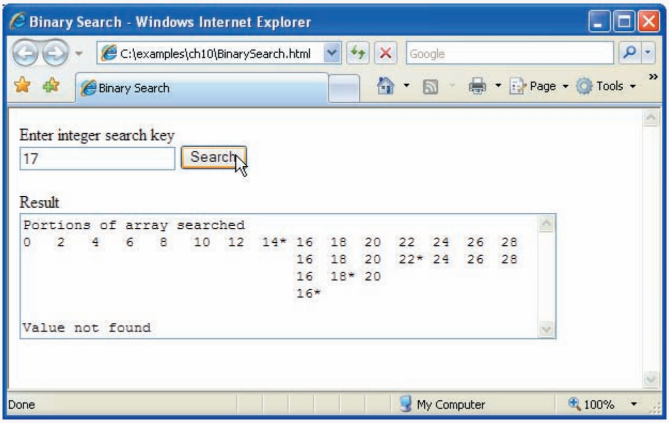
```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 10.11: BinarySearch.html -->
 6  <!-- Binary search of an array. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Binary Search</title>
10        <script type = "text/javascript">
11           <!--
12           var a = new Array( 15 );
13
14           for ( var i = 0; i < a.length; ++i )
15              a[ i ] = 2 * i;
16
17           // function called when "Search" button is pressed
18           function buttonPressed()
19           {
20              var inputVal = document.getElementById( "inputVal" );
21              var result = document.getElementById( "result" );
22              var searchKey = inputVal.value;
23
24              result.value = "Portions of array searched\n";
25
26              // Array a is passed to binarySearch even though it
27              // is a global variable. This is done because
28              // normally an array is passed to a method
29              // for searching.
30              var element =
31                 binarySearch( a, parseInt( searchKey ) );
32
33              if ( element != -1 )
34                 result.value += "\nFound value in element " + element;
35              else
36                 result.value += "\nValue not found";
37           } // end function buttonPressed
38
```

Fig. 10.11 | Binary search of an array. (Part 1 of 3.)

```
         // binary search function
         function binarySearch( theArray, key )
         {
            var low = 0; // low subscript
            var high = theArray.length - 1; // high subscript
            var middle; // middle subscript

            while ( low <= high ) {
               middle = ( low + high ) / 2;

               // The following line is used to display the
               // part of theArray currently being manipulated
               // during each iteration of the binary
               // search loop.
               buildOutput( theArray, low, middle, high );

               if ( key == theArray[ middle ] )  // match
                  return middle;
               else if ( key < theArray[ middle ] )
                  high = middle - 1; // search low end of array
               else
                  low = middle + 1; // search high end of array
            } // end while

            return -1;   // searchKey not found
         } // end function binarySearch

         // Build one row of output showing the current
         // part of the array being processed.
         function buildOutput( theArray, low, mid, high )
         {
            var result = document.getElementById( "result" );

            for ( var i = 0; i < theArray.length; i++ )
            {
               if ( i < low || i > high )
                  result.value += "    ";
               else if ( i == mid ) // mark middle element in output
                  result.value += theArray[ i ] +
                     ( theArray[ i ] < 10 ? "*  " : "* " );
               else
                  result.value += theArray[ i ] +
                     ( theArray[ i ] < 10 ? "   " : "  " );
            } // end for

            result.value += "\n";
         } // end function buildOutput
         // -->
      </script>
   </head>

```

Fig. 10.11 | Binary search of an array. (Part 2 of 3.)

```
   <body>
      <form action = "">
         <p>Enter integer search key<br />
         <input id = "inputVal" type = "text" />
         <input type = "button" value = "Search"
            onclick = "buttonPressed()" /><br /></p>
         <p>Result<br />
         <textarea id = "result" rows = "7" cols = "60">
         </textarea></p>
      </form>
   </body>
</html>
```

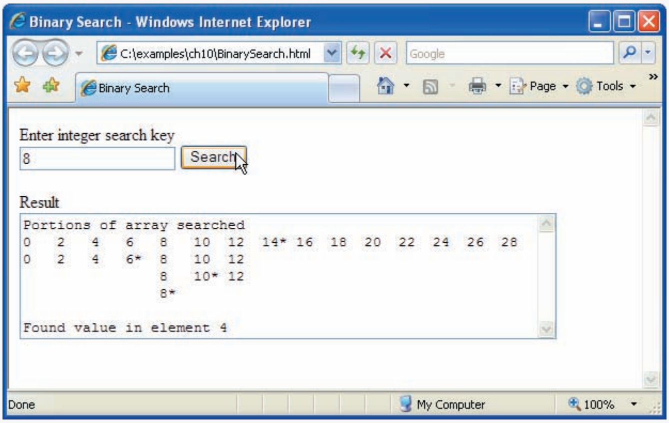

Binary Search - Windows Internet Explorer
C:\examples\ch10\BinarySearch.html
Google
Binary Search
Page
Tools
Search
Done
My Computer
100%

Enter integer search key

17

Result

```
Portions of array searched
0    2    4    6    8    10   12   14*  16   18   20   22   24   26   28
                                        16   18   20   22*  24   26   28
                                        16   18*  20
                                        16*

Value not found
```


Binary Search - Windows Internet Explorer
C:\examples\ch10\BinarySearch.html
Google
Binary Search
Page
Tools
Search
Done
My Computer
100%

Enter integer search key

8

Result

```
Portions of array searched
0    2    4    6    8    10   12   14*  16   18   20   22   24   26   28
0    2    4    6*   8    10   12
                    8    10*  12
                    8*

Found value in element 4
```

Fig. 10.11 | Binary search of an array. (Part 3 of 3.)

In a worst-case scenario, searching an array of 1023 elements will take only 10 comparisons using a binary search. Repeatedly dividing 1024 by 2 (because after each comparison we are able to eliminate half of the array) yields the values 512, 256, 128, 64, 32, 16, 8, 4, 2 and 1. The number 1024 (2^{10}) is divided by 2 only ten times to get the value 1. Dividing by 2 is equivalent to one comparison in the binary search algorithm. An array of about one million elements takes a maximum of 20 comparisons to find the key. An array of about one billion elements takes a maximum of 30 comparisons to find the key. When searching a sorted array, this is a tremendous increase in performance over the linear search that required comparing the search key to an average of half the elements in the array. For a one-billion-element array, this is the difference between an average of 500 million comparisons and a maximum of 30 comparisons! The maximum number of comparisons needed for the binary search of any sorted array is the exponent of the first power of 2 greater than the number of elements in the array.

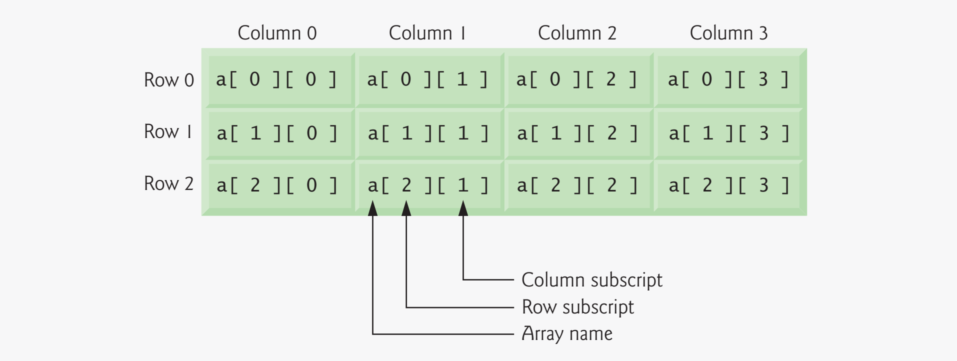
Our program uses a 15-element array (defined at line 12). The first power of 2 greater than the number of elements is 16 (2^4), so `binarySearch` requires at most four comparisons to find the `key`. To illustrate this, line 53 calls method `buildOutput` (declared in lines 68–85) to output each subarray during the binary search process. Method `buildOutput` marks the middle element in each subarray with an asterisk (*) to indicate the element with which the `key` is compared. No matter what search key is entered, each search in this example results in a maximum of four lines of output—one per comparison.

10.10 Multidimensional Arrays

Multidimensional arrays with two subscripts are often used to represent tables of values consisting of information arranged in **rows** and **columns**. To identify a particular table element, we must specify the two subscripts; by convention, the first identifies the element's row, and the second identifies the element's column. Arrays that require two subscripts to identify a particular element are called **two-dimensional arrays**.

Multidimensional arrays can have more than two dimensions. JavaScript does not support multidimensional arrays directly, but does allow the programmer to specify arrays whose elements are also arrays, thus achieving the same effect. When an array contains one-dimensional arrays as its elements, we can imagine these one-dimensional arrays as rows of a table, and the positions in these arrays as columns. Figure 10.12 illustrates a two-dimensional array named `a` that contains three rows and four columns (i.e., a three-by-four array—three one-dimensional arrays, each with 4 elements). In general, an array with *m* rows and *n* columns is called an ***m*-by-*n* array**.

Every element in array `a` is identified in Fig. 10.12 by an element name of the form `a[ i ][ j ]`; `a` is the name of the array, and `i` and `j` are the subscripts that uniquely identify



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Fig. 10.12 | Two-dimensional array with three rows and four columns.

the row and column, respectively, of each element in a. Note that the names of the elements in the first row all have a first subscript of 0; the names of the elements in the fourth column all have a second subscript of 3.

Arrays of One-Dimensional Arrays

Multidimensional arrays can be initialized in declarations like a one-dimensional array. Array b with two rows and two columns could be declared and initialized with the statement

```
var b = [ [ 1, 2 ], [ 3, 4 ] ];
```

The values are grouped by row in square brackets. The array [1, 2] initializes element b[0], and the array [3, 4] initializes element b[1]. So 1 and 2 initialize b[0][0] and b[0][1], respectively. Similarly, 3 and 4 initialize b[1][0] and b[1][1], respectively. The interpreter determines the number of rows by counting the number of sub initializer lists—arrays nested within the outermost array. The interpreter determines the number of columns in each row by counting the number of values in the sub-array that initializes the row.

Two-Dimensional Arrays with Rows of Different Lengths

The rows of a two-dimensional array can vary in length. The declaration

```
var b = [ [ 1, 2 ], [ 3, 4, 5 ] ];
```

creates array b with row 0 containing two elements (1 and 2) and row 1 containing three elements (3, 4 and 5).

Creating Two-Dimensional Arrays with **new**

A multidimensional array in which each row has a different number of columns can be allocated dynamically, as follows:

```
var b;
b = new Array( 2 );        // allocate rows
b[ 0 ] = new Array( 5 );  // allocate columns for row 0
b[ 1 ] = new Array( 3 );  // allocate columns for row 1
```

The preceding code creates a two-dimensional array with two rows. Row 0 has five columns, and row 1 has three columns.

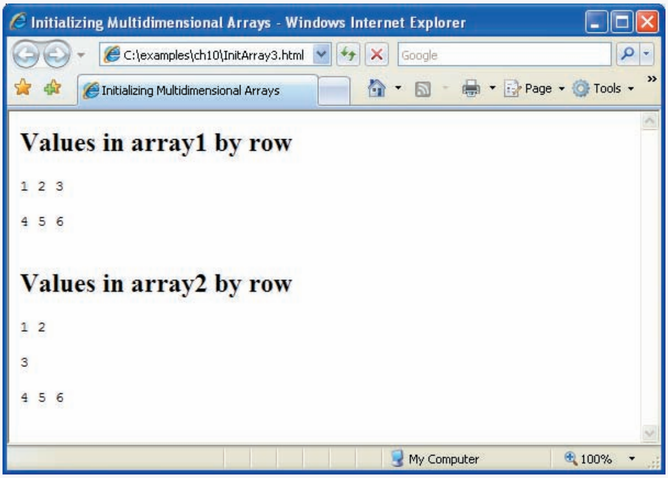
Two-Dimensional Array Example: Displaying Element Values

Figure 10.13 initializes two-dimensional arrays in declarations and uses nested for…in loops to traverse the arrays (i.e., manipulate every element of the array).

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 10.13: InitArray3.html -->
<!-- Initializing multidimensional arrays. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
```

Fig. 10.13 | Initializing multidimensional arrays. (Part 1 of 2.)

```
   <head>
      <title>Initializing Multidimensional Arrays</title>
      <script type = "text/javascript">
         <!--
         var array1 = [ [ 1, 2, 3 ], // first row
                        [ 4, 5, 6 ] ]; // second row
         var array2 = [ [ 1, 2 ], // first row
                        [ 3 ], // second row
                        [ 4, 5, 6 ] ]; // third row

         outputArray( "Values in array1 by row", array1 );
         outputArray( "Values in array2 by row", array2 );

         function outputArray( heading, theArray )
         {
            document.writeln( "<h2>" + heading + "</h2><pre>" );

            // iterates through the set of one-dimensional arrays
            for ( var i in theArray )
            {
               // iterates through the elements of each one-dimensional
               // array
               for ( var j in theArray[ i ] )
                  document.write( theArray[ i ][ j ] + " " );

               document.writeln( "<br />" );
            } // end for

            document.writeln( "</pre>" );
         } // end function outputArray
         // -->
      </script>
   </head><body></body>
</html>
```


Initializing Multidimensional Arrays - Windows Internet Explorer
C:\examples\ch10\InitArray3.html
Google
Initializing Multidimensional Arrays
Page
Tools
My Computer
100%

Values in array1 by row

```
1 2 3
4 5 6
```

Values in array2 by row

```
1 2
3
4 5 6
```

Fig. 10.13 | Initializing multidimensional arrays. (Part 2 of 2.)

The program declares two arrays in main script (in the XHTML head element). The declaration of array1 (lines 12–13 provides six initializers in two sublists. The first sublist initializes the first row of the array to the values 1, 2 and 3; the second sublist initializes the second row of the array to the values 4, 5 and 6. The declaration of array2 (lines 14–16) provides six initializers in three sublists. The sublist for the first row explicitly initializes the first row to have two elements, with values 1 and 2, respectively. The sublist for the second row initializes the second row to have one element, with value 3. The sublist for the third row initializes the third row to the values 4, 5 and 6.

The script calls function outputArray from lines 18–19 to display each array's elements in the web page. Function outputArray (lines 21–37) receives two arguments—a string heading to output before the array and the array to output (called theArray). Note the use of a nested for…in statement to output the rows of each two-dimensional array. The outer for…in statement iterates over the rows of the array. The inner for…in statement iterates over the columns of the current row being processed. The nested for…in statement in this example could have been written with for statements, as follows:

```
for ( var i = 0; i < theArray.length; ++i ) {

   for ( var j = 0; j < theArray[ i ].length; ++j )
      document.write( theArray[ i ][ j ] + " " );

   document.writeln( "<br />" );
}
```

In the outer for statement, the expression theArray.length determines the number of rows in the array. In the inner for statement, the expression theArray[i].length determines the number of columns in each row of the array. This condition enables the loop to determine, for each row, the exact number of columns.

Common Multidimensional-Array Manipulations with for *and* for…in *Statements*

Many common array manipulations use for or for…in repetition statements. For example, the following for statement sets all the elements in the third row of array a in Fig. 10.12 to zero:

```
for ( var col = 0; col < a[ 2 ].length; ++col )
   a[ 2 ][ col ] = 0;
```

We specified the *third* row; therefore, we know that the first subscript is always 2 (0 is the first row and 1 is the second row). The for loop varies only the second subscript (i.e., the column subscript). The preceding for statement is equivalent to the assignment statements

```
a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;
```

The following for…in statement is also equivalent to the preceding for statement:

```
for ( var col in a[ 2 ] )
   a[ 2 ][ col ] = 0;
```

The following nested `for` statement determines the total of all the elements in array `a`:

```
var total = 0;

for ( var row = 0; row < a.length; ++row )

   for ( var col = 0; col < a[ row ].length; ++col )
      total += a[ row ][ col ];
```

The `for` statement totals the elements of the array, one row at a time. The outer `for` statement begins by setting the `row` subscript to `0`, so that the elements of the first row may be totaled by the inner `for` statement. The outer `for` statement then increments `row` to `1`, so that the elements of the second row can be totaled. Then the outer `for` statement increments `row` to `2`, so that the elements of the third row can be totaled. The result can be displayed when the nested `for` statement terminates. The preceding `for` statement is equivalent to the following `for…in` statement:

```
var total = 0;

for ( var row in a )

   for ( var col in a[ row ] )
      total += a[ row ][ col ];
```

10.11 Building an Online Quiz

Online quizzes and polls are popular web applications often used for educational purposes or just for fun. Web developers typically build quizzes using simple XHTML forms and process the results with JavaScript. Arrays allow a programmer to represent several possible answer choices in a single data structure. Figure 10.14 contains an online quiz consisting of one question. The quiz page contains one of the tip icons used throughout this book and an XHTML form in which the user identifies the type of tip the image represents by selecting one of four radio buttons. After the user selects one of the radio button choices and submits the form, the script determines whether the user selected the correct type of tip to match the mystery image. The JavaScript function that checks the user's answer combines several of the concepts from the current chapter and previous chapters in a concise and useful script.

Before we discuss the script code, we first discuss the `body` element (lines 25–48) of the XHTML document. The `body`'s GUI components play an important role in the script.

Lines 26–47 define the `form` that presents the quiz to users. Line 26 begins the `form` element and specifies the `onsubmit` attribute to `"checkAnswers()"`, indicating that the interpreter should execute the JavaScript function `checkAnswers` (lines 12–21) when the user submits the form (i.e., clicks the **Submit** button or presses *Enter*).

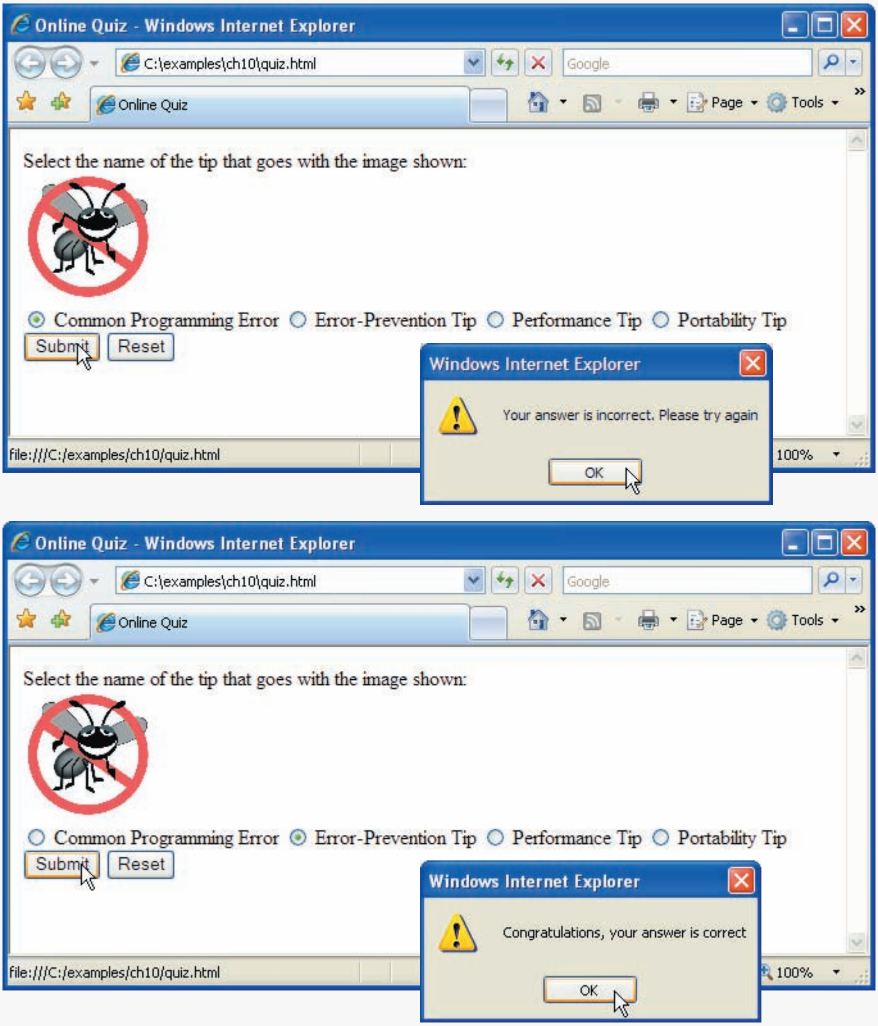
Line 29 adds the tip image to the page. Lines 32–42 display the radio buttons and corresponding `label`s that display possible answer choices. Lines 44–45 add the `submit` and `reset` buttons to the page.

We now examine the script used to check the answer submitted by the user. Lines 12–21 declare the function `checkAnswers` that contains all the JavaScript required to grade the quiz. The `if…else` statement in lines 17–20 determines whether the user answered the question correctly. The image that the user is asked to identify is the Error-Prevention Tip icon. Thus the correct answer to the quiz corresponds to the second radio button.

An XHTML form's elements can be accessed individually using `getElementById` or through the **`elements` property** of the containing `form` object. The `elements` property

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 10.14: quiz.html -->
<!-- Online quiz graded with JavaScript. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Online Quiz</title>
      <script type = "text/JavaScript">
         <!--
         function checkAnswers()
         {
            var myQuiz = document.getElementById( "myQuiz" );

            // determine whether the answer is correct
            if ( myQuiz.elements[ 1 ].checked )
               alert( "Congratulations, your answer is correct" );
            else // if the answer is incorrect
               alert( "Your answer is incorrect. Please try again" );
         } // end function checkAnswers
         -->
      </script>
   </head>
   <body>
      <form id = "myQuiz" onsubmit = "checkAnswers()" action = "">
         <p>Select the name of the tip that goes with the
            image shown:<br />
            <img src="EPT.gif" alt="mystery tip"/>
            <br />

            <input type = "radio" name = "radiobutton" value = "CPE" />
            <label>Common Programming Error</label>

            <input type = "radio" name = "radiobutton" value = "EPT" />
            <label>Error-Prevention Tip</label>

            <input type = "radio" name = "radiobutton" value = "PERF" />
            <label>Performance Tip</label>

            <input type = "radio" name = "radiobutton" value = "PORT" />
            <label>Portability Tip</label><br />

            <input type = "submit" name = "submit" value = "Submit" />
            <input type = "reset" name = "reset" value = "Reset" />
         </p>
      </form>
   </body>
</html>
```

Fig. 10.14 | Online quiz graded with JavaScript. (Part 1 of 2.)



Online Quiz - Windows Internet Explorer

C:\examples\ch10\quiz.html

Select the name of the tip that goes with the image shown:

● Common Programming Error ○ Error-Prevention Tip ○ Performance Tip ○ Portability Tip

Submit Reset

Windows Internet Explorer

Your answer is incorrect. Please try again

OK

file:///C:/examples/ch10/quiz.html

Online Quiz - Windows Internet Explorer

C:\examples\ch10\quiz.html

Select the name of the tip that goes with the image shown:

○ Common Programming Error ● Error-Prevention Tip ○ Performance Tip ○ Portability Tip

Submit Reset

Windows Internet Explorer

Congratulations, your answer is correct

OK

file:///C:/examples/ch10/quiz.html

Fig. 10.14 | Online quiz graded with JavaScript. (Part 2 of 2.)

contains an array of all the `form`'s controls. The radio buttons are part of the XHTML form `myQuiz`, so we access the elements array in line 17 using dot notation (`myQuiz.elements[1]`). The array element `myQuiz.elements[1]` corresponds to the correct answer (i.e., the second radio button). Finally, line 17 determines whether the property **`checked`** of the second radio button is `true`. Property `checked` of a radio button is `true` when the radio button is selected, and it is `false` when the radio button is not selected. Recall that only one radio button may be selected at any given time. If property `myQuiz.elements[ 1 ].checked` is `true`, indicating that the correct answer is selected, the script alerts a congratulatory message. If property `checked` of the radio button is `false`, then the script alerts an alternate message (line 20).

10.12 Wrap-Up

In this chapter, we discussed how to store related data items into an array structure. After demonstrating the declaration and initialization of an array, we described how to access and manipulate individual elements of the array using subscripts. We then introduced the concept of an `Array` object, and learned that objects are passed to a method by reference instead of by value. We explored the sorting of arrays, and described two different methods of searching them. Finally, we discussed multidimensional arrays and how to use them to organize arrays in other arrays.

10.13 Web Resources

`www.deitel.com/javascript/`

The Deitel JavaScript Resource Center contains links to some of the best JavaScript resources on the web. There you'll find categorized links to JavaScript tools, code generators, forums, books, libraries, frameworks and more. Also check out the tutorials for all skill levels, from introductory to advanced. Be sure to visit the related Resource Centers on XHTML (`www.deitel.com/xhtml/`) and CSS 2.1 (`www.deitel.com/css21/`).

Summary

Section 10.1 Introduction

- Arrays are data structures consisting of related data items (sometimes called collections of data items).
- JavaScript arrays are "dynamic" entities in that they can change size after they are created.

Section 10.2 Arrays

- An array is a group of memory locations that all have the same name and normally are of the same type (although this attribute is not required in JavaScript).
- Each individual location is called an element. Any one of these elements may be referred to by giving the name of the array followed by the position number (an integer normally referred to as the subscript or index) of the element in square brackets (`[]`).
- The first element in every array is the zeroth element. In general, the *i*th element of array `c` is referred to as `c[i-1]`. Array names follow the same conventions as other identifiers.
- A subscripted array name is a left-hand-side expression—it can be used on the left side of an assignment to place a new value into an array element. It can also be used on the right side of an assignment operation to assign its value to another left-hand-side expression.
- Every array in JavaScript knows its own length, which it stores in its `length` attribute.

Section 10.3 Declaring and Allocating Arrays

- JavaScript arrays are represented by `Array` objects.
- The process of creating new objects using the new operator is known as creating an instance or instantiating an object, and operator `new` is known as the dynamic memory allocation operator.

Section 10.4 Examples Using Arrays

- Zero-based counting is usually used to iterate through arrays.

- JavaScript automatically reallocates an Array when a value is assigned to an element that is outside the bounds of the original Array. Elements between the last element of the original Array and the new element have undefined values.
- Arrays can be created using a comma-separated initializer list enclosed in square brackets ([and]). The array's size is determined by the number of values in the initializer list.
- The initial values of an array can also be specified as arguments in the parentheses following new Array. The size of the array is determined by the number of values in parentheses.
- JavaScript's for...in statement enables a script to perform a task for each element in an array. This process is known as iterating over the elements of an array.

Section 10.5 Random Image Generator Using Arrays

- We create a more elegant random image generator than the one in the previous chapter that does not require the image filenames to be integers by using a pictures array to store the names of the image files as strings and accessing the array using a randomized index.

Section 10.6 References and Reference Parameters

- Two ways to pass arguments to functions (or methods) in many programming languages are pass-by-value and pass-by-reference.
- When an argument is passed to a function by value, a *copy* of the argument's value is made and is passed to the called function.
- In JavaScript, numbers, boolean values and strings are passed to functions by value.
- With pass-by-reference, the caller gives the called function direct access to the caller's data and allows it to modify the data if it so chooses. Pass-by-reference can improve performance because it can eliminate the overhead of copying large amounts of data, but it can weaken security because the called function can access the caller's data.
- In JavaScript, all objects (and thus all Arrays) are passed to functions by reference.
- Arrays are objects in JavaScript, so Arrays are passed to a function by reference—a called function can access the elements of the caller's original Arrays. The name of an array is actually a reference to an object that contains the array elements and the length variable, which indicates the number of elements in the array.

Section 10.7 Passing Arrays to Functions

- To pass an array argument to a function, specify the name of the array (a reference to the array) without brackets.
- Although entire arrays are passed by reference, *individual numeric and boolean array elements* are passed *by value* exactly as simple numeric and boolean variables are passed. Such simple single pieces of data are called scalars, or scalar quantities. To pass an array element to a function, use the subscripted name of the element as an argument in the function call.
- The join method of an Array returns a string that contains all of the elements of an array, separated by the string supplied in the function's argument. If an argument is not specified, the empty string is used as the separator.

Section 10.8 Sorting Arrays

- Sorting data (putting data in a particular order, such as ascending or descending) is one of the most important computing functions.
- The Array object in JavaScript has a built-in method sort for sorting arrays.
- By default, Array method sort (with no arguments) uses string comparisons to determine the sorting order of the Array elements.

- Method `sort` takes as its optional argument the name of a function (called the comparator function) that compares its two arguments and returns a negative value, zero, or a positive value, if the first argument is less than, equal to, or greater than the second, respectively.
- Functions in JavaScript are considered to be data. Therefore, functions can be assigned to variables, stored in `Array`s and passed to functions just like other data types.

Section 10.9 Searching Arrays: Linear Search and Binary Search

- The linear search algorithm iterates through the elements of an array until it finds an element that matches a search key.
- If the array being searched is not in any particular order, it is just as likely that the value will be found in the first element as the last. On average, therefore, the program will have to compare the search key with half the elements of the array.
- The binary search algorithm is more efficient than the linear search algorithm, but it requires that the array be sorted.
- The binary search algorithm tests the middle element in the array and returns the index if it matches the search key. If not, it cuts the list in half, depending on whether the key is greater than or less than the middle element, and repeats the process on the remaining half of the sorted list. The algorithm ends by either finding an element that matches the search key or reducing the subarray to zero size.
- When searching a sorted array, the binary search provides a tremendous increase in performance over the linear search. For a one-billion-element array, this is the difference between an average of 500 million comparisons and a maximum of 30 comparisons.
- The maximum number of comparisons needed for the binary search of any sorted array is the exponent of the first power of 2 greater than the number of elements in the array.

Section 10.10 Multidimensional Arrays

- To identify a particular two-dimensional multidimensional array element, we must specify the two subscripts; by convention, the first identifies the element's row, and the second identifies the element's column.
- In general, an array with *m* rows and *n* columns is called an *m*-by-*n* array.
- Every element in a two-dimensional array is accessed using an element name of the form `a[ i ][ j ]`; `a` is the name of the array, and `i` and `j` are the subscripts that uniquely identify the row and column, respectively, of each element in `a`.
- Multidimensional arrays are maintained as arrays of arrays.

Section 10.11 Building an Online Quiz

- An XHTML form's elements can be accessed individually using `getElementById` or through the `elements` property of the containing `form` object. The `elements` property contains an array of all the `form`'s controls.
- Property `checked` of a radio button is `true` when the radio button is selected, and it is `false` when the radio button is not selected.

Terminology

`a[i]`
`a[i][j]`
address in memory
array data structure
array initializer list
`Array` object
binary search
bounds of an array

`checked` property of a radio button
collections of data items
column subscript
comma-separated initializer list
comparator function
creating an instance
data structure
declare an array
dynamic memory allocation operator (`new`)
element of an array
`elements` property of a form object
`for...in` repetition statement
index of an element
initialize an array
initializer
initializer list
instantiating an object
iterating over an array's elements
`join` method of an `Array` object
left-hand-side expression
`length` of an `Array` object
linear search of an array
location in an array
m-by-*n* array
multidimensional array
name of an array
`new` operator
off-by-one error
one-dimensional array
pass-by-reference
pass-by-value
passing arrays to functions
place holder in an initializer list (`,`)
position number of an element
reserve a space in an `Array`
row subscript
scalar quantities
separator
search key
searching an array
`sort` method of the `Array` object
sorting an array
square brackets `[]`
subscript
table of values
tabular format
traverse an array
two-dimensional array
value of an element
zeroth element

Self-Review Exercises

10.1 Fill in the blanks in each of the following statements:

a) Lists and tables of values can be stored in ________.
b) The elements of an array are related by the fact that they normally have the same ________.
c) The number used to refer to a particular element of an array is called its ________.
d) The process of putting the elements of an array in order is called ________ the array.
e) Determining whether an array contains a certain key value is called ________ the array.
f) An array that uses two subscripts is referred to as a(n) __________ array.

10.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) An array can store many different types of values.
b) An array subscript should normally be a floating-point value.
c) An individual array element that is passed to a function and modified in it will contain the modified value when the called function completes execution.

10.3 Write JavaScript statements (regarding array `fractions`) to accomplish each of the following tasks:

a) Declare an array with 10 elements, and initialize the elements of the array to `0`.
b) Refer to the fourth element of the array.
c) Refer to array element 4.
d) Assign the value `1.667` to array element 9.
e) Assign the value `3.333` to the seventh element of the array.
f) Sum all the elements of the array, using a `for...in` statement. Define variable `x` as a control variable for the loop.

10.4 Write JavaScript statements (regarding array `table`) to accomplish each of the following tasks:

a) Declare and create the array with three rows and three columns.
b) Display the number of elements.
c) Use a `for…in` statement to initialize each element of the array to the sum of its subscripts. Assume that the variables `x` and `y` are declared as control variables.

10.5 Find the error(s) in each of the following program segments, and correct them.

a)
```
var b = new Array( 10 );
for ( var i = 0; i <= b.length; ++i )
   b[ i ] = 1;
```
b)
```
var a = [ [ 1, 2 ], [ 3, 4 ] ];
a[ 1, 1 ] = 5;
```

Answers to Self-Review Exercises

10.1 a) arrays. b) type. c) subscript. d) sorting. e) searching. f) two-dimensional.

10.2 a) True. b). False. An array subscript must be an integer or an integer expression. c) False. Individual primitive-data-type elements are passed by value. If a reference to an array is passed, then modifications to the elements of the array are reflected in the original element of the array. Also, an individual element of an object type passed to a function is passed by reference, and changes to the object will be reflected in the original array element.

10.3 a) `var fractions = [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ];`
b) `fractions[ 3 ]`
c) `fractions[ 4 ]`
d) `fractions[ 9 ] = 1.667;`
e) `fractions[ 6 ] = 3.333;`
f)
```
var total = 0;
for ( var x in fractions )
   total += fractions[ x ];
```

10.4 a)
```
var table = new Array( new Array( 3 ), new Array( 3 ),
   new Array( 3 ) );
```
b) `document.write( "total: " + ( table.length * table[ 0 ].length ) );`
c)
```
for ( var x in table )
   for ( var y in table[ x ] )
      table[ x ][ y ] = x + y;
```

10.5 a) Error: Referencing an array element outside the bounds of the array (`b[10]`). [*Note:* This error is actually a logic error, not a syntax error.] Correction: Change the `<=` operator to `<`. b) Error: The array subscripting is done incorrectly. Correction: Change the statement to `a[ 1 ][ 1 ] = 5;`.

Exercises

10.6 Fill in the blanks in each of the following statements:

a) JavaScript stores lists of values in ________.
b) The names of the four elements of array `p` are ________, ________, ________ and ________.
c) In a two-dimensional array, the first subscript identifies the ________ of an element, and the second subscript identifies the ________ of an element.
d) An *m*-by-*n* array contains ________ rows, ________ columns and ________ elements.
e) The name the element in row 3 and column 5 of array `d` is ________.

f) The name of the element in the third row and fifth column of array `d` is ________.

10.7 State whether each of the following is *true* or *false*. If *false*, explain why.

a) To refer to a particular location or element in an array, we specify the name of the array and the value of the element.

b) A variable declaration reserves space for an array.

c) To indicate that 100 locations should be reserved for integer array `p`, the programmer should write the declaration

```
p[ 100 ];
```

d) A JavaScript program that initializes the elements of a 15-element array to zero must contain at least one `for` statement.

e) A JavaScript program that totals the elements of a two-dimensional array must contain nested `for` statements.

10.8 Write JavaScript statements to accomplish each of the following tasks:

a) Display the value of the seventh element of array `f`.

b) Initialize each of the five elements of one-dimensional array `g` to `8`.

c) Total the elements of array `c`, which contains 100 numeric elements.

d) Copy 11-element array `a` into the first portion of array `b`, which contains 34 elements.

e) Determine and print the smallest and largest values contained in 99-element floating-point array `w`.

10.9 Consider a two-by-three array `t` that will store integers.

a) Write a statement that declares and creates array `t`.

b) How many rows does `t` have?

c) How many columns does `t` have?

d) How many elements does `t` have?

e) Write the names of all the elements in the second row of `t`.

f) Write the names of all the elements in the third column of `t`.

g) Write a single statement that sets the elements of `t` in row 1 and column 2 to zero.

h) Write a series of statements that initializes each element of `t` to zero. Do not use a repetition structure.

i) Write a nested `for` statement that initializes each element of `t` to zero.

j) Write a series of statements that determines and prints the smallest value in array `t`.

k) Write a statement that displays the elements of the first row of `t`.

l) Write a statement that totals the elements of the fourth column of `t`.

m) Write a series of statements that prints the array `t` in neat, tabular format. List the column subscripts as headings across the top, and list the row subscripts at the left of each row.

10.10 Use a one-dimensional array to solve the following problem: A company pays its salespeople on a commission basis. The salespeople receive $200 per week plus 9 percent of their gross sales for that week. For example, a salesperson who grosses $5000 in sales in a week receives $200 plus 9 percent of $5000, or a total of $650. Write a script (using an array of counters) that obtains the gross sales for each employee through an XHTML form and determines how many of the salespeople earned salaries in each of the following ranges (assume that each salesperson's salary is truncated to an integer amount):

a) $200–299

b) $300–399

c) $400–499

d) $500–599

e) $600–699

f) $700–799

g) $800–899
h) $900–999
i) $1000 and over

10.11 Write statements that perform the following operations for a one-dimensional array:

a) Set the 10 elements of array `counts` to zeros.
b) Add 1 to each of the 15 elements of array `bonus`.
c) Display the five values of array `bestScores`, separated by spaces.

10.12 Use a one-dimensional array to solve the following problem: Read in 20 numbers, each of which is between 10 and 100. As each number is read, print it only if it is not a duplicate of a number that has already been read. Provide for the "worst case," in which all 20 numbers are different. Use the smallest possible array to solve this problem.

10.13 Label the elements of three-by-five two-dimensional array `sales` to indicate the order in which they are set to zero by the following program segment:

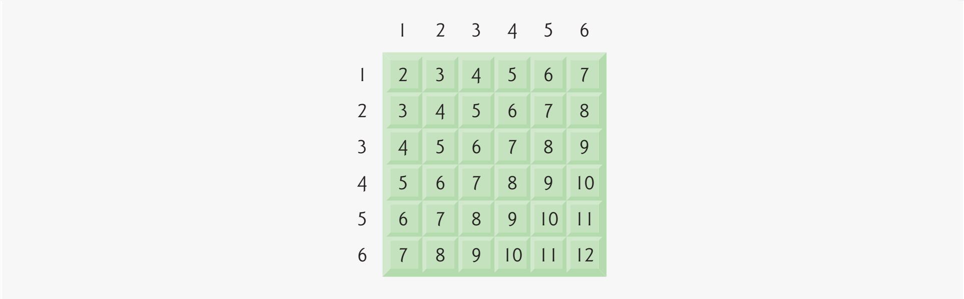
```
for ( var row in sales )
   for ( var col in sales[ row ] )
      sales[ row ][ col ] = 0;
```

10.14 Write a script to simulate the rolling of two dice. The script should use `Math.random` to roll the first die and again to roll the second die. The sum of the two values should then be calculated. [*Note:* Since each die can show an integer value from 1 to 6, the sum of the values will vary from 2 to 12, with 7 being the most frequent sum, and 2 and 12 the least frequent sums. Figure 10.15 shows the 36 possible combinations of the two dice. Your program should roll the dice 36,000 times. Use a one-dimensional array to tally the numbers of times each possible sum appears. Display the results in an XHTML table. Also determine whether the totals are reasonable (e.g., there are six ways to roll a 7, so approximately 1/6 of all the rolls should be 7).]

10.15 Write a script that runs 1000 games of craps and answers the following questions:

a) How many games are won on the first roll, second roll, …, twentieth roll and after the twentieth roll?
b) How many games are lost on the first roll, second roll, …, twentieth roll and after the twentieth roll?
c) What are the chances of winning at craps? [*Note:* You should discover that craps is one of the fairest casino games. What do you suppose this means?]
d) What is the average length of a game of craps?
e) Do the chances of winning improve with the length of the game?

10.16 *(Airline Reservations System)* A small airline has just purchased a computer for its new automated reservations system. You have been asked to program the new system. You are to write a program to assign seats on each flight of the airline's only plane (capacity: 10 seats).



	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Fig. 10.15 | Thirty-six possible outcomes of rolling two dice.

Your program should display the following menu of alternatives: `Please type 1 for "First Class"` and `Please type 2 for "Economy"`. If the person types `1`, your program should assign a seat in the first-class section (seats 1–5). If the person types `2`, your program should assign a seat in the economy section (seats 6–10). Your program should print a boarding pass indicating the person's seat number and whether it is in the first-class or economy section of the plane.

Use a one-dimensional array to represent the seating chart of the plane. Initialize all the elements of the array to `0` to indicate that all the seats are empty. As each seat is assigned, set the corresponding elements of the array to `1` to indicate that the seat is no longer available.

Your program should, of course, never assign a seat that has already been assigned. When the first-class section is full, your program should ask the person if it is acceptable to be placed in the economy section (and vice versa). If yes, then make the appropriate seat assignment. If no, then print the message `"Next flight leaves in 3 hours."`

10.17 Use a two-dimensional array to solve the following problem: A company has four salespeople (1 to 4) who sell five different products (1 to 5). Once a day, each salesperson passes in a slip for each different type of product actually sold. Each slip contains

a) the salesperson number,
b) the product number, and
c) the total dollar value of the product sold that day.

Thus, each salesperson passes in between zero and five sales slips per day. Assume that the information from all of the slips for last month is available. Write a script that will read all this information for last month's sales and summarize the total sales by salesperson by product. All totals should be stored in the two-dimensional array `sales`. After processing all the information for last month, display the results in an XHTML table format, with each of the columns representing a different salesperson and each of the rows representing a different product. Cross-total each row to get the total sales of each product for last month; cross-total each column to get the total sales by salesperson for last month. Your tabular printout should include these cross-totals to the right of the totaled rows and to the bottom of the totaled columns.

10.18 *(Turtle Graphics)* The Logo language, which is popular among young computer users, made the concept of **turtle graphics** famous. Imagine a mechanical turtle that walks around the room under the control of a JavaScript program. The turtle holds a pen in one of two positions, up or down. When the pen is down, the turtle traces out shapes as it moves; when the pen is up, the turtle moves about freely without writing anything. In this problem, you will simulate the operation of the turtle and create a computerized sketchpad as well.

Use a 20-by-20 array `floor` that is initialized to zeros. Read commands from an array that contains them. Keep track of the current position of the turtle at all times and of whether the pen is currently up or down. Assume that the turtle always starts at position (0, 0) of the floor, with its pen up. The set of turtle commands your script must process are as in Fig. 10.16.

Suppose that the turtle is somewhere near the center of the floor. The following "program" would draw and print a 12-by-12 square, then leave the pen in the up position:

```
2
5,12
3
5,12
3
5,12
3
5,12
1
6
9
```

Command	Meaning
`1`	Pen up
`2`	Pen down
`3`	Turn right
`4`	Turn left
`5,10`	Move forward 10 spaces (or a number other than 10)
`6`	Print the 20-by-20 array
`9`	End of data (sentinel)

Fig. 10.16 | Turtle graphics commands.

As the turtle moves with the pen down, set the appropriate elements of array `floor` to `1`s. When the `6` command (print) is given, display an asterisk or some other character of your choosing wherever there is a `1` in the array. Wherever there is a zero, display a blank. Write a script to implement the turtle-graphics capabilities discussed here. Write several turtle-graphics programs to draw interesting shapes. Add other commands to increase the power of your turtle-graphics language.

10.19 *(The Sieve of Eratosthenes)* A prime integer is an integer greater than 1 that is evenly divisible only by itself and 1. The Sieve of Eratosthenes is an algorithm for finding prime numbers. It operates as follows:

a) Create an array with all elements initialized to 1 (true). Array elements with prime subscripts will remain as 1. All other array elements will eventually be set to zero.

b) Set the first two elements to zero, since 0 and 1 are not prime. Starting with array subscript 2, every time an array element is found whose value is 1, loop through the remainder of the array and set to zero every element whose subscript is a multiple of the subscript for the element with value 1. For array subscript 2, all elements beyond 2 in the array that are multiples of 2 will be set to zero (subscripts 4, 6, 8, 10, etc.); for array subscript 3, all elements beyond 3 in the array that are multiples of 3 will be set to zero (subscripts 6, 9, 12, 15, etc.); and so on.

When this process is complete, the array elements that are still set to 1 indicate that the subscript is a prime number. These subscripts can then be printed. Write a script that uses an array of 1000 elements to determine and print the prime numbers between 1 and 999. Ignore element 0 of the array.

10.20 *(Simulation: The Tortoise and the Hare)* In this problem, you will re-create one of the truly great moments in history, namely the classic race of the tortoise and the hare. You will use random number generation to develop a simulation of this memorable event.

Our contenders begin the race at square 1 of 70 squares. Each square represents a possible position along the race course. The finish line is at square 70. The first contender to reach or pass square 70 is rewarded with a pail of fresh carrots and lettuce. The course weaves its way up the side of a slippery mountain, so occasionally the contenders lose ground.

There is a clock that ticks once per second. With each tick of the clock, your script should adjust the position of the animals according to the rules in Fig. 10.17.

Use variables to keep track of the positions of the animals (i.e., position numbers are 1–70). Start each animal at position 1 (i.e., the "starting gate"). If an animal slips left before square 1, move the animal back to square 1.

Animal	Move type	Percentage of the time	Actual move
Tortoise	Fast plod	50%	3 squares to the right
	Slip	20%	6 squares to the left
	Slow plod	30%	1 square to the right
Hare	Sleep	20%	No move at all
	Big hop	20%	9 squares to the right
	Big slip	10%	12 squares to the left
	Small hop	30%	1 square to the right
	Small slip	20%	2 squares to the left

Fig. 10.17 | Rules for adjusting the position of the tortoise and the hare.

Generate the percentages in Fig. 10.17 by producing a random integer i in the range $1 \leq i \leq 10$. For the tortoise, perform a "fast plod" when $1 \leq i \leq 5$, a "slip" when $6 \leq i \leq 7$ and a "slow plod" when $8 \leq i \leq 10$. Use a similar technique to move the hare.

Begin the race by printing

```
BANG !!!!!
AND THEY'RE OFF !!!!!
```

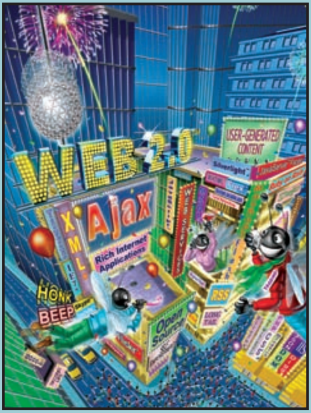
Then, for each tick of the clock (i.e., each repetition of a loop), print a 70-position line showing the letter `T` in the position of the tortoise and the letter `H` in the position of the hare. Occasionally, the contenders will land on the same square. In this case, the tortoise bites the hare, and your script should print `OUCH!!!` beginning at that position. All print positions other than the `T`, the `H` or the `OUCH!!!` (in case of a tie) should be blank.

After each line is printed, test whether either animal has reached or passed square 70. If so, print the winner, and terminate the simulation. If the tortoise wins, print `TORTOISE WINS!!! YAY!!!` If the hare wins, print `Hare wins. Yuck!` If both animals win on the same tick of the clock, you may want to favor the turtle (the "underdog"), or you may want to print `It's a tie`. If neither animal wins, perform the loop again to simulate the next tick of the clock. When you are ready to run your script, assemble a group of fans to watch the race. You'll be amazed at how involved your audience gets!

Later in the book, we introduce a number of Dynamic HTML capabilities, such as graphics, images, animation and sound. As you study those features, you might enjoy enhancing your tortoise-and-hare contest simulation.

11

JavaScript: Objects



My object all sublime
I shall achieve in time.

—W. S. Gilbert

Is it a world to hide virtues in?

—William Shakespeare

Good as it is to inherit a library, it is better to collect one.

—Augustine Birrell

A philosopher of imposing stature doesn't think in a vacuum. Even his most abstract ideas are, to some extent, conditioned by what is or is not known in the time when he lives.

—Alfred North Whitehead

OBJECTIVES

In this chapter you will learn:

- Object-based programming terminology and concepts.
- The concepts of encapsulation and data hiding.
- The value of object orientation.
- To use the JavaScript objects `Math`, `String`, `Date`, `Boolean` and `Number`.
- To use the browser's `document` and `window` objects.
- To use cookies.
- To represent objects simply using JSON.

Outline

11.1 Introduction

Most of the JavaScript programs we've demonstrated illustrate basic programming concepts. These programs provide you with the foundation you need to build powerful and complex scripts as part of your web pages. As you proceed beyond this chapter, you will use JavaScript to manipulate every element of an XHTML document from a script.

This chapter presents a more formal treatment of objects. We begin by giving a brief introduction to the concepts behind object-orientation. The remainder of the chapter overviews—and serves as a reference for—several of JavaScript's built-in objects and demonstrates many of their capabilities. We also provide a brief introduction to JSON, a means for creating JavaScript objects. In the chapters on the Document Object Model and Events that follow this chapter, you will be introduced to many objects provided by the browser that enable scripts to interact with the elements of an XHTML document.

11.2 Introduction to Object Technology

This section provides a general introduction to object orientation. The terminology and technologies discussed here support various chapters that come later in the book. Here, you'll learn that objects are a natural way of thinking about the world and about scripts that manipulate XHTML documents. In Chapters 6–10, we used built-in JavaScript objects—`Math` and `Array`—and objects provided by the web browser—`document` and `win-`

`dow`—to perform tasks in our scripts. JavaScript uses objects to perform many tasks and therefore is referred to as an **object-based programming language**. As we have seen, JavaScript also uses constructs from the "conventional" structured programming methodology supported by many other programming languages. The first five JavaScript chapters concentrated on these conventional parts of JavaScript because they are important components of all JavaScript programs. Our goal here is to help you develop an object-oriented way of thinking. Many concepts in this book, including CSS, JavaScript, Ajax, Ruby on Rails, ASP.NET, and JavaServer Faces are based on at least some of the concepts introduced in this section.

Basic Object-Technology Concepts

We begin our introduction to object technology with some key terminology. Everywhere you look in the real world you see objects—people, animals, plants, cars, planes, buildings, computers, monitors and so on. Humans think in terms of objects. Telephones, houses, traffic lights, microwave ovens and water coolers are just a few more objects we see around us every day.

We sometimes divide objects into two categories: animate and inanimate. Animate objects are "alive" in some sense—they move around and do things. Inanimate objects do not move on their own. Objects of both types, however, have some things in common. They all have **attributes** (e.g., size, shape, color and weight), and they all exhibit **behaviors** (e.g., a ball rolls, bounces, inflates and deflates; a baby cries, sleeps, crawls, walks and blinks; a car accelerates, brakes and turns; a towel absorbs water). We'll study the kinds of attributes and behaviors that software objects have.

Humans learn about existing objects by studying their attributes and observing their behaviors. Different objects can have similar attributes and can exhibit similar behaviors. Comparisons can be made, for example, between babies and adults, and between humans and chimpanzees.

Object-oriented design (OOD) models software in terms similar to those that people use to describe real-world objects. It takes advantage of class relationships, where objects of a certain class, such as a class of vehicles, have the same characteristics—cars, trucks, little red wagons and roller skates have much in common. OOD takes advantage of **inheritance** relationships, where new classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own. An object of class "convertible" certainly has the characteristics of the more general class "automobile," but more specifically, the roof goes up and down.

Object-oriented design provides a natural and intuitive way to view the software design process—namely, modeling objects by their attributes, behaviors and interrelationships just as we describe real-world objects. OOD also models communication between objects. Just as people send messages to one another (e.g., a sergeant commands a soldier to stand at attention), objects also communicate via messages. A bank account object may receive a message to decrease its balance by a certain amount because the customer has withdrawn that amount of money.

OOD **encapsulates** (i.e., wraps) attributes and **operations** (behaviors) into objects—an object's attributes and operations are intimately tied together. Objects have the property of **information hiding**. This means that objects may know how to communicate with one another across well-defined **interfaces**, but normally they are not allowed to know how other objects are implemented—implementation details are hidden within the objects themselves. We can drive a car effectively, for instance, without knowing the details of how engines, transmissions, brakes and exhaust systems work internally—as long as we know how to use the accelerator pedal, the brake pedal, the steering wheel and so on. Information hiding, as we'll see, is crucial to good software engineering.

Like the designers of an automobile, the designers of web browsers have defined a set of objects that encapsulate an XHTML document's elements and expose to a JavaScript programmer the attributes and behaviors that enable a JavaScript program to interact with (or script) those elements (objects). You'll soon see that the browser's `document` object contains attributes and behaviors that provide access to every element of an XHTML document. Similarly, JavaScript provides objects that encapsulate various capabilities in a script. For example, the JavaScript `Array` object provides attributes and behaviors that enable a script to manipulate a collection of data. The `Array` object's `length` property (attribute) contains the number of elements in the `Array`. The `Array` object's `sort` method (behavior) orders the elements of the `Array`.

Some programming languages—like Java, Visual Basic, C# and C++—are **object oriented**. Programming in such a language is called **object-oriented programming (OOP),** and it allows computer programmers to implement object-oriented designs as working software systems. Languages like C, on the other hand, are **procedural**, so programming tends to be **action oriented**. In procedural languages, the unit of programming is the **function.** In object-oriented languages, the unit of programming is the **class** from which objects are eventually **instantiated** (an OOP term for "created"). Classes contain functions that implement operations and data that comprises attributes.

Procedural programmers concentrate on writing functions. Programmers group actions that perform some common task into functions, and group functions to form programs. Data is certainly important in procedural languages, but the view is that data exists primarily in support of the actions that functions perform. The **verbs** in a system specification help a procedural programmer determine the set of functions that work together to implement the system.

Classes, Properties and Methods

Object-oriented programmers concentrate on creating their own **user-defined types** called **classes**. Each class contains data as well as the set of functions that manipulate that data and provide services to **clients** (i.e., other classes or functions that use the class). The data components of a class are called properties. For example, a bank account class might include an account number and a balance. The function components of a class are called methods. For example, a bank account class might include methods to make a deposit (increasing the balance), make a withdrawal (decreasing the balance) and inquire what the current balance is. You use built-in types (and other user-defined types) as the "building blocks" for constructing new user-defined types (classes). The **nouns** in a system specification help you determine the set of classes from which objects are created that work together to implement the system.

Classes are to objects as blueprints are to houses—a class is a "plan" for building an object of the class. Just as we can build many houses from one blueprint, we can instantiate (create) many objects from one class. You cannot cook meals in the kitchen of a blueprint; you can cook meals in the kitchen of a house. You cannot sleep in the bedroom of a blueprint; you can sleep in the bedroom of a house.

Classes can have relationships with other classes. For example, in an object-oriented design of a bank, the “bank teller” class needs to relate to other classes, such as the “customer” class, the “cash drawer” class, the “safe” class, and so on. These relationships are called **associations**.

Packaging software as classes makes it possible for future software systems to **reuse** the classes. Groups of related classes are often packaged as reusable **components**. Just as realtors often say that the three most important factors affecting the price of real estate are “location, location and location,” some people in the software development community say that the three most important factors affecting the future of software development are “reuse, reuse and reuse.

Indeed, with object technology, you can build much of the new software you’ll need by combining existing classes, just as automobile manufacturers combine interchangeable parts. Each new class you create will have the potential to become a valuable software asset that you and other programmers can reuse to speed and enhance the quality of future software development efforts. Now that we’ve introduced the terminology associated with object-orientation, you’ll see it used in the upcoming discussions of some of JavaScript’s objects.

11.3 Math Object

The Math object’s methods allow you to perform many common mathematical calculations. As shown previously, an object’s methods are called by writing the name of the object followed by a dot (.) and the name of the method. In parentheses following the method name is the argument (or a comma-separated list of arguments) to the method. For example, to calculate and display the square root of 900.0 you might write

```
document.writeln( Math.sqrt( 900.0 ) );
```

which calls method Math.sqrt to calculate the square root of the number contained in the parentheses (900.0), then outputs the result. The number 900.0 is the argument of the Math.sqrt method. The preceding statement would display 30.0. Some **Math** object methods are summarized in Fig. 11.1.

Method	Description	Examples
abs(x)	absolute value of x	abs(7.2) is 7.2 abs(0.0) is 0.0 abs(-5.6) is 5.6
ceil(x)	rounds x to the smallest integer not less than x	ceil(9.2) is 10.0 ceil(-9.8) is -9.0
cos(x)	trigonometric cosine of x (x in radians)	cos(0.0) is 1.0
exp(x)	exponential method e^x	exp(1.0) is 2.71828 exp(2.0) is 7.38906

Fig. 11.1 | Math object methods. (Part 1 of 2.)

Method	Description	Examples

`floor( x )`	rounds x to the largest integer not greater than x	`floor( 9.2 )` is `9.0`
		`floor( -9.8 )` is `-10.0`

`log( x )`	natural logarithm of x (base *e*)	`log( 2.718282 )` is `1.0`
		`log( 7.389056 )` is `2.0`

`max( x, y )`	larger value of x and y	`max( 2.3, 12.7 )` is `12.7`
		`max( -2.3, -12.7 )` is `-2.3`

`min( x, y )`	smaller value of x and y	`min( 2.3, 12.7 )` is `2.3`
		`min( -2.3, -12.7 )` is `-12.7`

`pow( x, y )`	x raised to power y (x^y)	`pow( 2.0, 7.0 )` is `128.0`
		`pow( 9.0, .5 )` is `3.0`

`round( x )`	rounds x to the closest integer	`round( 9.75 )` is `10`
		`round( 9.25 )` is `9`

`sin( x )`	trigonometric sine of x (x in radians)	`sin( 0.0 )` is `0.0`

`sqrt( x )`	square root of x	`sqrt( 900.0 )` is `30.0`
		`sqrt( 9.0 )` is `3.0`

`tan( x )`	trigonometric tangent of x (x in radians)	`tan( 0.0 )` is `0.0`

Fig. 11.1 | `Math` object methods. (Part 2 of 2.)

Common Programming Error 11.1

Forgetting to invoke a `Math` *method by preceding the method name with the object name* `Math` *and a dot (*`.`*) is an error.*

Software Engineering Observation 11.1

The primary difference between invoking a standalone function and invoking a method of an object is that an object name and a dot are not required to call a standalone function.

The `Math` object defines several commonly used mathematical constants, summarized in Fig. 11.2. [*Note:* By convention, the names of constants are written in all uppercase letters so they stand out in a program.]

Constant	Description	Value
`Math.E`	Base of a natural logarithm (*e*).	Approximately 2.718
`Math.LN2`	Natural logarithm of 2	Approximately 0.693
`Math.LN10`	Natural logarithm of 10	Approximately 2.302

Fig. 11.2 | Properties of the `Math` object. (Part 1 of 2.)

Constant	Description	Value
`Math.LOG2E`	Base 2 logarithm of *e*	Approximately 1.442
`Math.LOG10E`	Base 10 logarithm of *e*	Approximately 0.434
`Math.PI`	π—the ratio of a circle's circumference to its diameter	Approximately 3.141592653589793
`Math.SQRT1_2`	Square root of 0.5	Approximately 0.707
`Math.SQRT2`	Square root of 2.0	Approximately 1.414

Fig. 11.2 | Properties of the `Math` object. (Part 2 of 2.)

Good Programming Practice 11.1

Use the mathematical constants of the `Math` object rather than explicitly typing the numeric value of the constant.

11.4 String Object

In this section, we introduce JavaScript's string- and character-processing capabilities. The techniques discussed here are appropriate for processing names, addresses, telephone numbers, and similar items.

11.4.1 Fundamentals of Characters and Strings

Characters are the fundamental building blocks of JavaScript programs. Every program is composed of a sequence of characters grouped together meaningfully that is interpreted by the computer as a series of instructions used to accomplish a task.

A string is a series of characters treated as a single unit. A string may include letters, digits and various **special characters**, such as +, -, *, /, and $. JavaScript supports the set of characters called **Unicode®**, which represents a large portion of the world's languages. (We discuss Unicode in detail in Appendix F.) A string is an object of type **`String`**. **String literals** or **string constants** (often called **anonymous `String` objects**) are written as a sequence of characters in double quotation marks or single quotation marks, as follows:

```
"John Q. Doe"                  (a name)
'9999 Main Street'             (a street address)
"Waltham, Massachusetts"       (a city and state)
'(201) 555-1212'               (a telephone number)
```

A `String` may be assigned to a variable in a declaration. The declaration

```
var color = "blue";
```

initializes variable `color` with the `String` object containing the string `"blue"`. `String`s can be compared via the relational (<, <=, > and >=) and equality operators (== and !=). Strings are compared using the Unicode values of the corresponding characters. For example, the expression `"hello"` < `"Hello"` evaluates to false because lowercase letters have higher Unicode values.

11.4.2 Methods of the `String` Object

The `String` object encapsulates the attributes and behaviors of a string of characters. It provides many methods (behaviors) that accomplish useful tasks such as selecting characters from a string, combining strings (called **concatenation**), obtaining substrings of a string, searching for substrings within a string, tokenizing strings (i.e., splitting strings into individual words) and converting strings to all uppercase or lowercase letters. The `String` object also provides several methods that generate XHTML tags. Figure 11.3 summarizes many `String` methods. Figures 11.4–11.7 demonstrate some of these methods.

Method	Description
`charAt(` *index* `)`	Returns a string containing the character at the specified *index*. If there is no character at the *index*, `charAt` returns an empty string. The first character is located at *index* 0.
`charCodeAt(` *index* `)`	Returns the Unicode value of the character at the specified *index*, or `NaN` (not a number) if there is no character at that *index*.
`concat(` *string* `)`	Concatenates its argument to the end of the string that invokes the method. The string invoking this method is not modified; instead a new `String` is returned. This method is the same as adding two strings with the string-concatenation operator + (e.g., `s1.concat(s2)` is the same as `s1 + s2`).
`fromCharCode(` *value1*`,` *value2*`, …)`	Converts a list of Unicode values into a string containing the corresponding characters.
`indexOf(` *substring*`,` *index* `)`	Searches for the first occurrence of *substring* starting from position *index* in the string that invokes the method. The method returns the starting index of *substring* in the source string or –1 if *substring* is not found. If the *index* argument is not provided, the method begins searching from index 0 in the source string.
`lastIndexOf(` *substring*`,` *index* `)`	Searches for the last occurrence of *substring* starting from position *index* and searching toward the beginning of the string that invokes the method. The method returns the starting index of *substring* in the source string or –1 if *substring* is not found. If the *index* argument is not provided, the method begins searching from the end of the source string.
`replace(` *searchString*`,` *replaceString* `)`	Searches for the substring *searchString*, and replaces the first occurrence with *replaceString* and returns the modified string, or the original string if no replacement was made.
`slice(` *start*`,` *end* `)`	Returns a string containing the portion of the string from index *start* through index *end*. If the *end* index is not specified, the method returns a string from the *start* index to the end of the source string. A negative *end* index specifies an offset from the end of the string, starting from a position one past the end of the last character (so –1 indicates the last character position in the string).

Fig. 11.3 | Some `String` object methods. (Part 1 of 2.)

Method	Description
`split(` *string* `)`	Splits the source string into an array of strings (tokens), where its *string* argument specifies the delimiter (i.e., the characters that indicate the end of each token in the source string).
`substr(` *start*`,` *length* `)`	Returns a string containing *length* characters starting from index *start* in the source string. If *length* is not specified, a string containing characters from *start* to the end of the source string is returned.
`substring(` *start*`,` *end* `)`	Returns a string containing the characters from index *start* up to but not including index *end* in the source string.
`toLowerCase()`	Returns a string in which all uppercase letters are converted to lowercase letters. Nonletter characters are not changed.
`toUpperCase()`	Returns a string in which all lowercase letters are converted to uppercase letters. Nonletter characters are not changed.
Methods that generate XHTML tags	
`anchor(` *name* `)`	Wraps the source string in an anchor element (`<a></a>`) with *name* as the anchor name.
`fixed()`	Wraps the source string in a `<tt></tt>` element (same as `<pre></pre>`).
`link(` *url* `)`	Wraps the source string in an anchor element (`<a></a>`) with *url* as the hyperlink location.
`strike()`	Wraps the source string in a `<strike></strike>` element.
`sub()`	Wraps the source string in a `<sub></sub>` element.
`sup()`	Wraps the source string in a `<sup></sup>` element.

Fig. 11.3 | Some `String` object methods. (Part 2 of 2.)

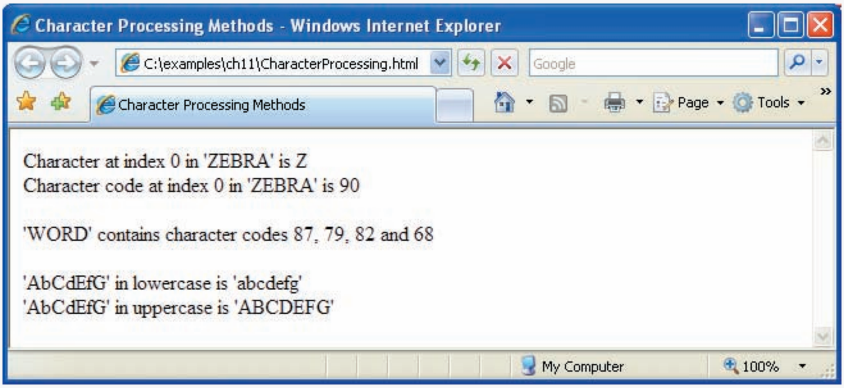
11.4.3 Character-Processing Methods

The script in Fig. 11.4 demonstrates some of the `String` object's character-processing methods, including **`charAt`** (returns the character at a specific position), **`charCodeAt`** (returns the Unicode value of the character at a specific position), **`fromCharCode`** (returns a string created from a series of Unicode values), **`toLowerCase`** (returns the lowercase version of a string) and **`toUpperCase`** (returns the uppercase version of a string).

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 11.4: CharacterProcessing.html -->
<!-- String methods charAt, charCodeAt, fromCharCode, toLowercase and
   toUpperCase. -->
```

Fig. 11.4 | String methods `charAt`, `charCodeAt`, `fromCharCode`, `toLowercase` and `toUpperCase`. (Part 1 of 2.)

```
 8  <html xmlns = "http://www.w3.org/1999/xhtml">
 9     <head>
10        <title>Character Processing Methods</title>
11        <script type = "text/javascript">
12           <!--
13           var s = "ZEBRA";
14           var s2 = "AbCdEfG";
15
16           document.writeln( "<p>Character at index 0 in '" +
17              s + "' is " + s.charAt( 0 ) );
18           document.writeln( "<br />Character code at index 0 in '"
19              + s + "' is " + s.charCodeAt( 0 ) + "</p>" );
20
21           document.writeln( "<p>'" +
22              String.fromCharCode( 87, 79, 82, 68 ) +
23              "' contains character codes 87, 79, 82 and 68</p>" )
24
25           document.writeln( "<p>'" + s2 + "' in lowercase is '" +
26              s2.toLowerCase() + "'" );
27           document.writeln( "<br />'" + s2 + "' in uppercase is '"
28              + s2.toUpperCase() + "'</p>" );
29            // -->
30        </script>
31     </head><body></body>
32  </html>
```


Character Processing Methods - Windows Internet Explorer
C:\examples\ch11\CharacterProcessing.html
Google
Character Processing Methods
Page
Tools
My Computer
100%

Character at index 0 in 'ZEBRA' is Z
Character code at index 0 in 'ZEBRA' is 90

'WORD' contains character codes 87, 79, 82 and 68

'AbCdEfG' in lowercase is 'abcdefg'
'AbCdEfG' in uppercase is 'ABCDEFG'

Fig. 11.4 | String methods `charAt`, `charCodeAt`, `fromCharCode`, `toLowercase` and `toUpperCase`. (Part 2 of 2.)

Lines 16–17 display the first character in `String` `s` (`"ZEBRA"`) using `String` method `charAt`. Method **`charAt`** returns a string containing the character at the specified index (0 in this example). Indices for the characters in a string start at 0 (the first character) and go up to (but do not include) the string's `length` (i.e., if the string contains five characters, the indices are 0 through 4). If the index is outside the bounds of the string, the method returns an empty string.

Lines 18–19 display the character code for the first character in `String` `s` (`"ZEBRA"`) by calling `String` method `charCodeAt`. Method `charCodeAt` returns the Unicode value of the character at the specified index (0 in this example). If the index is outside the bounds of the string, the method returns NaN.

String method fromCharCode receives as its argument a comma-separated list of Unicode values and builds a string containing the character representation of those Unicode values. Lines 21–23 display the string "WORD", which consists of the character codes 87, 79, 82 and 68. Note that the String object calls method fromCharCode, rather than a specific String variable. Appendix D, ASCII Character Set, contains the character codes for the ASCII character set—a subset of the Unicode character set (Appendix F) that contains only Western characters.

The statements in lines 25–26 and 27–28 use String methods **toLowerCase** and **toUpperCase** to display versions of String s2 ("AbCdEfG") in all lowercase letters and all uppercase letters, respectively.

11.4.4 Searching Methods

Being able to search for a character or a sequence of characters in a string is often useful. For example, if you are creating your own word processor, you may want to provide a capability for searching through the document. The script in Fig. 11.5 demonstrates the String object methods **indexOf** and **lastIndexOf** that search for a specified substring in a string. All the searches in this example are performed on the global string letters (initialized in line 14 with "abcdefghijklmnopqrstuvwxyzabcdefghijklm" in the script).

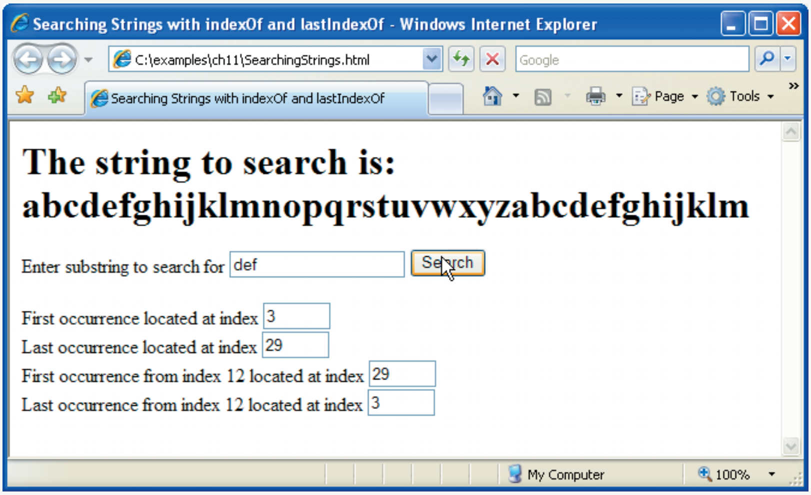
The user types a substring in the XHTML form searchForm's inputVal text field and presses the **Search** button to search for the substring in letters. Clicking the **Search** button calls function buttonPressed (defined in lines 16–29) to respond to the onclick event and perform the searches. The results of each search are displayed in the appropriate text field of searchForm.

Lines 21–22 use String method indexOf to determine the location of the first occurrence in string letters of the string inputVal.value (i.e., the string the user typed in the

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 11.5: SearchingStrings.html -->
 6  <!-- String searching with indexOf and lastIndexOf. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>
10           Searching Strings with indexOf and lastIndexOf
11        </title>
12        <script type = "text/javascript">
13           <!--
14           var letters = "abcdefghijklmnopqrstuvwxyzabcdefghijklm";
15
16           function buttonPressed()
17           {
18              var searchForm = document.getElementById( "searchForm" );
19              var inputVal = document.getElementById( "inputVal" );
```

Fig. 11.5 | String searching with indexOf and lastIndexOf. (Part 1 of 3.)

```

            searchForm.elements[2].value =
               letters.indexOf( inputVal.value );
            searchForm.elements[3].value =
               letters.lastIndexOf( inputVal.value );
            searchForm.elements[4].value =
               letters.indexOf( inputVal.value, 12 );
            searchForm.elements[5].value =
               letters.lastIndexOf( inputVal.value, 12 );
         } // end function buttonPressed
         // -->
      </script>
   </head>
   <body>
      <form id = "searchForm" action = "">
         <h1>The string to search is:<br />
             abcdefghijklmnopqrstuvwxyzabcdefghijklm</h1>
         <p>Enter substring to search for
         <input id = "inputVal" type = "text" />
         <input id = "search" type = "button" value = "Search"
                onclick = "buttonPressed()" /><br /></p>

         <p>First occurrence located at index
         <input id = "first" type = "text" size = "5" />
         <br />Last occurrence located at index
         <input id = "last" type = "text" size = "5" />
         <br />First occurrence from index 12 located at index
         <input id = "first12" type = "text" size = "5" />
         <br />Last occurrence from index 12 located at index
         <input id = "last12" type = "text" size = "5" /></p>
      </form>
   </body>
</html>
```



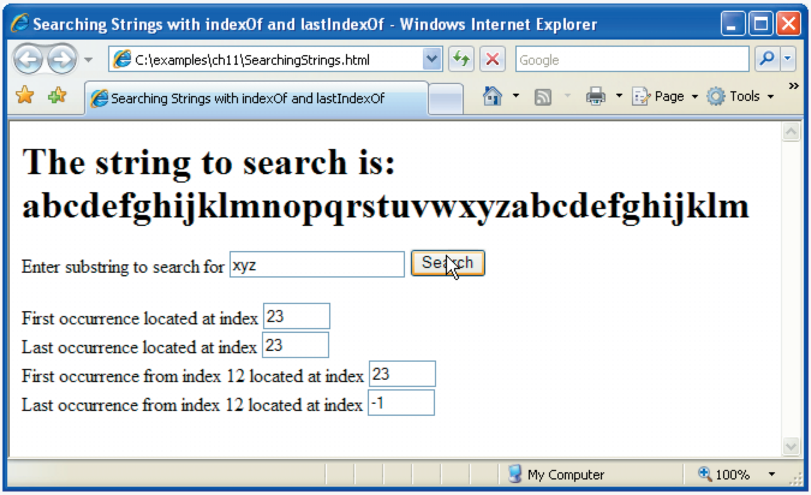
Searching Strings with indexOf and lastIndexOf - Windows Internet Explorer
C:\examples\ch11\SearchingStrings.html
Google
Searching Strings with indexOf and lastIndexOf
Page
Tools
My Computer
100%

The string to search is: abcdefghijklmnopqrstuvwxyzabcdefghijklm

Enter substring to search for def [Search]

First occurrence located at index 3
Last occurrence located at index 29
First occurrence from index 12 located at index 29
Last occurrence from index 12 located at index 3

Fig. 11.5 | `String` searching with `indexOf` and `lastIndexOf`. (Part 2 of 3.)


Searching Strings with indexOf and lastIndexOf - Windows Internet Explorer
C:\examples\ch11\SearchingStrings.html
Google
Searching Strings with indexOf and lastIndexOf
Page
Tools
My Computer
100%

The string to search is: abcdefghijklmnopqrstuvwxyzabcdefghijklm

Enter substring to search for xyz Search

First occurrence located at index 23
Last occurrence located at index 23
First occurrence from index 12 located at index 23
Last occurrence from index 12 located at index -1

Fig. 11.5 | `String` searching with `indexOf` and `lastIndexOf`. (Part 3 of 3.)

`inputVal` text field). If the substring is found, the index at which the first occurrence of the substring begins is returned; otherwise, –1 is returned.

Lines 23–24 use `String` method `lastIndexOf` to determine the location of the last occurrence in `letters` of the string in `inputVal`. If the substring is found, the index at which the last occurrence of the substring begins is returned; otherwise, –1 is returned.

Lines 25–26 use `String` method `indexOf` to determine the location of the first occurrence in string `letters` of the string in the `inputVal` text field, starting from index `12` in `letters`. If the substring is found, the index at which the first occurrence of the substring (starting from index `12`) begins is returned; otherwise, –1 is returned.

Lines 27–28 use `String` method `lastIndexOf` to determine the location of the last occurrence in `letters` of the string in the `inputVal` text field, starting from index `12` in `letters` and moving toward the beginning of the input. If the substring is found, the index at which the first occurrence of the substring (if one appears before index `12`) begins is returned; otherwise, –1 is returned.

Software Engineering Observation 11.2

`String` methods `indexOf` and `lastIndexOf`, with their optional second argument (the starting index from which to search), are particularly useful for continuing a search through a large amount of text.

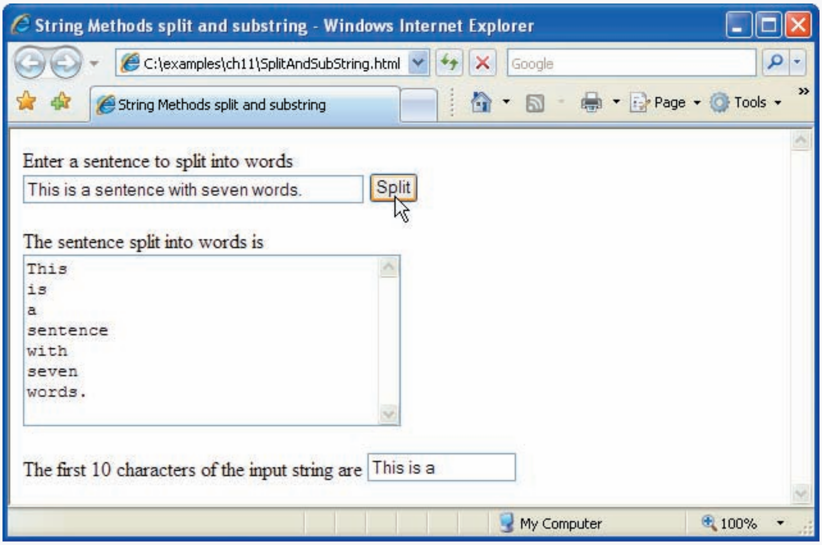
11.4.5 Splitting Strings and Obtaining Substrings

When you read a sentence, your mind breaks it into individual words, or **tokens**, each of which conveys meaning to you. The process of breaking a string into tokens is called **tokenization**. Interpreters also perform tokenization. They break up statements into such individual pieces as keywords, identifiers, operators and other elements of a programming language. Figure 11.6 demonstrates `String` method **`split`**, which breaks a string into its component tokens. Tokens are separated from one another by **delimiters**, typically white-space characters such as blanks, tabs, newlines and carriage returns. Other characters may also be used as delimiters to separate tokens. The XHTML document displays a form containing a text field where the user types a sentence to tokenize. The results of the tokenization process are displayed in an XHTML `textarea` GUI component. The script also demonstrates `String` method **`substring`**, which returns a portion of a string.

The user types a sentence into the text field with `id inputVal` text field and presses the **Split** button to tokenize the string. Function `splitButtonPressed` (lines 12–21) handles the button's `onclick` event.

```
1  <?xml version = "1.0" encoding = "utf-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 11.6: SplitAndSubString.html -->
6  <!-- String object methods split and substring. -->
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8     <head>
9        <title>String Methods split and substring</title>
10       <script type = "text/javascript">
11          <!--
12          function splitButtonPressed()
13          {
14             var inputString = document.getElementById( "inputVal" ).value;
15             var tokens = inputString.split( " " );
16             document.getElementById( "output" ).value =
17                tokens.join( "\n" ) ;
18
19             document.getElementById( "outputSubstring" ).value =
20                inputString.substring( 0, 10 );
21          } // end function splitButtonPressed
22          // -->
23       </script>
24    </head>
25    <body>
26       <form action = "">
27          <p>Enter a sentence to split into words<br />
28          <input id = "inputVal" type = "text" size = "40" />
29          <input type = "button" value = "Split"
30             onclick = "splitButtonPressed()" /></p>
31
32          <p>The sentence split into words is<br />
33          <textarea id = "output" rows = "8" cols = "34">
34          </textarea></p>
35
36          <p>The first 10 characters of the input string are
37          <input id = "outputSubstring" type = "text"
38                size = "15" /></p>
39       </form>
40    </body>
41 </html>
```

Fig. 11.6 | `String` object methods `split` and `substring`. (Part 1 of 2.)


String Methods split and substring - Windows Internet Explorer
C:\examples\ch11\SplitAndSubString.html
Google
String Methods split and substring
Page
Tools
Split
My Computer
100%

Enter a sentence to split into words

This is a sentence with seven words.

The sentence split into words is

```
This
is
a
sentence
with
seven
words.
```

The first 10 characters of the input string are This is a

Fig. 11.6 | `String` object methods `split` and `substring`. (Part 2 of 2.)

Line 14 gets the value of the input field and stores it in variable `inputString`. Line 15 calls `String` method `split` to tokenize `inputString`. The argument to method `split` is the **delimiter string**—the string that determines the end of each token in the original string. In this example, the space character delimits the tokens. The delimiter string can contain multiple characters that should be used as delimiters. Method `split` returns an array of strings containing the tokens. Line 17 uses `Array` method `join` to combine the tokens in array `tokens` and separate each token with a newline character (`\n`). The resulting string is assigned to the `value` property of the XHTML form's `output` GUI component (an XHTML `textarea`).

Lines 19–20 use `String` method `substring` to obtain a string containing the first 10 characters of the string the user entered (still stored in `inputString`). The method returns the substring from the **starting index** (`0` in this example) up to but not including the **ending index** (`10` in this example). If the ending index is greater than the length of the string, the substring returned includes the characters from the starting index to the end of the original string.

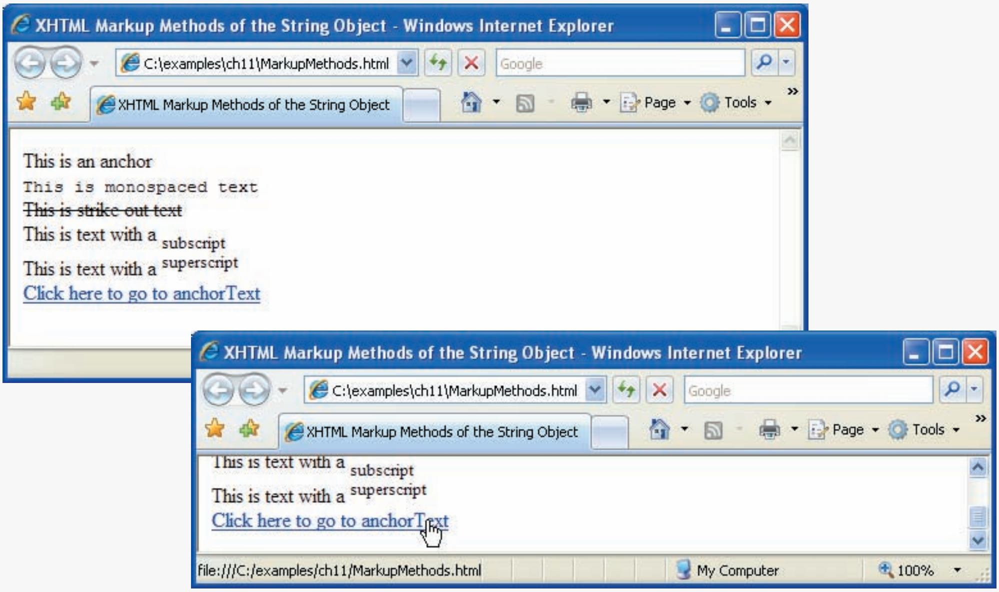
11.4.6 XHTML Markup Methods

The script in Fig. 11.7 demonstrates the `String` object's methods that generate XHTML markup tags. When a `String` object invokes a markup method, the method wraps the `String`'s contents in the appropriate XHTML tag. These methods are particularly useful for generating XHTML dynamically during script processing.

Lines 12–17 define the strings that call each of the XHTML markup methods of the `String` object. Line 19 uses `String` method **`anchor`** to format the string in variable `anchorText` (`"This is an anchor"`) as

```
<a name = "top">This is an anchor</a>
```

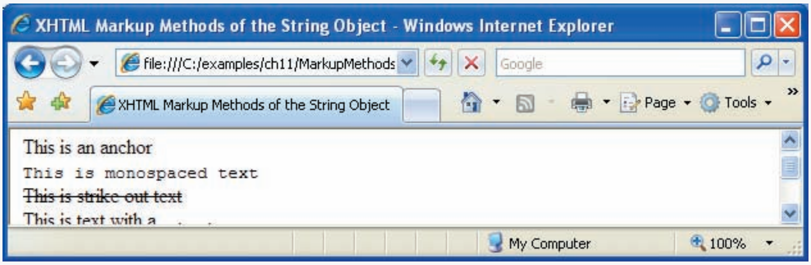
```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 11.7: MarkupMethods.html -->
<!-- String object XHTML markup methods. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>XHTML Markup Methods of the String Object</title>
      <script type = "text/javascript">
         <!--
         var anchorText = "This is an anchor";
         var fixedText = "This is monospaced text";
         var linkText = "Click here to go to anchorText";
         var strikeText = "This is strike out text";
         var subText = "subscript";
         var supText = "superscript";

         document.writeln( anchorText.anchor( "top" ) );
         document.writeln( "<br />" + fixedText.fixed() );
         document.writeln( "<br />" + strikeText.strike() );
         document.writeln(
            "<br />This is text with a " + subText.sub() );
         document.writeln(
            "<br />This is text with a " + supText.sup() );
         document.writeln( "<br />" + linkText.link( "#top" ) );
         // -->
      </script>
   </head><body></body>
</html>
```


XHTML Markup Methods of the String Object - Windows Internet Explorer
C:\examples\ch11\MarkupMethods.html
Google
XHTML Markup Methods of the String Object
Page
Tools
file:///C:/examples/ch11/MarkupMethods.html
My Computer
100%

This is an anchor
This is monospaced text
~~This is strike out text~~
This is text with a $_{\text{subscript}}$
This is text with a $^{\text{superscript}}$
Click here to go to anchorText

This is text with a $_{\text{subscript}}$
This is text with a $^{\text{superscript}}$
Click here to go to anchorText

Fig. 11.7 | String object XHTML markup methods. (Part 1 of 2.)


XHTML Markup Methods of the String Object - Windows Internet Explorer
file:///C:/examples/ch11/MarkupMethods
Google
XHTML Markup Methods of the String Object
Page
Tools
My Computer
100%

This is an anchor

This is monospaced text

~~This is strike out text~~

This is text with a

Fig. 11.7 | String object XHTML markup methods. (Part 2 of 2.)

The name of the anchor is the argument to the method. This anchor will be used later in the example as the target of a hyperlink.

Line 20 uses String method **fixed** to display text in a fixed-width font by formatting the string in variable fixedText ("This is monospaced text") as

```
<tt>This is monospaced text</tt>
```

Line 21 uses String method **strike** to display text with a line through it by formatting the string in variable strikeText ("This is strike out text") as

```
<strike>This is strike out text</strike>
```

Lines 22–23 use String method **sub** to display subscript text by formatting the string in variable subText ("subscript") as

```
<sub>subscript</sub>
```

Note that the resulting line in the XHTML document displays the word subscript smaller than the rest of the line and slightly below the line.

Lines 24–25 call String method **sup** to display superscript text by formatting the string in variable supText ("superscript") as

```
<sup>superscript</sup>
```

Note that the resulting line in the XHTML document displays the word superscript smaller than the rest of the line and slightly above the line.

Line 26 uses String method **link** to create a hyperlink by formatting the string in variable linkText ("Click here to go to anchorText") as

```
<a href = "#top">Click here to go to anchorText</a>
```

The target of the hyperlink (#top in this example) is the argument to the method and can be any URL. In this example, the hyperlink target is the anchor created in line 19. If you make your browser window short and scroll to the bottom of the web page, then click this link, the browser will reposition to the top of the web page.

11.5 Date Object

JavaScript's **Date** object provides methods for date and time manipulations. Date and time processing can be performed based on the computer's **local time zone** or based on World Time Standard's **Coordinated Universal Time** (abbreviated **UTC**)—formerly called

Greenwich Mean Time (GMT). Most methods of the Date object have a local time zone and a UTC version. The methods of the Date object are summarized in Fig. 11.8.

Method	Description

getDate()
getUTCDate()
	Returns a number from 1 to 31 representing the day of the month in local time or UTC.

getDay()
getUTCDay()
	Returns a number from 0 (Sunday) to 6 (Saturday) representing the day of the week in local time or UTC.

getFullYear()
getUTCFullYear()
	Returns the year as a four-digit number in local time or UTC.

getHours()
getUTCHours()
	Returns a number from 0 to 23 representing hours since midnight in local time or UTC.

getMilliseconds()
getUTCMilliSeconds()
	Returns a number from 0 to 999 representing the number of milliseconds in local time or UTC, respectively. The time is stored in hours, minutes, seconds and milliseconds.

getMinutes()
getUTCMinutes()
	Returns a number from 0 to 59 representing the minutes for the time in local time or UTC.

getMonth()
getUTCMonth()
	Returns a number from 0 (January) to 11 (December) representing the month in local time or UTC.

getSeconds()
getUTCSeconds()
	Returns a number from 0 to 59 representing the seconds for the time in local time or UTC.

getTime()	Returns the number of milliseconds between January 1, 1970, and the time in the Date object.
getTimezoneOffset()	Returns the difference in minutes between the current time on the local computer and UTC (Coordinated Universal Time).

setDate(*val*)
setUTCDate(*val*)
	Sets the day of the month (1 to 31) in local time or UTC.

setFullYear(*y*, *m*, *d*)
setUTCFullYear(*y*, *m*, *d*)
	Sets the year in local time or UTC. The second and third arguments representing the month and the date are optional. If an optional argument is not specified, the current value in the Date object is used.

setHours(*h*, *m*, *s*, *ms*)
setUTCHours(*h*, *m*, *s*, *ms*)
	Sets the hour in local time or UTC. The second, third and fourth arguments, representing the minutes, seconds and milliseconds, are optional. If an optional argument is not specified, the current value in the Date object is used.

setMilliSeconds(*ms*)
setUTCMilliseconds(*ms*)
	Sets the number of milliseconds in local time or UTC.

Fig. 11.8 | Date object methods. (Part 1 of 2.)

Method	Description

setMinutes(*m*, *s*, *ms*)
setUTCMinutes(*m*, *s*, *ms*)
	Sets the minute in local time or UTC. The second and third arguments, representing the seconds and milliseconds, are optional. If an optional argument is not specified, the current value in the Date object is used.

setMonth(*m*, *d*)
setUTCMonth(*m*, *d*)
	Sets the month in local time or UTC. The second argument, representing the date, is optional. If the optional argument is not specified, the current date value in the Date object is used.

setSeconds(*s*, *ms*)
setUTCSeconds(*s*, *ms*)
	Sets the second in local time or UTC. The second argument, representing the milliseconds, is optional. If this argument is not specified, the current millisecond value in the Date object is used.

setTime(*ms*)	Sets the time based on its argument—the number of elapsed milliseconds since January 1, 1970.
toLocaleString()	Returns a string representation of the date and time in a form specific to the computer's locale. For example, September 13, 2007, at 3:42:22 PM is represented as *09/13/07 15:47:22* in the United States and *13/09/07 15:47:22* in Europe.
toUTCString()	Returns a string representation of the date and time in the form: *15 Sep 2007 15:47:22 UTC*
toString()	Returns a string representation of the date and time in a form specific to the locale of the computer (*Mon Sep 17 15:47:22 EDT 2007* in the United States).
valueOf()	The time in number of milliseconds since midnight, January 1, 1970. (Same as getTime.)

Fig. 11.8 | Date object methods. (Part 2 of 2.)

The script of Fig. 11.9 demonstrates many of the local time zone methods in Fig. 11.8. Line 12 creates a new Date object. The new operator allocates the memory for the Date object. The empty parentheses indicate a call to the Date object's **constructor** with no arguments. A constructor is an initializer method for an object. Constructors are called automatically when an object is allocated with new. The Date constructor with no arguments initializes the Date object with the local computer's current date and time.

Software Engineering Observation 11.3

When an object is allocated with new*, the object's constructor is called automatically to initialize the object before it is used in the program.*

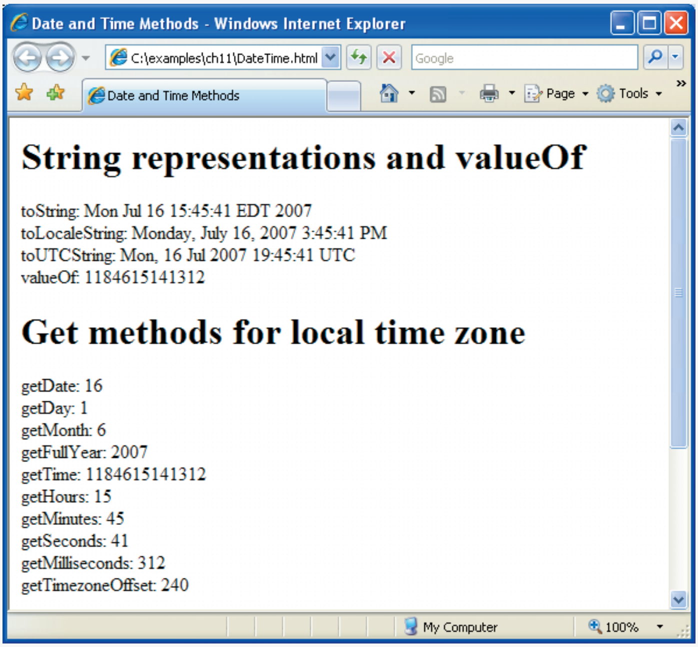
Lines 16–19 demonstrate the methods toString, toLocaleString, toUTCString and valueOf. Note that method valueOf returns a large integer value representing the total number of milliseconds between midnight, January 1, 1970, and the date and time stored in Date object current.

Lines 23–32 demonstrate the `Date` object’s *get* methods for the local time zone. Note that method `getFullYear` returns the year as a four-digit number. Note as well that method `getTimeZoneOffset` returns the difference in minutes between the local time zone and UTC time (i.e., a difference of four hours in our time zone when this example was executed).

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 11.9: DateTime.html -->
 6  <!-- Date and time methods of the Date object. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Date and Time Methods</title>
10        <script type = "text/javascript">
11           <!--
12           var current = new Date();
13
14           document.writeln(
15              "<h1>String representations and valueOf</h1>" );
16           document.writeln( "toString: " + current.toString() +
17              "<br />toLocaleString: " + current.toLocaleString() +
18              "<br />toUTCString: " + current.toUTCString() +
19              "<br />valueOf: " + current.valueOf() );
20
21           document.writeln(
22              "<h1>Get methods for local time zone</h1>" );
23           document.writeln( "getDate: " + current.getDate() +
24              "<br />getDay: " + current.getDay() +
25              "<br />getMonth: " + current.getMonth() +
26              "<br />getFullYear: " + current.getFullYear() +
27              "<br />getTime: " + current.getTime() +
28              "<br />getHours: " + current.getHours() +
29              "<br />getMinutes: " + current.getMinutes() +
30              "<br />getSeconds: " + current.getSeconds() +
31              "<br />getMilliseconds: " + current.getMilliseconds() +
32              "<br />getTimezoneOffset: " + current.getTimezoneOffset() );
33
34           document.writeln(
35              "<h1>Specifying arguments for a new Date</h1>" );
36           var anotherDate = new Date( 2007, 2, 18, 1, 5, 0, 0 );
37           document.writeln( "Date: " + anotherDate );
38
39           document.writeln( "<h1>Set methods for local time zone</h1>" );
40           anotherDate.setDate( 31 );
41           anotherDate.setMonth( 11 );
42           anotherDate.setFullYear( 2007 );
43           anotherDate.setHours( 23 );
44           anotherDate.setMinutes( 59 );
45           anotherDate.setSeconds( 59 );
46           document.writeln( "Modified date: " + anotherDate );
```

Fig. 11.9 | Date and time methods of the `Date` object. (Part 1 of 2.)

```
47            // -->
48         </script>
49      </head><body></body>
50   </html>
```

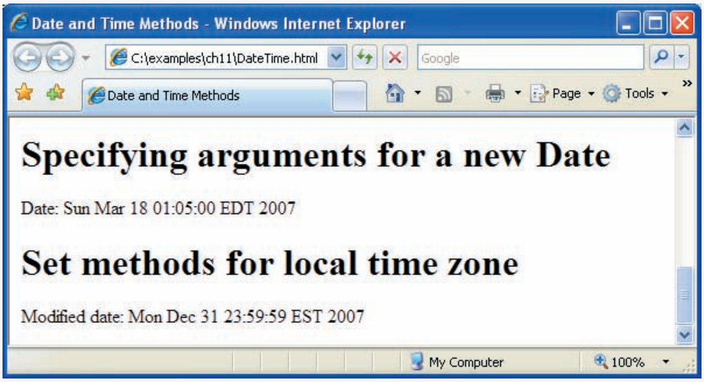

Date and Time Methods - Windows Internet Explorer
C:\examples\ch11\DateTime.html
Google
Date and Time Methods
Page
Tools
My Computer
100%

String representations and valueOf

toString: Mon Jul 16 15:45:41 EDT 2007
toLocaleString: Monday, July 16, 2007 3:45:41 PM
toUTCString: Mon, 16 Jul 2007 19:45:41 UTC
valueOf: 1184615141312

Get methods for local time zone

getDate: 16
getDay: 1
getMonth: 6
getFullYear: 2007
getTime: 1184615141312
getHours: 15
getMinutes: 45
getSeconds: 41
getMilliseconds: 312
getTimezoneOffset: 240


Date and Time Methods - Windows Internet Explorer
C:\examples\ch11\DateTime.html
Google
Date and Time Methods
Page
Tools
My Computer
100%

Specifying arguments for a new Date

Date: Sun Mar 18 01:05:00 EDT 2007

Set methods for local time zone

Modified date: Mon Dec 31 23:59:59 EST 2007

Fig. 11.9 | Date and time methods of the `Date` object. (Part 2 of 2.)

Line 36 demonstrates creating a new `Date` object and supplying arguments to the `Date` constructor for *year*, *month*, *date*, *hours*, *minutes*, *seconds* and *milliseconds*. Note that the *hours*, *minutes*, *seconds* and *milliseconds* arguments are all optional. If any one of these arguments is not specified, a zero is supplied in its place. For the *hours*, *minutes* and *seconds* arguments, if the argument to the right of any of these arguments is specified, it too must be specified (e.g., if the *minutes* argument is specified, the *hours* argument must be specified; if the *milliseconds* argument is specified, all the arguments must be specified).

Lines 40–45 demonstrate the `Date` object *set* methods for the local time zone. `Date` objects represent the month internally as an integer from 0 to 11. These values are off by one from what you might expect (i.e., 1 for January, 2 for February, …, and 12 for December). When creating a `Date` object, you must specify 0 to indicate January, 1 to indicate February, …, and 11 to indicate December.

Common Programming Error 11.2

Assuming that months are represented as numbers from 1 to 12 leads to off-by-one errors when you are processing `Dates`*.*

The `Date` object provides two other methods that can be called without creating a new `Date` object—**`Date.parse`** and **`Date.UTC`**. Method `Date.parse` receives as its argument a string representing a date and time, and returns the number of milliseconds between midnight, January 1, 1970, and the specified date and time. This value can be converted to a `Date` object with the statement

```
var theDate = new Date( numberOfMilliseconds );
```

which passes to the `Date` constructor the number of milliseconds since midnight, January 1, 1970, for the `Date` object.

Method `parse` converts the string using the following rules:

- Short dates can be specified in the form `MM-DD-YY`, `MM-DD-YYYY`, `MM/DD/YY` or `MM/DD/YYYY`. The month and day are not required to be two digits.
- Long dates that specify the complete month name (e.g., “January”), date and year can specify the month, date and year in any order.
- Text in parentheses within the string is treated as a comment and ignored. Commas and white-space characters are treated as delimiters.
- All month and day names must have at least two characters. The names are not required to be unique. If the names are identical, the name is resolved as the last match (e.g., “Ju” represents “July” rather than “June”).
- If the name of the day of the week is supplied, it is ignored.
- All standard time zones (e.g., EST for Eastern Standard Time), Coordinated Universal Time (UTC) and Greenwich Mean Time (GMT) are recognized.
- When specifying hours, minutes and seconds, separate each by colons.
- When using a 24-hour-clock format, “PM” should not be used for times after 12 noon.

`Date` method `UTC` returns the number of milliseconds between midnight, January 1, 1970, and the date and time specified as its arguments. The arguments to the `UTC` method include the required *year*, *month* and *date*, and the optional *hours*, *minutes*, *seconds* and *milliseconds*. If any of the *hours*, *minutes*, *seconds* or *milliseconds* arguments is not specified, a zero is supplied in its place. For the *hours*, *minutes* and *seconds* arguments, if the argument to the right of any of these arguments in the argument list is specified, that argument must also be specified (e.g., if the *minutes* argument is specified, the *hours* argument must be specified; if the *milliseconds* argument is specified, all the arguments must be specified). As with the result of Date.parse, the result of Date.UTC can be converted to a Date object by creating a new Date object with the result of Date.UTC as its argument.

11.6 Boolean and Number Objects

JavaScript provides the **Boolean** and **Number** objects as object **wrappers** for boolean true/false values and numbers, respectively. These wrappers define methods and properties useful in manipulating boolean values and numbers. Wrappers provide added functionality for working with simple data types.

When a JavaScript program requires a boolean value, JavaScript automatically creates a Boolean object to store the value. JavaScript programmers can create Boolean objects explicitly with the statement

```
var b = new Boolean( booleanValue );
```

The constructor argument *booleanValue* specifies whether the value of the Boolean object should be true or false. If *booleanValue* is false, 0, null, Number.NaN or an empty string (""), or if no argument is supplied, the new Boolean object contains false. Otherwise, the new Boolean object contains true. Figure 11.10 summarizes the methods of the Boolean object.

JavaScript automatically creates Number objects to store numeric values in a JavaScript program. JavaScript programmers can create a Number object with the statement

```
var n = new Number( numericValue );
```

The constructor argument *numericValue* is the number to store in the object. Although you can explicitly create Number objects, normally the JavaScript interpreter creates them as needed. Figure 11.11 summarizes the methods and properties of the Number object.

Method	Description
toString()	Returns the string "true" if the value of the Boolean object is true; otherwise, returns the string "false".
valueOf()	Returns the value true if the Boolean object is true; otherwise, returns false.

Fig. 11.10 | Boolean object methods.

Method or property	Description
toString(*radix*)	Returns the string representation of the number. The optional *radix* argument (a number from 2 to 36) specifies the number's base. For example, radix 2 results in the binary representation of the number, 8 results in the octal representation, 10 results in the decimal representation and 16 results in the hexadecimal representation. See Appendix E, Number Systems, for a review of the binary, octal, decimal and hexadecimal number systems.

Fig. 11.11 | Number object methods and properties. (Part 1 of 2.)

Method or property	Description
`valueOf()`	Returns the numeric value.
`Number.MAX_VALUE`	This property represents the largest value that can be stored in a JavaScript program—approximately 1.79E+308.
`Number.MIN_VALUE`	This property represents the smallest value that can be stored in a JavaScript program—approximately 5.00E–324.
`Number.NaN`	This property represents *not a number*—a value returned from an arithmetic expression that does not result in a number (e.g., the expression `parseInt( "hello" )` cannot convert the string `"hello"` into a number, so `parseInt` would return `Number.NaN`. To determine whether a value is `NaN`, test the result with function `isNaN`, which returns `true` if the value is `NaN`; otherwise, it returns `false`.
`Number.NEGATIVE_INFINITY`	This property represents a value less than `-Number.MAX_VALUE`.
`Number.POSITIVE_INFINITY`	This property represents a value greater than `Number.MAX_VALUE`.

Fig. 11.11 | `Number` object methods and properties. (Part 2 of 2.)

11.7 document Object

The **`document`** object is used to manipulate the document that is currently visible in the browser window. The `document` object has many properties and methods, such as methods `document.write` and `document.writeln`, which have both been used in prior JavaScript examples. Figure 11.12 shows the methods and properties of the `document` objects that are used in this chapter. You can learn more about the properties and methods of the `document` object in our JavaScript Resource Center (`www.deitel.com/javascript`).

Method or property	Description
`getElementById(` *id* `)`	Returns the DOM node representing the XHTML element whose `id` attribute matches *id*.
`write(` *string* `)`	Writes the string to the XHTML document as XHTML code.
`writeln(` *string* `)`	Writes the string to the XHTML document as XHTML code and adds a newline character at the end.
`cookie`	A string containing the values of all the cookies stored on the user's computer for the current document. See Section 11.9, Using Cookies.
`lastModified`	The date and time that this document was last modified.

Fig. 11.12 | Important `document` object methods and properties.

11.8 window Object

The **window** object provides methods for manipulating browser windows. The following script shows many of the commonly used properties and methods of the `window` object and uses them to create a website that spans multiple browser windows. Figure 11.13 allows the user to create a new, fully customized browser window by completing an XHTML form and clicking the **Submit** button. The script also allows the user to add text to the new window and navigate the window to a different URL.

The script starts in line 10. Line 12 declares a variable to refer to the new window. We refer to the new window as the **child window** because it is created and controlled by the main, or **parent**, window in this script. Lines 14–50 define the `createChildWindow` function, which determines the features that have been selected by the user and creates a child window with those features (but does not add any content to the window). Lines 18–20 declare several variables to store the status of the checkboxes on the page. Lines 23–38 set each variable to `"yes"` or `"no"` based on whether the corresponding checkbox is checked or unchecked.

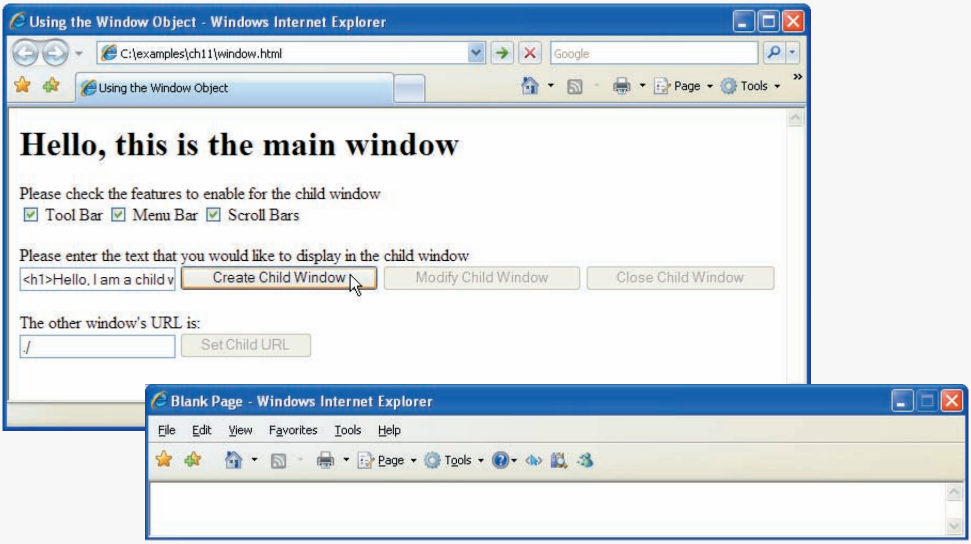
```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 11.13: window.html -->
 6  <!-- Using the window object to create and modify child windows. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8  <head>
 9  <title>Using the Window Object</title>
10  <script type = "text/javascript">
11     <!--
12     var childWindow; // variable to control the child window
13
14     function createChildWindow()
15     {
16        // these variables all contain either "yes" or "no"
17        // to enable or disable a feature in the child window
18        var toolBar;
19        var menuBar;
20        var scrollBars;
21
22        // determine whether the Tool Bar checkbox is checked
23        if ( document.getElementById( "toolBarCheckBox" ).checked )
24           toolBar = "yes";
25        else
26           toolBar = "no";
27
28        // determine whether the Menu Bar checkbox is checked
29        if ( document.getElementById( "menuBarCheckBox" ).checked )
30           menuBar = "yes";
31        else
32           menuBar = "no";
33
```

Fig. 11.13 | Using the `window` object to create and modify child windows. (Part 1 of 4.)

```
         // determine whether the Scroll Bar checkbox is checked
         if ( document.getElementById( "scrollBarsCheckBox" ).checked )
            scrollBars = "yes";
         else
            scrollBars = "no";

         //display window with selected features
         childWindow = window.open( "", "",
            ",toolbar = " + toolBar +
            ",menubar = " + menuBar +
            ",scrollbars = " + scrollBars );

         // disable buttons
         document.getElementById( "closeButton" ).disabled = false;
         document.getElementById( "modifyButton" ).disabled = false;
         document.getElementById( "setURLButton" ).disabled = false;
      } // end function createChildWindow

      // insert text from the textbox in the child window
      function modifyChildWindow()
      {
         if ( childWindow.closed )
            alert( "You attempted to interact with a closed window" );
         else
            childWindow.document.write(
               document.getElementById( "textForChild" ).value );
      } // end function modifyChildWindow

      // close the child window
      function closeChildWindow()
      {
         if ( childWindow.closed )
            alert( "You attempted to interact with a closed window" );
         else
            childWindow.close();

         document.getElementById( "closeButton" ).disabled = true;
         document.getElementById( "modifyButton" ).disabled = true;
         document.getElementById( "setURLButton" ).disabled = true;
      } // end function closeChildWindow

      // set the URL of the child window to the URL
      // in the parent window's myChildURL
      function setChildWindowURL()
      {
         if ( childWindow.closed )
            alert( "You attempted to interact with a closed window" );
         else
            childWindow.location =
               document.getElementById( "myChildURL" ).value;
      } // end function setChildWindowURL
      //-->
   </script>
```

Fig. 11.13 | Using the `window` object to create and modify child windows. (Part 2 of 4.)

```
</head>
<body>
   <h1>Hello, this is the main window</h1>
   <p>Please check the features to enable for the child window<br/>
   <input id = "toolBarCheckBox" type = "checkbox" value = ""
      checked = "checked" />
      <label>Tool Bar</label>
   <input id = "menuBarCheckBox" type = "checkbox" value = ""
      checked = "checked" />
      <label>Menu Bar</label>
   <input id = "scrollBarsCheckBox" type = "checkbox" value = ""
      checked = "checked" />
      <label>Scroll Bars</label></p>

   <p>Please enter the text that you would like to display
   in the child window<br/>
   <input id = "textForChild" type = "text"
      value = "<h1>Hello, I am a child window.</h1> " />
   <input id = "createButton" type = "button"
      value = "Create Child Window" onclick = "createChildWindow()" />
   <input id= "modifyButton" type = "button" value = "Modify Child Window"
      onclick = "modifyChildWindow()" disabled = "disabled" />
   <input id = "closeButton" type = "button" value = "Close Child Window"
      onclick = "closeChildWindow()" disabled = "disabled" /></p>

   <p>The other window's URL is: <br/>
   <input id = "myChildURL" type = "text" value = "./" />
   <input id = "setURLButton" type = "button" value = "Set Child URL"
      onclick = "setChildWindowURL()" disabled = "disabled" /></p>
</body>
</html>
```



Using the Window Object - Windows Internet Explorer

C:\examples\ch11\window.html

Hello, this is the main window

Please check the features to enable for the child window

☑ Tool Bar ☑ Menu Bar ☑ Scroll Bars

Please enter the text that you would like to display in the child window

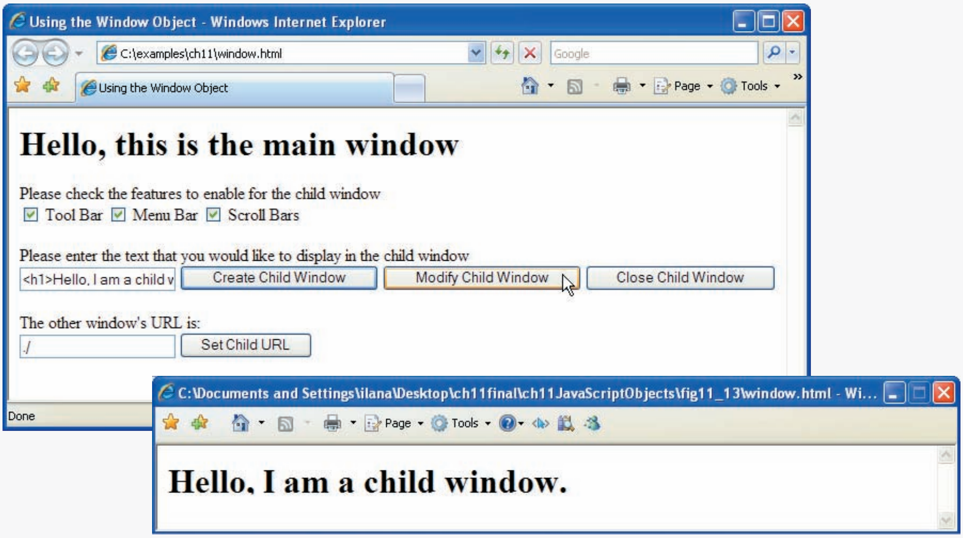
<h1>Hello, I am a child v

The other window's URL is:

./

Blank Page - Windows Internet Explorer

Fig. 11.13 | Using the `window` object to create and modify child windows. (Part 3 of 4.)


Using the Window Object - Windows Internet Explorer

C:\examples\ch11\window.html

Hello, this is the main window

Please check the features to enable for the child window

☑ Tool Bar ☑ Menu Bar ☑ Scroll Bars

Please enter the text that you would like to display in the child window

<h1>Hello, I am a child v

The other window's URL is:

./

Done

C:\Documents and Settings\ilana\Desktop\ch11final\ch11JavaScriptObjects\fig11_13\window.html - Wi...

Hello, I am a child window.

Fig. 11.13 | Using the `window` object to create and modify child windows. (Part 4 of 4.)

The statement in lines 41–44 uses the `window` object's `open` method to create the requested child window. Method `open` has three parameters. The first parameter is the URL of the page to open in the new window, and the second parameter is the name of the window. If you specify the `target` attribute of an `a` (anchor) element to correspond to the name of a window, the `href` of the link will be opened in the window. In our example, we pass `window.open` empty strings as the first two parameter values because we want the new window to open a blank page, and we use a different method to manipulate the child window's URL.

The third parameter of the `open` method is a string of comma-separated, all-lowercase feature names, each followed by an = sign and either `"yes"` or `"no"` to determine whether that feature should be displayed in the new window. If these parameters are omitted, the browser defaults to a new window containing an empty page, no title and all features visible. [*Note:* If your menu bar is normally hidden in IE7, it will not appear in the child window. Press the *Alt* key to display it.] Lines 47–49 enable the buttons for manipulating the child window—these are initially disabled when the page loads.

Lines 53–60 define the function `modifyChildWindow`, which adds a line of text to the content of the child window. In line 55, the script determines whether the child window is closed. Function `modifyChildWindow` uses property `childWindow.closed` to obtain a boolean value that is `true` if `childWindow` is closed and `false` if the window is still open. If the window is closed, an alert box is displayed notifying the user that the window is currently closed and cannot be modified. If the child window is open, lines 58–59 obtain text from the `textForChild` input (lines 103–104) in the XHTML form in the parent window and uses the child's `document.write` method to write this text to the child window.

Function `closeChildWindow` (lines 63–73) also determines whether the child window is closed before proceeding. If the child window is closed, the script displays an alert box telling the user that the window is already closed. If the child window is open, line 68 closes it using the `childWindow.close` method. Lines 70–72 disable the buttons that interact with the child window.

Look-and-Feel Observation 11.1

Popup windows should be used sparingly. Many users dislike websites that open additional windows, or that resize or reposition the browser. Some some users have popup blockers that will prevent new windows from opening.

Software Engineering Observation 11.4

`window.location` *is a property that always contains a string representation of the URL displayed in the current window. Typically, web browsers will allow a script to retrieve the* `window.location` *property of another window only if the script belongs to the same website as the page in the other window.*

Function `setChildWindowURL` (lines 77–84) copies the contents of the `myChildURL` text field to the `location` property of the child window. If the child window is open, lines 81–82 set property `location` of the child window to the string in the `myChildURL` textbox. This action changes the URL of the child window and is equivalent to typing a new URL into the window's address bar and clicking **Go** (or pressing *Enter*).

The script ends in line 86. Lines 88–116 contain the body of the XHTML document, comprising a form that contains checkboxes, buttons, textboxes and form field labels. The script uses the form elements defined in the body to obtain input from the user. Lines 106, 108, 110, and 115 specify the `onclick` attributes of XHTML buttons. Each button is set to call a corresponding JavaScript function when clicked.

Figure 11.14 contains a list of some commonly used methods and properties of the `window` object.

Method or property	Description
`open(` *url*, *name*, *options* `)`	Creates a new window with the URL of the window set to *url*, the name set to *name* to refer to it in the script, and the visible features set by the string passed in as *option*.
`prompt(` *prompt*, *default* `)`	Displays a dialog box asking the user for input. The text of the dialog is *prompt*, and the default value is set to *default*.
`close()`	Closes the current window and deletes its object from memory.
`focus()`	This method gives focus to the window (i.e., puts the window in the foreground, on top of any other open browser windows).
`blur()`	This method takes focus away from the window (i.e., puts the window in the background).
`window.document`	This property contains the `document` object representing the document currently inside the window.
`window.closed`	This property contains a boolean value that is set to true if the window is closed, and false if it is not.

Fig. 11.14 | Important `window` object methods and properties. (Part 1 of 2.)

Method or property	Description
`window.opener`	This property contains the `window` object of the window that opened the current window, if such a window exists.

Fig. 11.14 | Important `window` object methods and properties. (Part 2 of 2.)

11.9 Using Cookies

Cookies provide web developers with a tool for personalizing web pages. A **cookie** is a piece of data that is stored on the user's computer to maintain information about the client during and between browser sessions. A website may store a cookie on the client's computer to record user preferences or other information that the website can retrieve during the client's subsequent visits. For example, a website can retrieve the user's name from a cookie and use it to display a personalized greeting.

Microsoft Internet Explorer and Mozilla Firefox store cookies as small text files on the client's hard drive. When a user visits a website, the browser locates any cookies written by scripts on that site and makes them available to any scripts located on the site. Note that cookies may be accessed only by scripts belonging to the same website from which they originated (i.e., a cookie set by a script on `amazon.com` can be read only by other scripts on `amazon.com`).

Cookies are accessible in JavaScript through the `document` object's **`cookie`** property. JavaScript treats a cookie as a string of text. Any standard string function or method can manipulate a cookie. A cookie has the syntax "*identifier=value*," where *identifier* is any valid JavaScript variable identifier, and *value* is the value of the cookie variable. When multiple cookies exist for one website, *identifier-value* pairs are separated by semicolons in the `document.cookie` string.

Cookies differ from ordinary strings in that each cookie has an expiration date, after which the web browser deletes it. This date can be defined by setting the **`expires`** property in the cookie string. If a cookie's expiration date is not set, then the cookie expires by default after the user closes the browser window. A cookie can be deleted immediately by setting the `expires` property to a date and time in the past.

The assignment operator does not overwrite the entire list of cookies, but appends a cookie to the end of it. Thus, if we set two cookies

```
document.cookie = "name1=value1;";
document.cookie = "name2=value2;";
```

`document.cookie` will contain `"name1=value1; name2=value2"`.

Figure 11.15 uses a cookie to store the user's name and displays a personalized greeting. This example improves upon the functionality in the dynamic welcome page example of Fig. 6.17 by requiring the user to enter a name only during the first visit to the web page. On each subsequent visit, the script can display the user name that is stored in the cookie.

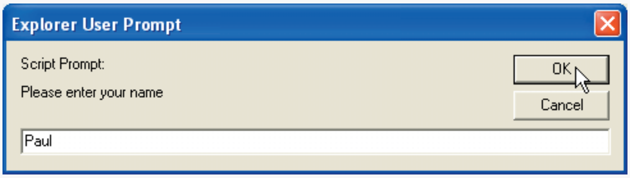
Line 10 begins the script. Lines 12–13 declare the variables needed to obtain the time, and line 14 declares the variable that stores the name of the user. Lines 16–27 contain the same `if...else` statement used in Fig. 6.17 to display a time-sensitive greeting.

Lines 30–66 contain the code used to manipulate the cookie. Line 30 determines whether a cookie exists on the client computer. The expression `document.cookie` evaluates to `true` if a cookie exists. If a cookie does not exist, then the script prompts the user to enter a name (line 45). The script creates a cookie containing the string `"name="`, followed by a copy of the user's name produced by the built-in JavaScript function **`escape`** (line 49). The function `escape` converts any non-alphanumeric characters, such as spaces

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 11.15: cookie.html -->
 6  <!-- Using cookies to store user identification data. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Using Cookies</title>
10        <script type = "text/javascript">
11           <!--
12           var now = new Date(); // current date and time
13           var hour = now.getHours(); // current hour (0-23)
14           var name;
15
16           if ( hour < 12 ) // determine whether it is morning
17              document.write( "<h1>Good Morning, " );
18           else
19           {
20              hour = hour - 12; // convert from 24-hour clock to PM time
21
22              // determine whether it is afternoon or evening
23              if ( hour < 6 )
24                 document.write( "<h1>Good Afternoon, " );
25              else
26                 document.write( "<h1>Good Evening, " );
27           } // end else
28
29           // determine whether there is a cookie
30           if ( document.cookie )
31           {
32              // convert escape characters in the cookie string to their
33              // English notation
34              var myCookie = unescape( document.cookie );
35
36              // split the cookie into tokens using = as delimiter
37              var cookieTokens = myCookie.split( "=" );
38
39              // set name to the part of the cookie that follows the = sign
40              name = cookieTokens[ 1 ];
41           } // end if
42           else
43           {
44              // if there was no cookie, ask the user to input a name
45              name = window.prompt( "Please enter your name", "Paul" );
```

Fig. 11.15 | Using cookies to store user identification data. (Part 1 of 3.)

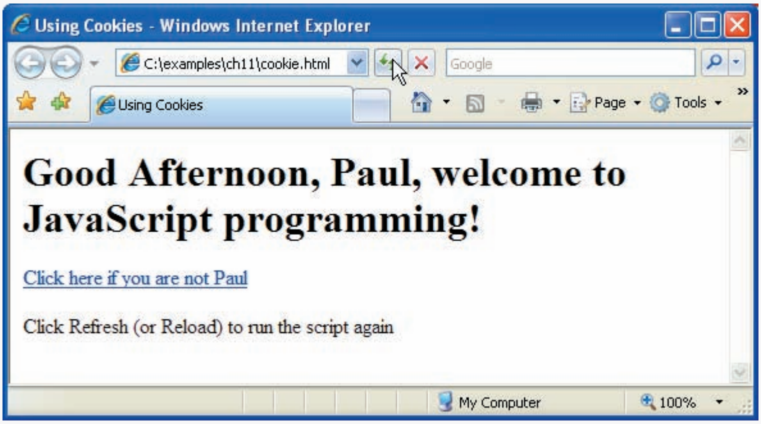
```

            // escape special characters in the name string
            // and add name to the cookie
            document.cookie = "name=" + escape( name );
         } // end else

         document.writeln(
            name + ", welcome to JavaScript programming!</h1>" );
         document.writeln( "<a href = 'javascript:wrongPerson()'> " +
            "Click here if you are not " + name + "</a>" );

         // reset the document's cookie if wrong person
         function wrongPerson()
         {
            // reset the cookie
            document.cookie= "name=null;" +
               " expires=Thu, 01-Jan-95 00:00:01 GMT";

            // reload the page to get a new name after removing the cookie
            location.reload();
         } // end function wrongPerson

         // -->
      </script>
   </head>
   <body>
      <p>Click Refresh (or Reload) to run the script again</p>
   </body>
</html>
```



Script Prompt:

Please enter your name

Paul


Using Cookies - Windows Internet Explorer
C:\examples\ch11\cookie.html
Google
Using Cookies
Page
Tools
My Computer
100%

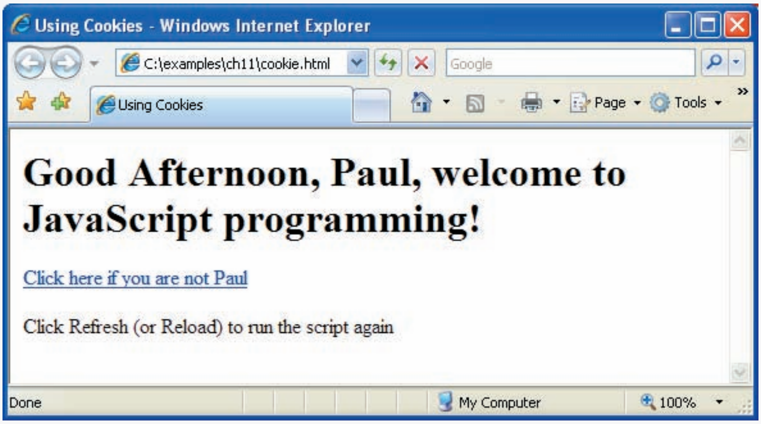
Good Afternoon, Paul, welcome to JavaScript programming!

Click here if you are not Paul

Click Refresh (or Reload) to run the script again

Fig. 11.15 | Using cookies to store user identification data. (Part 2 of 3.)

AFter "refresh" is clicked, the website recalls the previously input data.



Using Cookies - Windows Internet Explorer
C:\examples\ch11\cookie.html
Google
Using Cookies
Page
Tools
Done
My Computer
100%

Good Afternoon, Paul, welcome to JavaScript programming!

Click here if you are not Paul

Click Refresh (or Reload) to run the script again

Fig. 11.15 | Using cookies to store user identification data. (Part 3 of 3.)

and semicolons, in a string to their equivalent **hexadecimal escape sequences** of the form "%*XX*," where *XX* is the two-digit hexadecimal ASCII value of a special character. For example, if `name` contains the value `"David Green"`, the statement `escape( name )` evaluates to `"David%20Green"`, because the hexadecimal ASCII value of a blank space is `20`. It is a good idea to always escape cookie values before writing them to the client. This conversion prevents any special characters in the cookie from being misinterpreted as having a special meaning in the code, rather than being a character in a cookie value. For instance, a semicolon in a cookie value could be misinterpreted as a semicolon separating two adjacent *identifier-value* pairs. Applying the function **`unescape`** to cookies when they are read out of the `document.cookie` string converts the hexadecimal escape sequences back to English characters for display in a web page.

Good Programming Practice 11.2

Always store values in cookies with self-documenting identifiers. Do not forget to include the identifier followed by an = sign before the value being stored.

If a cookie exists (i.e., the user has been to the page before), then the script **parses** the user name out of the cookie string and stores it in a local variable. Parsing generally refers to the act of splitting a string into smaller, more useful components. Line 34 uses the JavaScript function `unescape` to replace all the escape sequences in the cookie with their equivalent English-language characters. The script stores the unescaped cookie value in the variable `myCookie` (line 34) and uses the JavaScript function `split` (line 37), introduced in Section 11.4.5, to break the cookie into identifier and value tokens. At this point in the script, `myCookie` contains a string of the form `"name=`*value*`"`. We call `split` on `myCookie` with = as the delimiter to obtain the `cookieTokens` array, with the first element equal to the name of the identifier and the second element equal to the value of the identifier. Line 40 assigns the value of the second element in the `cookieTokens` array (i.e., the actual value stored in the cookie) to the variable `name`. Lines 52–53 add the personalized greeting to the web page, using the user's name stored in the cookie.

The script allows the user to reset the cookie, which is useful in case someone new is using the computer. Lines 54–55 create a hyperlink that, when clicked, calls the JavaScript function `wrongPerson` (lines 58–66). Lines 61–62 set the cookie `name` to `null` and the `expires` property to January 1, 1995 (though any date in the past will suffice). Internet Explorer detects that the `expires` property is set to a date in the past and deletes the cookie from the user's computer. The next time this page loads, no cookie will be found. The **`reload`** method of the `location` object forces the page to refresh (line 65), and, unable to find an existing cookie, the script prompts the user to enter a new name.

11.10 Final JavaScript Example

The past few chapters have explored many JavaScript concepts and how they can be applied on the web. The next JavaScript example combines many of these concepts into a single web page. Figure 11.16 uses functions, cookies, arrays, loops, the `Date` object, the `window` object and the `document` object to create a sample welcome screen containing a personalized greeting, a short quiz, a random image and a random quotation. We have seen all of these concepts before, but this example illustrates how they work together on one web page.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 11.16: final.html -->
<!-- Rich welcome page using several JavaScript concepts. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Putting It All Together</title>
      <script type = "text/javascript">
         <!--
         var now = new Date(); // current date and time
         var hour = now.getHours(); // current hour

         // array with names of the images that will be randomly selected
         var pictures =
            [ "CPE", "EPT", "GPP", "GUI", "PERF", "PORT", "SEO" ];

         // array with the quotes that will be randomly selected
         var quotes = [ "Form ever follows function.<br/>" +
            " Louis Henri Sullivan", "E pluribus unum." +
            " (One composed of many.) <br/> Virgil", "Is it a" +
            " world to hide virtues in?<br/> William Shakespeare" ];

         // write the current date and time to the web page
         document.write( "<p>" + now.toLocaleString() + "<br/></p>" );

         // determine whether it is morning
         if ( hour < 12 )
            document.write( "<h2>Good Morning, " );
```

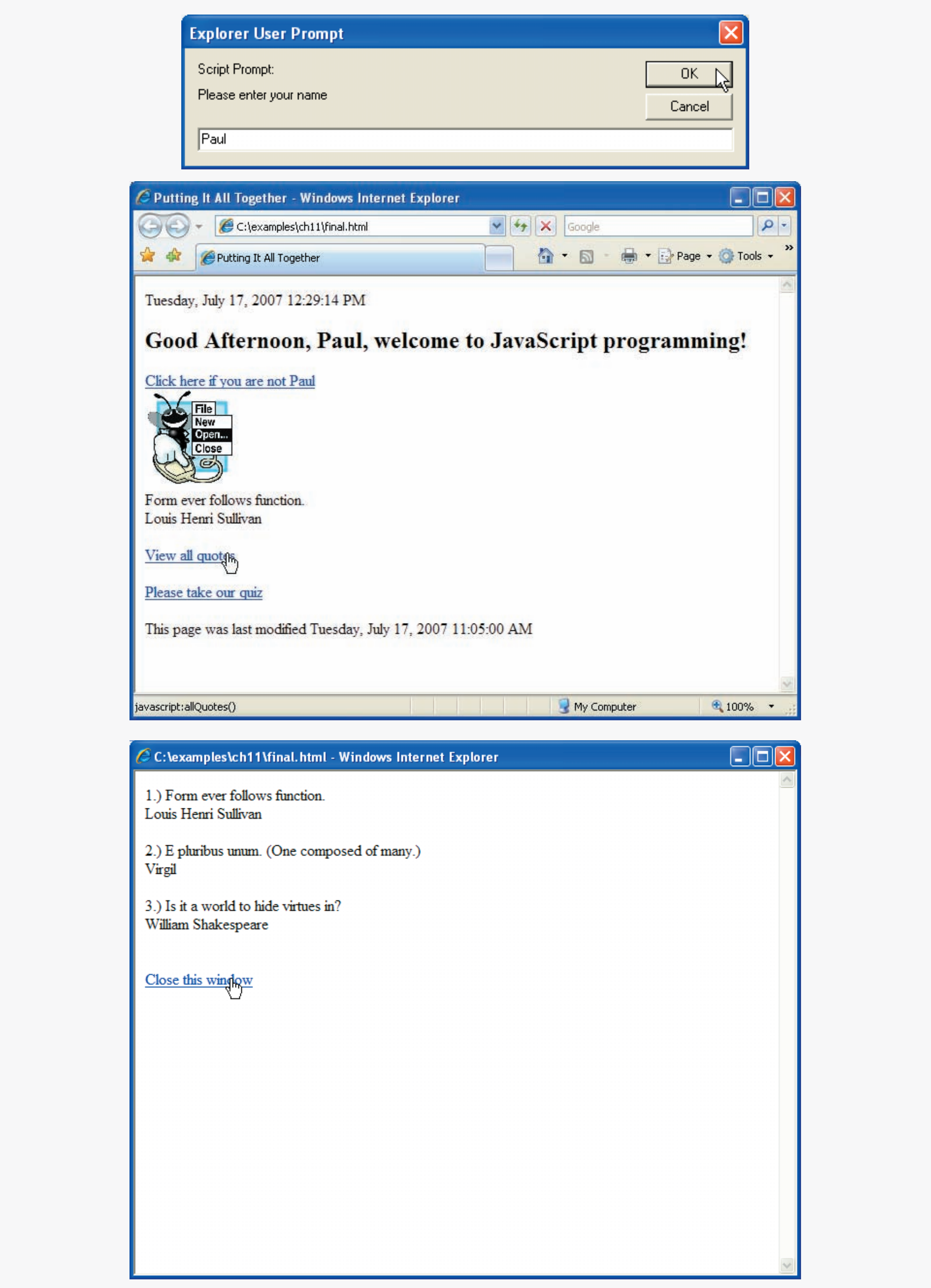
Fig. 11.16 | Rich welcome page using several JavaScript concepts. (Part 1 of 5.)

```
         else
         {
            hour = hour - 12; // convert from 24-hour clock to PM time

            // determine whether it is afternoon or evening
            if ( hour < 6 )
               document.write( "<h2>Good Afternoon, " );
            else
               document.write( "<h2>Good Evening, " );
         } // end else

         // determine whether there is a cookie
         if ( document.cookie )
         {
            // convert escape characters in the cookie string to their
            // English notation
            var myCookie = unescape( document.cookie );

            // split the cookie into tokens using = as delimiter
            var cookieTokens = myCookie.split( "=" );

            // set name to the part of the cookie that follows the = sign
            name = cookieTokens[ 1 ];
         } // end if
         else
         {
            // if there was no cookie, ask the user to input a name
            name = window.prompt( "Please enter your name", "Paul" );

            // escape special characters in the name string
            // and add name to the cookie
            document.cookie = "name =" + escape( name );
         } // end else

         // write the greeting to the page
         document.writeln(
            name + ", welcome to JavaScript programming!</h2>" );

         // write the link for deleting the cookie to the page
         document.writeln( "<a href = \"javascript:wrongPerson()\" > " +
            "Click here if you are not " + name + "</a><br/>" );

         // write the random image to the page
         document.write ( "<img src = \"" +
            pictures[ Math.floor( Math.random() * 7 ) ] +
            ".gif\" /> <br/>" );

         // write the random quote to the page
         document.write ( quotes[ Math.floor( Math.random() * 3 ) ] );

         // create a window with all the quotes in it
         function allQuotes()
         {
```

Fig. 11.16 | Rich welcome page using several JavaScript concepts. (Part 2 of 5.)

```
            // create the child window for the quotes
            var quoteWindow = window.open( "", "", "resizable=yes, " +
               "toolbar=no, menubar=no, status=no, location=no," +
               " scrollBars=yes" );
            quoteWindow.document.write( "<p>" )

            // loop through all quotes and write them in the new window
            for ( var i = 0; i < quotes.length; i++ )
               quoteWindow.document.write( ( i + 1 ) + ".) " +
                  quotes[ i ] + "<br/><br/>");

            // write a close link to the new window
            quoteWindow.document.write( "</p><br/><a href = " +
               "\"javascript:window.close()\">Close this window</a>" );
         } // end function allQuotes

         // reset the document's cookie if wrong person
         function wrongPerson()
         {
            // reset the cookie
            document.cookie= "name=null;" +
               " expires=Thu, 01-Jan-95 00:00:01 GMT";

            // reload the page to get a new name after removing the cookie
            location.reload();
         } // end function wrongPerson

         // open a new window with the quiz2.html file in it
         function openQuiz()
         {
            window.open( "quiz2.html", "", "toolbar = no, " +
                "menubar = no, scrollBars = no" );
         } // end function openQuiz
      // -->
      </script>
   </head>
   <body>
      <p><a href = "javascript:allQuotes()">View all quotes</a></p>

      <p id = "quizSpot">
         <a href = "javascript:openQuiz()">Please take our quiz</a></p>

      <script type = "text/javascript">
         // variable that gets the last modification date and time
         var modDate = new Date( document.lastModified );

         // write the last modified date and time to the page
         document.write ( "This page was last modified " +
            modDate.toLocaleString() );
      </script>
   </body>
</html>
```

Fig. 11.16 | Rich welcome page using several JavaScript concepts. (Part 3 of 5.)



Explorer User Prompt
OK
Cancel

Script Prompt:

Please enter your name

Paul

Putting It All Together - Windows Internet Explorer
C:\examples\ch11\final.html
Google
Putting It All Together
Page
Tools

Tuesday, July 17, 2007 12:29:14 PM

Good Afternoon, Paul, welcome to JavaScript programming!

Click here if you are not Paul

Form ever follows function.
Louis Henri Sullivan

View all quotes

Please take our quiz

This page was last modified Tuesday, July 17, 2007 11:05:00 AM

javascript:allQuotes()

My Computer
100%
C:\examples\ch11\final.html - Windows Internet Explorer

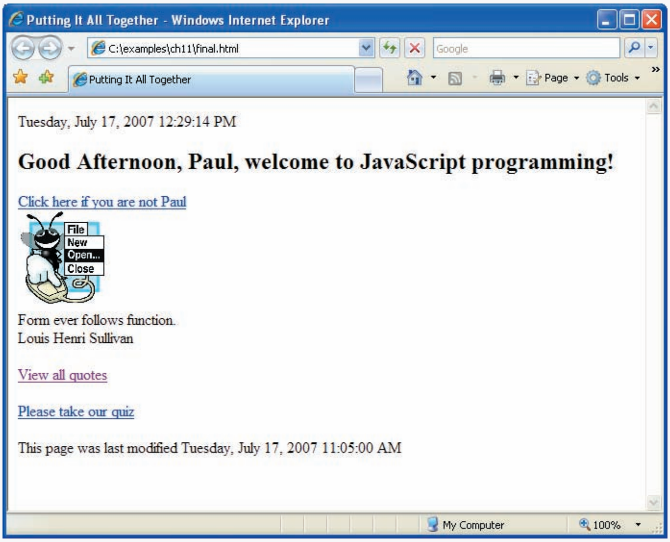
1.) Form ever follows function.
Louis Henri Sullivan

2.) E pluribus unum. (One composed of many.)
Virgil

3.) Is it a world to hide virtues in?
William Shakespeare

Close this window

Fig. 11.16 | Rich welcome page using several JavaScript concepts. (Part 4 of 5.)



Putting It All Together - Windows Internet Explorer
C:\examples\ch11\final.html
Google
Putting It All Together
Page
Tools
File
New
Open...
Close
My Computer
100%

Tuesday, July 17, 2007 12:29:14 PM

Good Afternoon, Paul, welcome to JavaScript programming!

Click here if you are not Paul

Form ever follows function.
Louis Henri Sullivan

View all quotes

Please take our quiz

This page was last modified Tuesday, July 17, 2007 11:05:00 AM

Fig. 11.16 | Rich welcome page using several JavaScript concepts. (Part 5 of 5.)

The script that builds most of this page starts in line 10. Lines 12–13 declare variables needed for determining the time of day. Lines 16–23 create two arrays from which content is randomly selected. This web page contains both an image (whose filename is randomly selected from the `pictures` array) and a quote (whose text is randomly selected from the `quotes` array). Line 26 writes the user's local date and time to the web page using the `Date` object's `toLocaleString` method. Lines 29–40 display a time-sensitive greeting using the same code as Fig. 6.17. The script either uses an existing cookie to obtain the user's name (lines 43–54) or prompts the user for a name, which the script then stores in a new cookie (lines 55–63). Lines 66–67 write the greeting to the web page, and lines 70–71 produce the link for resetting the cookie. This is the same code used in Fig. 11.15 to manipulate cookies. Lines 74–79 write the random image and random quote to the web page. The script chooses each by randomly selecting an index into each array. This code is similar to the code used in Fig. 10.7 to display a random image using an array.

Function `allQuotes` (lines 82–98) uses the `window` object and a `for` loop to open a new window containing all the quotes in the `quotes` array. Lines 85–87 create a new window called `quoteWindow`. The script does not assign a URL or a name to this window, but it does specify the window features to display. Line 88 opens a new paragraph in `quoteWindow`. A `for` loop (lines 91–93) traverses the quotes array and writes each quote to `quoteWindow`. Lines 96–97 close the paragraph in `quoteWindow`, insert a new line and add a link at the bottom of the page that allows the user to close the window. Note that `allQuotes` generates a web page and opens it in an entirely new window with JavaScript.

Function `wrongPerson` (lines 101–109) resets the cookie storing the user's name. This function is identical to function `wrongPerson` in Fig. 11.15.

Function `openQuiz` (lines 112–116) opens a new window to display a sample quiz. Using the `window.open` method, the script creates a new window containing `quiz2.html` (lines 114–115). We discuss `quiz2.html` later in this section.

The primary script ends in line 118, and the body of the XHTML document begins in line 120. Line 121 creates the link that calls function `allQuotes` when clicked. Lines 123–124 create a paragraph element containing the attribute `id = "quizSpot"`. This paragraph contains a link that calls function `openQuiz`.

Lines 126–133 contain a second script. This script appears in the XHTML document's `body` because it adds a dynamic footer to the page, which must appear after the static XHTML content contained in the first part of the `body`. This script creates another instance of the `Date` object, but the date is set to the last modified date and time of the XHTML document, rather than the current date and time (line 128). The script obtains the last modified date and time using property `document.lastModified`. Lines 131–132 add this information to the web page. Note that the last modified date and time appear at the bottom of the page, after the rest of the `body` content. If this script were in the `head` element, this information would be displayed before the entire `body` of the XHTML document. Lines 133–135 close the `script`, the `body` and the XHTML document.

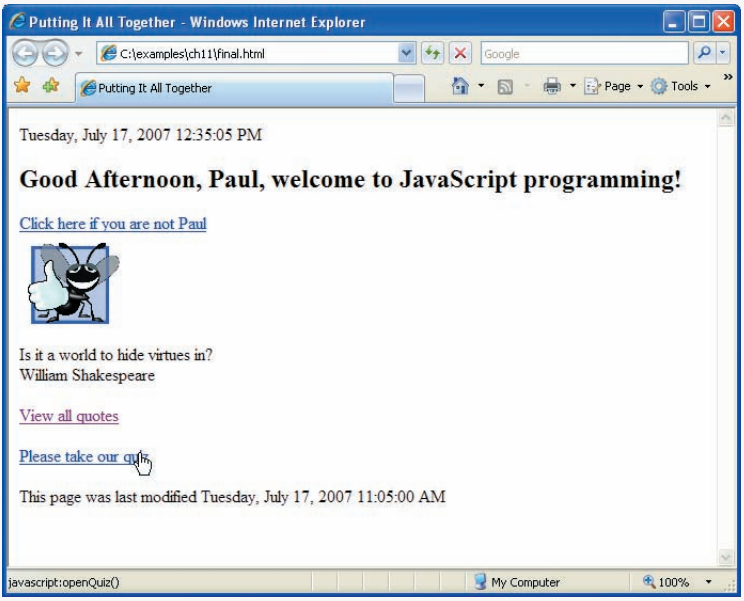
The Quiz Page

The quiz used in this example is in a separate XHTML document named `quiz2.html` (Fig. 11.17). This document is similar to `quiz.html` in Fig. 10.14. The quiz in this example differs from the quiz in Fig. 10.14 in that it shows the result in the main window in the example, whereas the earlier quiz example alerts the result. After the **Submit** button in the quiz window is clicked, the main window changes to reflect that the quiz was taken, and the quiz window closes.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 11.17: quiz2.html -->
<!-- Online quiz in a child window. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Online Quiz</title>
      <script type = "text/JavaScript">
         <!--
         function checkAnswers()
         {
            // determine whether the answer is correct
            if ( document.getElementById( "myQuiz" ).elements[1].checked )
               window.opener.document.getElementById( "quizSpot" ).
                  innerHTML = "Congratulations, your answer is correct";
            else // if the answer is incorrect
               window.opener.document.getElementById( "quizSpot" ).
                  innerHTML = "Your answer is incorrect. " +
                  "Please try again <br /> <a href = " +
                  \"javascript:openQuiz()\">Please take our quiz</a>";

            window.opener.focus();
            window.close();
         } // end function checkAnswers
```

Fig. 11.17 | Online quiz in a child window. (Part 1 of 3.)

```
         //-->
      </script>
   </head>
   <body>
      <form id = "myQuiz" action = "javascript:checkAnswers()">
         <p>Select the name of the tip that goes with the
            image shown:<br />
            <img src = "EPT.gif" alt = "mystery tip"/>
            <br />

            <input type = "radio" name = "radiobutton" value = "CPE" />
            <label>Common Programming Error</label>

            <input type = "radio" name = "radiobutton" value = "EPT" />
            <label>Error-Prevention Tip</label>

            <input type = "radio" name = "radiobutton" value = "PERF" />
            <label>Performance Tip</label>

            <input type = "radio" name = "radiobutton" value = "PORT" />
            <label>Portability Tip</label><br />

            <input type = "submit" name = "Submit" value = "Submit" />
            <input type = "reset" name = "reset" value = "Reset" />
         </p>
      </form>
   </body>
</html>
```


Putting It All Together - Windows Internet Explorer
C:\examples\ch11\final.html
Google
Putting It All Together
Page
Tools
javascript:openQuiz()
My Computer
100%

Tuesday, July 17, 2007 12:35:05 PM

Good Afternoon, Paul, welcome to JavaScript programming!

Click here if you are not Paul

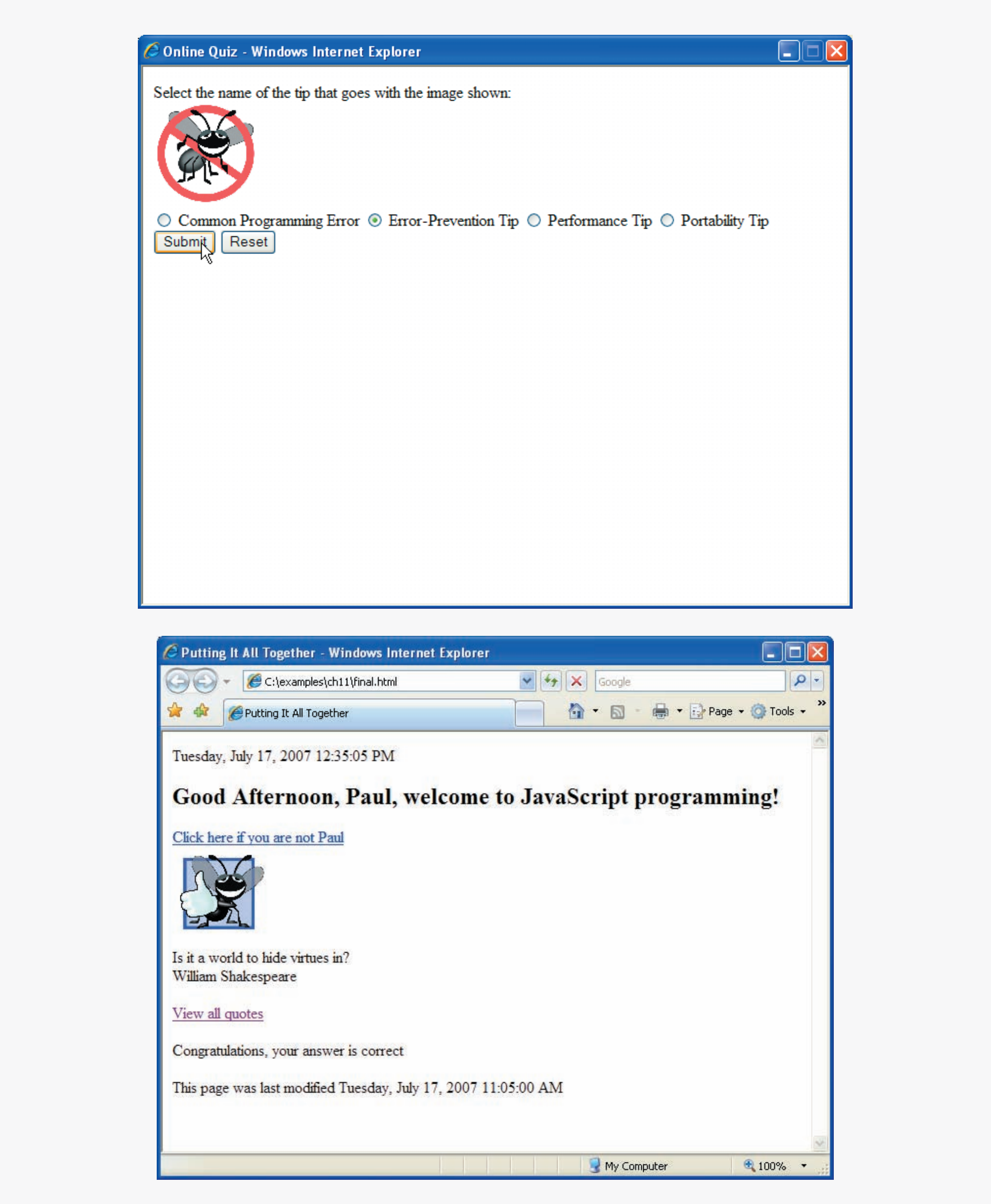
Is it a world to hide virtues in?
William Shakespeare

View all quotes

Please take our quiz

This page was last modified Tuesday, July 17, 2007 11:05:00 AM

Fig. 11.17 | Online quiz in a child window. (Part 2 of 3.)



Online Quiz - Windows Internet Explorer
Submit
Reset
Putting It All Together - Windows Internet Explorer
C:\examples\ch11\final.html
Google
Putting It All Together
Page
Tools
My Computer
100%

Select the name of the tip that goes with the image shown:

○ Common Programming Error ● Error-Prevention Tip ○ Performance Tip ○ Portability Tip

Tuesday, July 17, 2007 12:35:05 PM

Good Afternoon, Paul, welcome to JavaScript programming!

Click here if you are not Paul

Is it a world to hide virtues in?
William Shakespeare

View all quotes

Congratulations, your answer is correct

This page was last modified Tuesday, July 17, 2007 11:05:00 AM

Fig. 11.17 | Online quiz in a child window. (Part 3 of 3.)

Lines 15–22 of this script check the user's answer and output the result to the main window. Lines 16–17 use `window.opener` to write to the main window. The property **window.opener** always contains a reference to the window that opened the current window, if such a window exists. Lines 16–17 write to property `window.opener.document.getElementById("quizSpot").innerHTML`. Recall that `quizSpot` is the `id` of the paragraph in the main window that contains the link to open the quiz. Property **innerHTML** refers to the HTML code inside the quizSpot paragraph (i.e., the code between <p> and </p>). Modifying the innerHTML property dynamically changes the XHTML code in the paragraph. Thus, when lines 16–17 execute, the link in the main window disappears, and the string "Congratulations, your answer is correct." appears. Lines 19–22 modify window.opener.document.getElementById("quizSpot").innerHTML. Lines 19–22 use the same technique to display "Your answer is incorrect. Please try again", followed by a link to try the quiz again.

After checking the quiz answer, the script gives focus to the main window (i.e., puts the main window in the foreground, on top of any other open browser windows), using the method **focus** of the main window's window object. The property window.opener references the main window, so window.opener.focus() (line 24) gives the main window focus, allowing the user to see the changes made to the text of the main window's quizSpot paragraph. Finally, the script closes the quiz window, using method window.close (line 25).

Lines 28–29 close the script and head elements of the XHTML document. Line 30 opens the body of the XHTML document. The body contains the form, image, text labels and radio buttons that comprise the quiz. Lines 52–54 close the form, the body and the XHTML document.

11.11 Using JSON to Represent Objects

In 1999, JSON (JavaScript Object Notation)—a simple way to represent JavaScript objects as strings—was introduced as an alternative to XML as a data-exchange technique. JSON has gained acclaim due to its simple format, making objects easy to read, create and parse. Each JSON object is represented as a list of property names and values contained in curly braces, in the following format:

{ *propertyName1* : *value1*, *propertyName2* : *value2* }

Arrays are represented in JSON with square brackets in the following format:

[*value1*, *value2*, *value3*]

Each value can be a string, a number, a JSON object, true, false or null. To appreciate the simplicity of JSON data, examine this representation of an array of address-book entries from Chapter 15:

```
[ { first: 'Cheryl', last: 'Black' },
  { first: 'James', last: 'Blue' },
  { first: 'Mike', last: 'Brown' },
  { first: 'Meg', last: 'Gold' } ]
```

JSON provides a straightforward way to manipulate objects in JavaScript, and many other programming languages now support this format. In addition to simplifying object creation, JSON allows programs to extract data easily and to efficiently transmit data across the Internet. JSON integrates especially well with Ajax applications, discussed in Chapter 15. See Section 15.7 for a more detailed discussion of JSON, as well as an Ajax-specific example. For more information on JSON, visit our JSON Resource Center at www.deitel.com/json.

11.12 Wrap-Up

This chapter provided an introduction to Object Technology, many of JavaScript's built-in objects, and a brief introduction to JSON, a simple way to represent new JavaScript objects. We introduced vocabulary and concepts integral to object-oriented and object-based programming. We took a closer look at some methods and properties of the `Math`, `Date`, `Boolean`, `Number`, `String`, `document`, and `window` objects. In the next chapter, we introduce the Document Object Model, a set of JavaScript objects that represent the elements in a web page that allows you to dynamically modify the page's content.

11.13 Web Resources

`www.deitel.com/javascript/`

The Deitel JavaScript Resource Center contains links to some of the best JavaScript resources on the web. There you'll find categorized links to JavaScript tools, code generators, forums, books, libraries, frameworks and more. Also check out the tutorials for all skill levels, from introductory to advanced. Be sure to visit the related Resource Centers on XHTML (`www.deitel.com/xhtml/`) and CSS 2.1 (`www.deitel.com/css21/`).

Summary

Section 11.1 Introduction

- The chapter describes several of JavaScript's built-in objects, which will serve as a basis for understanding browser objects in the chapters on Dynamic HTML.

Section 11.2 Introduction to Object Technology

- Objects are a natural way of thinking about the world and about scripts that manipulate XHTML documents.
- JavaScript uses objects to perform many tasks and therefore is referred to as an object-based programming language.
- Objects have attributes and exhibit behaviors.
- Object-oriented design (OOD) models software in terms similar to those that people use to describe real-world objects. It takes advantage of class relationships, where objects of a certain class, such as a class of vehicles, have the same characteristics.
- OOD takes advantage of inheritance relationships, where new classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own.
- Object-oriented design provides a natural and intuitive way to view the software design process—namely, modeling objects by their attributes, behaviors and interrelationships just as we describe real-world objects.
- OOD also models communication between objects.
- OOD encapsulates attributes and operations (behaviors) into objects.
- Objects have the property of information hiding. This means that objects may know how to communicate with one another across well-defined interfaces, but normally they are not allowed to know how other objects are implemented—implementation details are hidden within the objects themselves.
- The designers of web browsers have defined a set of objects that encapsulate an XHTML document's elements and expose to a JavaScript programmer the attributes and behaviors that enable a JavaScript program to interact with (or script) those elements (objects).

- Some programming languages—like Java, Visual Basic, C# and C++—are object oriented. Programming in such a language is called object-oriented programming (OOP), and it allows computer programmers to implement object-oriented designs as working software systems.
- Languages like C are procedural, so programming tends to be action oriented.
- In procedural languages, the unit of programming is the function.
- In object-oriented languages, the unit of programming is the class from which objects are eventually instantiated. Classes contain functions that implement operations and data that implements attributes.
- Procedural programmers concentrate on writing functions. Programmers group actions that perform some common task into functions, and group functions to form programs.
- Object-oriented programmers concentrate on creating their own user-defined types called classes. Each class contains data as well as the set of functions that manipulate that data and provide services to clients.
- The data components of a class are called properties.
- The function components of a class are called methods.
- The nouns in a system specification help you determine the set of classes from which objects are created that work together to implement the system.
- Classes are to objects as blueprints are to houses.
- Classes can have relationships with other classes. These relationships are called associations.
- Packaging software as classes makes it possible for future software systems to reuse the classes. Groups of related classes are often packaged as reusable components.
- With object technology, you can build much of the new software you'll need by combining existing classes, just as automobile manufacturers combine interchangeable parts. Each new class you create will have the potential to become a valuable software asset that you and other programmers can reuse to speed and enhance the quality of future software development efforts.

Section 11.3 `Math` Object

- `Math` object methods allow you to perform many common mathematical calculations.
- An object's methods are called by writing the name of the object followed by a dot operator (`.`) and the name of the method. In parentheses following the method name is the argument (or a comma-separated list of arguments) to the method.

Section 11.4 `String` Object

- Characters are the fundamental building blocks of JavaScript programs. Every program is composed of a sequence of characters grouped together meaningfully that is interpreted by the computer as a series of instructions used to accomplish a task.
- A string is a series of characters treated as a single unit.
- A string may include letters, digits and various special characters, such as `+`, `-`, `*`, `/`, and `$`.
- JavaScript supports Unicode, which represents a large portion of the world's languages.
- String literals or string constants (often called anonymous `String` objects) are written as a sequence of characters in double quotation marks or single quotation marks.
- Combining strings is called concatenation.
- String method `charAt` returns the character at a specific index in a string. Indices for the characters in a string start at 0 (the first character) and go up to (but do not include) the string's `length` (i.e., if the string contains five characters, the indices are 0 through 4). If the index is outside the bounds of the string, the method returns an empty string.

- `String` method `charCodeAt` returns the Unicode value of the character at a specific index in a string. If the index is outside the bounds of the string, the method returns `NaN`. `String` method `fromCharCode` creates a string from a list of Unicode values.
- `String` method `toLowerCase` returns the lowercase version of a string. `String` method `toUpperCase` returns the uppercase version of a string.
- `String` method `indexOf` determines the location of the first occurrence of its argument in the string used to call the method. If the substring is found, the index at which the first occurrence of the substring begins is returned; otherwise, `-1` is returned. This method receives an optional second argument specifying the index from which to begin the search.
- `String` method `lastIndexOf` determines the location of the last occurrence of its argument in the string used to call the method. If the substring is found, the index at which the last occurrence of the substring begins is returned; otherwise, `-1` is returned. This method receives an optional second argument specifying the index from which to begin the search.
- The process of breaking a string into tokens is called tokenization. Tokens are separated from one another by delimiters, typically white-space characters such as blank, tab, newline and carriage return. Other characters may also be used as delimiters to separate tokens.
- `String` method `split` breaks a string into its component tokens. The argument to method `split` is the delimiter string—the string that determines the end of each token in the original string. Method `split` returns an array of strings containing the tokens.
- `String` method `substring` returns the substring from the starting index (its first argument) up to but not including the ending index (its second argument). If the ending index is greater than the length of the string, the substring returned includes the characters from the starting index to the end of the original string.
- `String` method `anchor` wraps the string that calls the method in XHTML element `<a></a>` with the `name` of the anchor supplied as the argument to the method.
- `String` method `fixed` displays text in a fixed-width font by wrapping the string that calls the method in a `<tt></tt>` XHTML element.
- `String` method `strike` displays struck-out text (i.e., text with a line through it) by wrapping the string that calls the method in a `<strike></strike>` XHTML element.
- `String` method `sub` displays subscript text by wrapping the string that calls the method in a `<sub></sub>` XHTML element.
- `String` method `sup` displays superscript text by wrapping the string that calls the method in a `<sup></sup>` XHTML element.
- `String` method `link` creates a hyperlink by wrapping the string that calls the method in XHTML element `<a></a>`. The target of the hyperlink (i.e, value of the `href` property) is the argument to the method and can be any URL.

Section 11.5 `Date` *Object*

- JavaScript's `Date` object provides methods for date and time manipulations.
- Date and time processing can be performed based either on the computer's local time zone or on World Time Standard's Coordinated Universal Time (abbreviated UTC)—formerly called Greenwich Mean Time (GMT).
- Most methods of the `Date` object have a local time zone and a UTC version.
- `Date` method `parse` receives as its argument a string representing a date and time and returns the number of milliseconds between midnight, January 1, 1970, and the specified date and time.

- `Date` method `UTC` returns the number of milliseconds between midnight, January 1, 1970, and the date and time specified as its arguments. The arguments to the `UTC` method include the required year, month and date, and the optional hours, minutes, seconds and milliseconds. If any of the hours, minutes, seconds or milliseconds arguments is not specified, a zero is supplied in its place. For the hours, minutes and seconds arguments, if the argument to the right of any of these arguments is specified, that argument must also be specified (e.g., if the minutes argument is specified, the hours argument must be specified; if the milliseconds argument is specified, all the arguments must be specified).

Section 11.6 `Boolean` *and* `Number` *Objects*

- JavaScript provides the `Boolean` and `Number` objects as object wrappers for boolean `true`/`false` values and numbers, respectively.
- When a boolean value is required in a JavaScript program, JavaScript automatically creates a `Boolean` object to store the value.
- JavaScript programmers can create `Boolean` objects explicitly with the statement

```
var b = new Boolean( booleanValue );
```

  The argument *booleanValue* specifies the value of the `Boolean` object (`true` or `false`). If *booleanValue* is `false`, `0`, `null`, `Number.NaN` or the empty string (`""`), or if no argument is supplied, the new `Boolean` object contains `false`. Otherwise, the new `Boolean` object contains `true`.
- JavaScript automatically creates `Number` objects to store numeric values in a JavaScript program.
- JavaScript programmers can create a `Number` object with the statement

```
var n = new Number( numericValue );
```

  The argument *numericValue* is the number to store in the object. Although you can explicitly create `Number` objects, normally they are created when needed by the JavaScript interpreter.

Section 11.7 `document` *Object*

- JavaScript provides the `document` object for manipulating the document that is currently visible in the browser window.

Section 11.8 `window` *Object*

- JavaScript's `window` object provides methods for manipulating browser windows.

Section 11.9 Using Cookies

- A cookie is a piece of data that is stored on the user's computer to maintain information about the client during and between browser sessions.
- Cookies are accessible in JavaScript through the `document` object's `cookie` property.
- A cookie has the syntax "*identifier=value*," where *identifier* is any valid JavaScript variable identifier, and *value* is the value of the cookie variable. When multiple cookies exist for one website, *identifier-value* pairs are separated by semicolons in the `document.cookie` string.
- The `expires` property in a cookie string sets an expiration date, after which the web browser deletes the cookie. If a cookie's expiration date is not set, then the cookie expires by default after the user closes the browser window. A cookie can be deleted immediately by setting the `expires` property to a date and time in the past.
- The assignment operator does not overwrite the entire list of cookies, but appends a cookie to the end of it.

Section 11.10 Final JavaScript Example

- `window.opener` always contains a reference to the window that opened the current window.
- The property `innerHTML` refers to the HTML code inside the current paragraph element.
- Method `focus` puts the window it references on top of all the others.
- The `window` object's `close` method closes the browser window represented by the `window` object.

Section 11.11 Using JSON to Represent Objects

- JSON (JavaScript Object Notation) is a simple way to represent JavaScript objects as strings.
- JSON was introduced in 1999 as an alternative to XML for data exchange.
- Each JSON object is represented as a list of property names and values contained in curly braces, in the following format:

  `{` *propertyName1* `:` *value1*`,` *propertyName2* `:` *value2* `}`

- Arrays are represented in JSON with square brackets in the following format:

  `[` *value1*`,` *value2*`,` *value3* `]`

- Values in JSON can be strings, numbers, JSON objects, `true`, `false` or `null`.

Terminology

`abs` method of `Math`
abstraction
action-oriented programming language
`anchor` method of `String`
anonymous `String` object
association
attribute (property)
behavior (method)
`blink` method of `String`
`Boolean` object
`ceil` method of `Math`
character
`charAt` method of `String`
`charCodeAt` method of `String`
class
`close` method of `window`
code reuse
components
`concat` method of `String`
cookie
Coordinated Universal Time (UTC)
`cos` method of `Math`
date
`Date` object
delimiter
`document` object
`E` property of `Math`
empty string
encapsulation
`escape` function
`exp` method of `Math`
`fixed` method of `String`
`floor` method of `Math`
`focus` method of `window`
`fromCharCode` method of `String`
`getDate` method of `Date`
`getDay` method of `Date`
`getFullYear` method of `Date`
`getHours` method of `Date`
`getMilliseconds` method of `Date`
`getMinutes` method of `Date`
`getMonth` method of `Date`
`getSeconds` method of `Date`
`getTime` method of `Date`
`getTimezoneOffset` method of `Date`
`getUTCDate` method of `Date`
`getUTCDay` method of `Date`
`getUTCFullYear` method of `Date`
`getUTCHours` method of `Date`
`getUTCMilliSeconds` method of `Date`
`getUTCMinutes` method of `Date`
`getUTCMonth` method of `Date`
`getUTCSeconds` method of `Date`
Greenwich Mean Time (GMT)
hexadecimal escape sequences
hiding
index in a string
`indexOf` method of `String`
information hiding
inheritance

- innerHTML property
- instantiation
- interface
- lastIndexOf method of String
- link method of String
- LN10 property of Math
- LN2 property of Math
- local time zone
- log method of Math
- LOG10E property of Math
- LOG2E property of Math
- Math object
- max method of Math
- MAX_SIZE property of Number
- method
- min method of Math
- MIN_SIZE property of Number
- NaN property of Number
- NEGATIVE_INFINITY property of Number
- Number object
- object
- object wrapper
- object-based programming language
- object-oriented design (OOD)
- object-oriented programming (OOP)
- open method of window
- opener property of window
- operation
- parse method of Date
- PI property of Math
- POSITIVE_INFINITY property of Number
- pow method of Math
- property
- round method of Math
- search a string
- sending a message to an object
- setDate method of Date
- setFullYear method of Date
- setHours method of Date
- setMilliSeconds method of Date
- setMinutes method of Date
- setMonth method of Date
- setSeconds method of Date
- setTime method of Date
- setUTCDate method of Date
- setUTCFullYear method of Date
- setUTCHours method of Date
- setUTCMilliseconds method of Date
- setUTCMinutes method of Date
- setUTCMonth method of Date
- setUTCSeconds method of Date
- sin method of Math
- slice method of String
- special characters
- split method of String
- sqrt method of Math
- SQRT1_2 property of Math
- SQRT2 property of Math
- strike method of String
- string
- string constant
- string literal
- String object
- sub method of String
- substr method of String
- substring
- substring method of String
- sup method of String
- tan method of Math
- time
- token
- tokenization
- toLocaleString method of Date
- toLowerCase method of String
- toString method of Date
- toString method of String
- toUpperCase method of String
- toUTCString method of Date
- unescape function
- Unicode
- user-defined type
- UTC (Coordinated Universal Time)
- UTC method of Date
- valueOf method of Boolean
- valueOf method of Date
- valueOf method of Number
- valueOf method of String
- well-defined interfaces
- window object
- wrap in XHTML tags

Self-Review Exercise

11.1 Fill in the blanks in each of the following statements:

a) Because JavaScript uses objects to perform many tasks, JavaScript is commonly referred to as a(n) ________.

b) All objects have ________ and exhibit ________.
c) The methods of the ________ object allow you to perform many common mathematical calculations.
d) Invoking (or calling) a method of an object is referred to as ________.
e) String literals or string constants are written as a sequence of characters in ________ or ________.
f) Indices for the characters in a string start at ________.
g) `String` methods ________ and ________ search for the first and last occurrences of a substring in a `String`, respectively.
h) The process of breaking a string into tokens is called ________.
i) `String` method ________ formats a `String` as a hyperlink.
j) Date and time processing can be performed based on the ________ or on World Time Standard's ________.
k) `Date` method ________ receives as its argument a string representing a date and time, and returns the number of milliseconds between midnight, January 1, 1970, and the specified date and time.

Answers to Self-Review Exercise

11.1 a) object-based programming language. b) attributes, behaviors. c) `Math`. d) sending a message to the object. e) double quotation marks, single quotation marks. f) 0. g) `indexOf`, `lastIndexOf`. h) tokenization. i) `link`. j) computer's local time zone, Coordinated Universal Time (UTC). k) `parse`.

Exercises

11.2 Create a web page that contains four XHTML buttons. Each button, when clicked, should cause an alert dialog to display a different time or date in relation to the current time. Create a `Now` button that alerts the current time and date and a `Yesterday` button that alerts the time and date 24 hours ago. The other two buttons should alert the time and date ten years ago and one week from today.

11.3 Write a script that tests as many of the `Math` library functions in Fig. 11.1 as you can. Exercise each of these functions by having your program display tables of return values for several argument values in an XHTML `textarea`.

11.4 `Math` method `floor` may be used to round a number to a specific decimal place. For example, the statement

```
y = Math.floor( x * 10 + .5 ) / 10;
```

rounds x to the tenths position (the first position to the right of the decimal point). The statement

```
y = Math.floor( x * 100 + .5 ) / 100;
```

rounds `x` to the hundredths position (i.e., the second position to the right of the decimal point). Write a script that defines four functions to round a number `x` in various ways:

a) `roundToInteger( number )`
b) `roundToTenths( number )`
c) `roundToHundredths( number )`
d) `roundToThousandths( number )`

For each value read, your program should display the original value, the number rounded to the nearest integer, the number rounded to the nearest tenth, the number rounded to the nearest hundredth and the number rounded to the nearest thousandth.

11.5 Modify the solution to Exercise 11.4 to use Math method round instead of method floor.

11.6 Write a script that uses relational and equality operators to compare two Strings input by the user through an XHTML form. Output in an XHTML textarea whether the first string is less than, equal to or greater than the second.

11.7 Write a script that uses random number generation to create sentences. Use four arrays of strings called article, noun, verb and preposition. Create a sentence by selecting a word at random from each array in the following order: article, noun, verb, preposition, article and noun. As each word is picked, concatenate it to the previous words in the sentence. The words should be separated by spaces. When the final sentence is output, it should start with a capital letter and end with a period.

The arrays should be filled as follows: the article array should contain the articles "the", "a", "one", "some" and "any"; the noun array should contain the nouns "boy", "girl", "dog", "town" and "car"; the verb array should contain the verbs "drove", "jumped", "ran", "walked" and "skipped"; the preposition array should contain the prepositions "to", "from", "over", "under" and "on".

The program should generate 20 sentences to form a short story and output the result to an XHTML textarea. The story should begin with a line reading "Once upon a time..." and end with a line reading "THE END".

11.8 *(Limericks)* A limerick is a humorous five-line verse in which the first and second lines rhyme with the fifth, and the third line rhymes with the fourth. Using techniques similar to those developed in Exercise 11.7, write a script that produces random limericks. Polishing this program to produce good limericks is a challenging problem, but the result will be worth the effort!

11.9 *(Pig Latin)* Write a script that encodes English-language phrases in pig Latin. Pig Latin is a form of coded language often used for amusement. Many variations exist in the methods used to form pig Latin phrases. For simplicity, use the following algorithm:

To form a pig Latin phrase from an English-language phrase, tokenize the phrase into an array of words using String method split. To translate each English word into a pig Latin word, place the first letter of the English word at the end of the word and add the letters "ay." Thus the word "jump" becomes "umpjay," the word "the" becomes "hetay" and the word "computer" becomes "omputercay." Blanks between words remain as blanks. Assume the following: The English phrase consists of words separated by blanks, there are no punctuation marks and all words have two or more letters. Function printLatinWord should display each word. Each token (i.e., word in the sentence) is passed to method printLatinWord to print the pig Latin word. Enable the user to input the sentence through an XHTML form. Keep a running display of all the converted sentences in an XHTML textarea.

11.10 Write a script that inputs a telephone number as a string in the form (555) 555-5555. The script should use String method split to extract the area code as a token, the first three digits of the phone number as a token and the last four digits of the phone number as a token. Display the area code in one text field and the seven-digit phone number in another text field.

11.11 Write a script that inputs a line of text, tokenizes it with String method split and outputs the tokens in reverse order.

11.12 Write a script that inputs text from an XHTML form and outputs it in uppercase and lowercase letters.

11.13 Write a script that inputs several lines of text and a search character and uses String method indexOf to determine the number of occurrences of the character in the text.

11.14 Write a script based on the program in Exercise 11.13 that inputs several lines of text and uses String method indexOf to determine the total number of occurrences of each letter of the alphabet in the text. Uppercase and lowercase letters should be counted together. Store the totals for each letter in an array, and print the values in tabular format in an XHTML `textarea` after the totals have been determined.

11.15 Write a script that reads a series of strings and outputs in an XHTML `textarea` only those strings beginning with the character "`b`."

11.16 Write a script that reads a series of strings and outputs in an XHTML `textarea` only those strings ending with the characters "`ed`."

11.17 Write a script that inputs an integer code for a character and displays the corresponding character.

11.18 Modify your solution to Exercise 11.17 so that it generates all possible three-digit codes in the range 000 to 255 and attempts to display the corresponding characters. Display the results in an XHTML `textarea`.

11.19 Write your own version of the `String` method `indexOf` and use it in a script.

11.20 Write your own version of the `String` method `lastIndexOf` and use it in a script.

11.21 Write a program that reads a five-letter word from the user and produces all possible three-letter words that can be derived from the letters of the five-letter word. For example, the three-letter words produced from the word "bathe" include the commonly used words "ate," "bat," "bet," "tab," "hat," "the" and "tea." Output the results in an XHTML `textarea`.

11.22 *(Printing Dates in Various Formats)* Dates are printed in several common formats. Write a script that reads a date from an XHTML form and creates a `Date` object in which to store it. Then use the various methods of the `Date` object that convert `Date`s into strings to display the date in several formats.

Special Section: Challenging String-Manipulation Exercises

The preceding exercises are keyed to the text and designed to test the reader's understanding of fundamental string-manipulation concepts. This section includes a collection of intermediate and advanced string-manipulation exercises. The reader should find these problems challenging, yet entertaining. The problems vary considerably in difficulty. Some require an hour or two of program writing and implementation. Others are useful for lab assignments that might require two or three weeks of study and implementation. Some are challenging term projects.

11.23 *(Text Analysis)* The availability of computers with string-manipulation capabilities has resulted in some rather interesting approaches to analyzing the writings of great authors. Much attention has been focused on whether William Shakespeare really wrote the works attributed to him. Some scholars believe there is substantial evidence indicating that Christopher Marlowe actually penned these masterpieces. Researchers have used computers to find similarities in the writings of these two authors. This exercise examines three methods for analyzing texts with a computer.

a) Write a script that reads several lines of text from the keyboard and prints a table indicating the number of occurrences of each letter of the alphabet in the text. For example, the phrase

```
To be, or not to be: that is the question:
```

contains one "a," two "b's," no "c's," etc.

b) Write a script that reads several lines of text and prints a table indicating the number of one-letter words, two-letter words, three-letter words, etc., appearing in the text. For example, the phrase

```
Whether 'tis nobler in the mind to suffer
```

contains

Word length	Occurrences
1	0
2	2
3	1
4	2 (including 'tis)
5	0
6	2
7	1

c) Write a script that reads several lines of text and prints a table indicating the number of occurrences of each different word in the text. The first version of your program should include the words in the table in the same order in which they appear in the text. For example, the lines

```
To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
```

contain the word "to" three times, the word "be" twice, and the word "or" once. A more interesting (and useful) printout should then be attempted in which the words are sorted alphabetically.

11.24 *(Check Protection)* Computers are frequently employed in check-writing systems such as payroll and accounts payable applications. Many strange stories circulate regarding weekly paychecks being printed (by mistake) for amounts in excess of $1 million. Incorrect amounts are printed by computerized check-writing systems because of human error and/or machine failure. Systems designers build controls into their systems to prevent erroneous checks from being issued.

Another serious problem is the intentional alteration of a check amount by someone who intends to cash a check fraudulently. To prevent a dollar amount from being altered, most computerized check-writing systems employ a technique called *check protection.*

Checks designed for imprinting by computer contain a fixed number of spaces in which the computer may print an amount. Suppose a paycheck contains eight blank spaces in which the computer is supposed to print the amount of a weekly paycheck. If the amount is large, then all eight of those spaces will be filled, for example:

```
1,230.60 (check amount)
--------
12345678 (position numbers)
```

On the other hand, if the amount is less than $1000, then several of the spaces would ordinarily be left blank. For example,

```
   99.87
--------
12345678
```

contains three blank spaces. If a check is printed with blank spaces, it is easier for someone to alter the amount of the check. To prevent a check from being altered, many check-writing systems insert *leading asterisks* to protect the amount as follows:

```
***99.87
--------
12345678
```

Write a script that inputs a dollar amount to be printed on a check, then prints the amount in check-protected format with leading asterisks if necessary. Assume that nine spaces are available for printing the amount.

11.25 *(Writing the Word Equivalent of a Check Amount)* Continuing the discussion in the preceding exercise, we reiterate the importance of designing check-writing systems to prevent alteration of check amounts. One common security method requires that the check amount be written both in numbers and spelled out in words. Even if someone is able to alter the numerical amount of the check, it is extremely difficult to change the amount in words.

Many computerized check-writing systems do not print the amount of the check in words. Perhaps the main reason for this omission is the fact that most high-level languages used in commercial applications do not contain adequate string-manipulation features. Another reason is that the logic for writing word equivalents of check amounts is somewhat involved.

Write a script that inputs a numeric check amount and writes the word equivalent of the amount. For example, the amount 112.43 should be written as

```
ONE HUNDRED TWELVE and 43/100
```

11.26 *(Morse Code)* Perhaps the most famous of all coding schemes is the Morse code, developed by Samuel Morse in 1832 for use with the telegraph system. The Morse code assigns a series of dots and dashes to each letter of the alphabet, each digit and a few special characters (e.g., period, comma, colon and semicolon). In sound-oriented systems, the dot represents a short sound and the dash represents a long sound. Other representations of dots and dashes are used with light-oriented systems and signal-flag systems.

Separation between words is indicated by a space or, quite simply, by the absence of a dot or dash. In a sound-oriented system, a space is indicated by a short period of time during which no sound is transmitted. The international version of the Morse code appears in Fig. 11.18.

Write a script that reads an English-language phrase and encodes it in Morse code. Also write a program that reads a phrase in Morse code and converts the phrase into the English-language equivalent. Use one blank between each Morse-coded letter and three blanks between each Morse-coded word.

11.27 *(Metric Conversion Program)* Write a script that will assist the user with metric conversions. Your program should allow the user to specify the names of the units as strings (e.g., centimeters, liters, grams, for the metric system and inches, quarts, pounds, for the English system) and should respond to simple questions such as

```
"How many inches are in 2 meters?"
"How many liters are in 10 quarts?"
```

Your program should recognize invalid conversions. For example, the question

```
"How many feet in 5 kilograms?"
```

is not a meaningful question because `"feet"` is a unit of length whereas `"kilograms"` is a unit of mass.

11.28 *(Project: A Spell Checker)* Many popular word-processing software packages have built-in spell checkers.

In this project, you are asked to develop your own spell-checker utility. We make suggestions to help get you started. You should then consider adding more capabilities. Use a computerized dictionary (if you have access to one) as a source of words.

Why do we type so many words with incorrect spellings? In some cases, it is because we simply do not know the correct spelling, so we make a best guess. In some cases, it is because we transpose two letters (e.g., “defualt” instead of “default”). Sometimes we double-type a letter

Character	Code	Character	Code
A	.-	T	-
B	-...	U	..-
C	-.-.	V	...-
D	-..	W	.--
E	.	X	-..-
F	..-.	Y	-.--
G	--.	Z	--..
H	....		
I	..	*Digits*	
J	.---	1	.----
K	-.-	2	..---
L	.-..	3	...--
M	--	4	....-
N	-.	5	.....
O	---	6	-....
P	.--.	7	--...
Q	--.-	8	---..
R	.-.	9	----.
S	...	0	-----

Fig. 11.18 | Letters of the alphabet as expressed in international Morse code.

accidentally (e.g., "hanndy" instead of "handy"). Sometimes we type a nearby key instead of the one we intended (e.g., "biryhday" instead of "birthday"). And so on.

Design and implement a spell-checker application in JavaScript. Your program should maintain an array `wordList` of strings. Enable the user to enter these strings.

Your program should ask a user to enter a word. The program should then look up the word in the `wordList` array. If the word is present in the array, your program should print "`Word is spelled correctly`."

If the word is not present in the array, your program should print "`word is not spelled correctly`." Then your program should try to locate other words in `wordList` that might be the word the user intended to type. For example, you can try all possible single transpositions of adjacent letters to discover that the word "default" is a direct match to a word in `wordList`. Of course, this implies that your program will check all other single transpositions, such as "edfault," "dfeault," "deafult," "defalut" and "defautl." When you find a new word that matches one in `wordList`, print that word in a message, such as "`Did you mean "default?"`"

Implement other tests, such as replacing each double letter with a single letter and any other tests you can develop, to improve the value of your spell checker.

11.29 *(Project: Crossword Puzzle Generator)* Most people have worked a crossword puzzle, but few have ever attempted to generate one. Generating a crossword puzzle is suggested here as a string-manipulation project requiring substantial sophistication and effort.

There are many issues you must resolve to get even the simplest crossword puzzle generator program working. For example, how does one represent the grid of a crossword puzzle in the computer? Should one use a series of strings, or use double-subscripted arrays?

You need a source of words (i.e., a computerized dictionary) that can be directly referenced by the program. In what form should these words be stored to facilitate the complex manipulations required by the program?

The really ambitious reader will want to generate the clues portion of the puzzle, in which the brief hints for each across word and each down word are printed for the puzzle worker. Merely printing a version of the blank puzzle itself is not a simple problem.

12

Document Object Model (DOM): Objects and Collections

Our children may learn about heroes of the past. Our task is to make ourselves architects of the future.

—Jomo Mzee Kenyatta

Though leaves are many, the root is one.

—William Butler Yeats

The thing that impresses me most about America is the way parents obey their children.

—Duke of Windsor

Most of us become parents long before we have stopped being children.

—Mignon McLaughlin

To write it, it took three months; to conceive it three minutes; to collect the data in it—all my life.

—F. Scott Fitzgerald

Sibling rivalry is inevitable. The only sure way to avoid it is to have one child.

—Nancy Samalin

OBJECTIVES

In this chapter you will learn:

- How to use JavaScript and the W3C Document Object Model to create dynamic web pages.
- The concept of DOM nodes and DOM trees.
- How to traverse, edit and modify elements in an XHTML document.
- How to change CSS styles dynamically.
- To create JavaScript animations.

Outline

12.1 Introduction

In this chapter we introduce the Document Object Model (DOM). The DOM gives you access to all the elements on a web page. Inside the browser, the whole web page—paragraphs, forms, tables, etc.—is represented in an object hierarchy. Using JavaScript, you can create, modify and remove elements in the page dynamically.

Previously, both Internet Explorer and Netscape had different versions of Dynamic HTML, which provided similar functionality to the DOM. However, while they provided many of the same capabilities, these two models were incompatible with each other. In an effort to encourage cross-browser websites, the W3C created the standardized Document Object Model. Firefox 2, Internet Explorer 7, and most other major browsers implement *most* of the features of the W3C DOM.

This chapter begins by formally introducing the concept of DOM nodes and DOM trees. We then discuss properties and methods of DOM nodes and cover additional methods of the `document` object. We also discuss how to dynamically change style properties, which enables you to create many types of effects, such as user-defined background colors and animations. Then, we present a diagram of the extensive object hierarchy, with explanations of the various objects and properties, and we provide links to websites with further information on the topic.

Software Engineering Observation 12.1

With the DOM, XHTML elements can be treated as objects, and many attributes of XHTML elements can be treated as properties of those objects. Then, objects can be scripted (through their `id` *attributes) with JavaScript to achieve dynamic effects.*

12.2 Modeling a Document: DOM Nodes and Trees

As we saw in previous chapters, the `document`'s `getElementById` method is the simplest way to access a specific element in a page. In this section and the next, we discuss more thoroughly the objects returned by this method.

The `getElementById` method returns objects called DOM nodes. Every element in an XHTML page is modeled in the web browser by a DOM node. All the nodes in a document make up the page's DOM tree, which describes the relationships among elements. Nodes are related to each other through child-parent relationships. An XHTML element inside another element is said to be a child of the containing element. The containing element is known as the parent. A node may have multiple children, but only one parent. Nodes with the same parent node are referred to as siblings.

Some browsers have tools that allow you to see a visual representation of the DOM tree of a document. When installing Firefox, you can choose to install a tool called the DOM Inspector, which allows you to view the DOM tree of an XHTML document. To inspect a document, Firefox users can access the **DOM Inspector** from the **Tools** menu of Firefox. If the DOM inspector is not in the menu, run the Firefox installer and choose **Custom** in the **Setup Type** screen, making sure the **DOM Inspector** box is checked in the **Optional Components** window.

Microsoft provides a Developer Toolbar for Internet Explorer that allows you to inspect the DOM tree of a document. The toolbar can be downloaded from Microsoft at `go.microsoft.com/fwlink/?LinkId=92716`. Once the toolbar is installed, restart the browser, then click the » icon at the right of the toolbar and choose **IE Developer Toolbar** from the menu. Figure 12.1 shows an XHTML document and its DOM tree displayed in Firefox's DOM Inspector and in IE's Web Developer Toolbar.

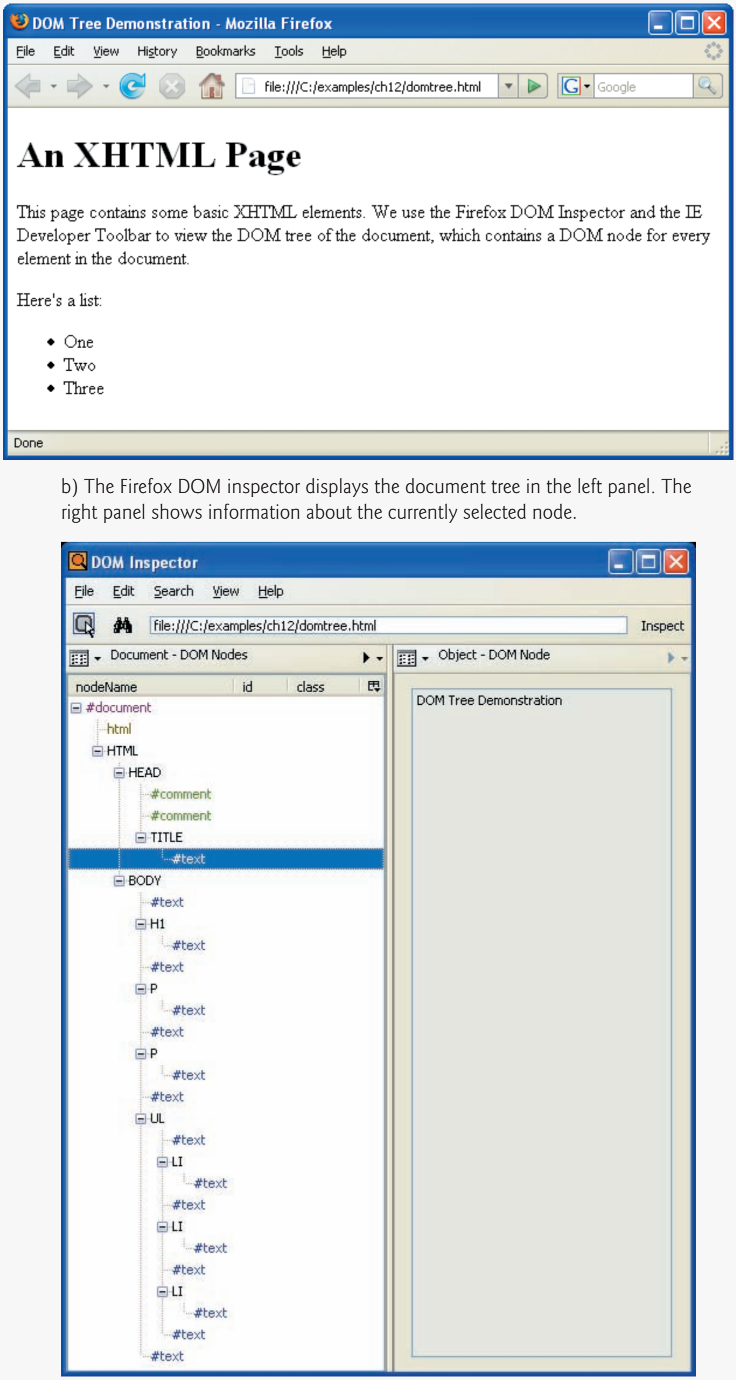
The XHTML document contains a few simple elements. We explain the example based on the Firefox DOM Inspector—the IE Toolbar displays the document with only minor differences. A node can be expanded and collapsed using the **+** and **-** buttons next to the node's name. Figure 12.1(b) shows all the nodes in the document fully expanded. The document node (shown as **#document**) at the top of the tree is called the root node, because it has no parent. Below the document node, the **HTML** node is indented from the document node to signify that the **HTML** node is a child of the **#document** node. The **HTML** node represents the `html` element (lines 7–24).

The **HEAD** and **BODY** nodes are siblings, since they are both children of the **HTML** node. The **HEAD** contains two **#comment** nodes, representing lines 5–6. The **TITLE** node

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 12.1: domtree.html -->
 6  <!-- Demonstration of a document's DOM tree. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>DOM Tree Demonstration</title>
10     </head>
11     <body>
12        <h1>An XHTML Page</h1>
13        <p>This page contains some basic XHTML elements. We use the Firefox
14           DOM Inspector and the IE Developer Toolbar to view the DOM tree
15           of the document, which contains a DOM node for every element in
16           the document.</p>
17        <p>Here's a list:</p>
18        <ul>
19           <li>One</li>
20           <li>Two</li>
21           <li>Three</li>
22        </ul>
23     </body>
24  </html>
```

Fig. 12.1 | Demonstration of a document's DOM tree. (Part 1 of 3.)

a) The XHTML document is rendered in Firefox.



DOM Tree Demonstration - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///C:/examples/ch12/domtree.html
Google
Done

An XHTML Page

This page contains some basic XHTML elements. We use the Firefox DOM Inspector and the IE Developer Toolbar to view the DOM tree of the document, which contains a DOM node for every element in the document.

Here's a list:

- One
- Two
- Three

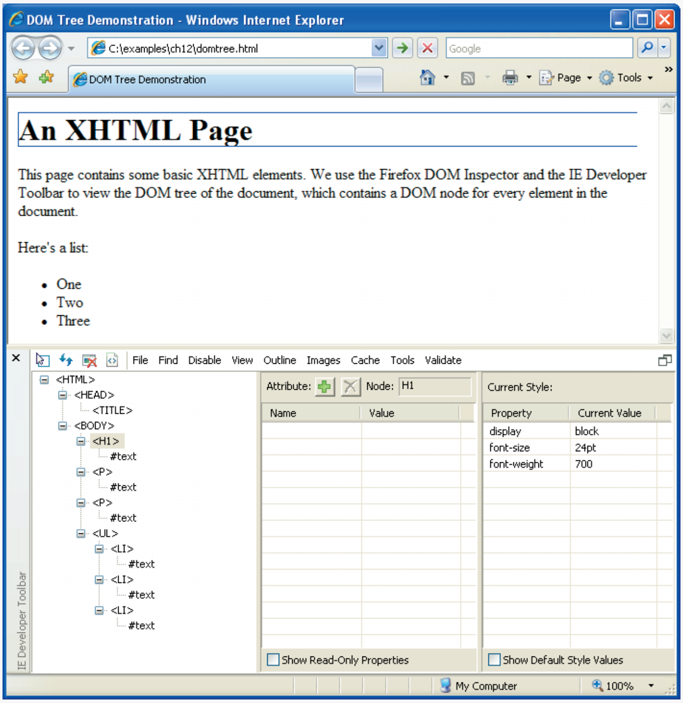
b) The Firefox DOM inspector displays the document tree in the left panel. The right panel shows information about the currently selected node.

```
#document
  html
  HTML
    HEAD
      #comment
      #comment
      TITLE
        #text
    BODY
      #text
      H1
        #text
      #text
      P
        #text
      #text
      P
        #text
      #text
      UL
        #text
        LI
          #text
        #text
        LI
          #text
        #text
        LI
          #text
        #text
      #text
```

DOM Tree Demonstration

Fig. 12.1 | Demonstration of a document's DOM tree. (Part 2 of 3.)

c) The Internet Explorer Web Developer Toolbar displays much of the same information as the DOM inspector in Firefox in a panel at the bottom of the browser window.



An XHTML Page

This page contains some basic XHTML elements. We use the Firefox DOM Inspector and the IE Developer Toolbar to view the DOM tree of the document, which contains a DOM node for every element in the document.

Here's a list:

- One
- Two
- Three

```
<HTML>
  <HEAD>
    <TITLE>
  <BODY>
    <H1>
      #text
    <P>
      #text
    <P>
      #text
    <UL>
      <LI>
        #text
      <LI>
        #text
      <LI>
        #text
```

Property	Current Value
display	block
font-size	24pt
font-weight	700

Fig. 12.1 | Demonstration of a document's DOM tree. (Part 3 of 3.)

has a child text node (**#text**) containing the text **DOM Tree Demonstration**, visible in the right pane of the DOM inspector when the text node is selected. The **BODY** node contains nodes representing each of the elements in the page. Note that the **LI** nodes are children of the **UL** node, since they are nested inside it.

Also, notice that, in addition to the text nodes representing the text inside the body, paragraphs and list elements, a number of other text nodes appear in the document. These text nodes contain nothing but white space. When Firefox parses an XHTML document into a DOM tree, the white space between sibling elements is interpreted as text and placed inside text nodes. Internet Explorer ignores white space and does not convert it into empty text nodes. If you run this example on your own computer, you will notice that the **BODY** node has a **#comment** child node not present above in both the Firefox and Internet Explorer DOM trees. This is a result of the copyright line at the end of the posted file.

This section introduced the concept of DOM nodes and DOM trees. The next section discusses DOM nodes in more detail, discussing methods and properties of DOM nodes that allow you to modify the DOM tree of a document using JavaScript.

12.3 Traversing and Modifying a DOM Tree

The DOM gives you access to the elements of a document, allowing you to modify the contents of a page dynamically using event-driven JavaScript. This section introduces properties and methods of all DOM nodes that enable you to traverse the DOM tree, modify nodes and create or delete content dynamically.

Figure 12.2 showcases some of the functionality of DOM nodes, as well as two additional methods of the `document` object. The program allows you to highlight, modify, insert and remove elements.

Lines 117–132 contain basic XHTML elements and content. Each element has an `id` attribute, which is also displayed at the beginning of the element in square brackets. For example, the `id` of the `h1` element in lines 117–118 is set to `bigheading`, and the heading text begins with `[bigheading]`. This allows the user to see the `id` of each element in the page. The body also contains an `h3` heading, several `p` elements, and an unordered list.

A `div` element (lines 133–162) contains the remainder of the XHTML `body`. Line 134 begins a `form` element, assigning the empty string to the required `action` attribute (because we're not submitting to a server) and returning `false` to the `onsubmit` attribute. When a `form`'s `onsubmit` handler returns `false`, the navigation to the address specified in the `action` attribute is aborted. This allows us to modify the page using JavaScript event handlers without reloading the original, unmodified XHTML.

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 12.2: dom.html -->
 6  <!-- Basic DOM functionality. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Basic DOM Functionality</title>
10        <style type = "text/css">
11           h1, h3       { text-align: center;
12                          font-family: tahoma, geneva, sans-serif }
13           p            { margin-left: 5%;
14                          margin-right: 5%;
15                          font-family: arial, helvetica, sans-serif }
16           ul           { margin-left: 10% }
17           a            { text-decoration: none }
18           a:hover      { text-decoration: underline }
19           .nav         { width: 100%;
20                          border-top: 3px dashed blue;
21                          padding-top: 10px }
22           .highlighted { background-color: yellow }
23           .submit      { width: 120px }
24        </style>
25        <script type = "text/javascript">
26           <!--
27           var currentNode; // stores the currently highlighted node
28           var idcount = 0; // used to assign a unique id to new elements
29
30           // get and highlight an element by its id attribute
31           function byId()
32           {
```

Fig. 12.2 | Basic DOM functionality. (Part 1 of 8.)

```
            var id = document.getElementById( "gbi" ).value;
            var target = document.getElementById( id );

            if ( target )
               switchTo( target );
         } // end function byId

         // insert a paragraph element before the current element
         // using the insertBefore method
         function insert()
         {
            var newNode = createNewNode(
               document.getElementById( "ins" ).value );
            currentNode.parentNode.insertBefore( newNode, currentNode );
            switchTo( newNode );
         } // end function insert

         // append a paragraph node as the child of the current node
         function appendNode()
         {
            var newNode = createNewNode(
               document.getElementById( "append" ).value );
            currentNode.appendChild( newNode );
            switchTo( newNode );
         } // end function appendNode

         // replace the currently selected node with a paragraph node
         function replaceCurrent()
         {
            var newNode = createNewNode(
               document.getElementById( "replace" ).value );
            currentNode.parentNode.replaceChild( newNode, currentNode );
            switchTo( newNode );
         } // end function replaceCurrent

         // remove the current node
         function remove()
         {
            if ( currentNode.parentNode == document.body )
               alert( "Can't remove a top-level element." );
            else
            {
               var oldNode = currentNode;
               switchTo( oldNode.parentNode );
               currentNode.removeChild( oldNode );
            }
         } // end function remove

         // get and highlight the parent of the current node
         function parent()
         {
            var target = currentNode.parentNode;

```

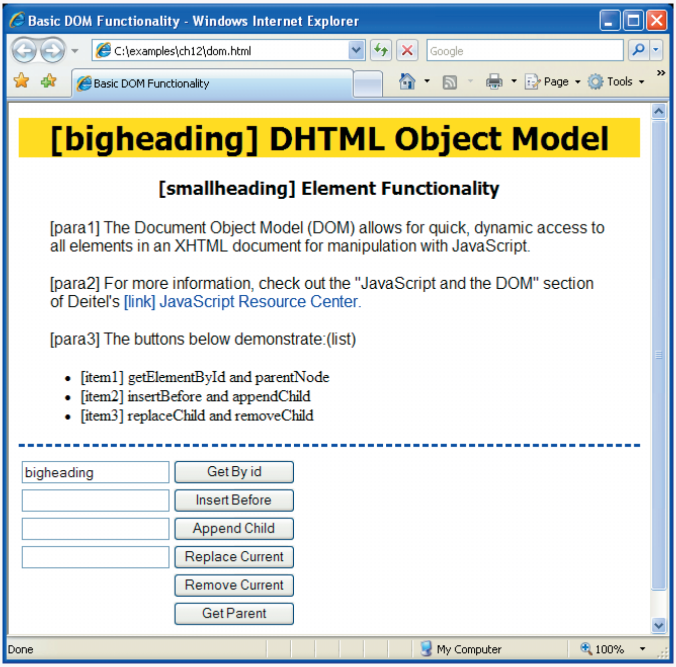
Fig. 12.2 | Basic DOM functionality. (Part 2 of 8.)

```
            if ( target != document.body )
               switchTo( target );
            else
               alert( "No parent." );
         } // end function parent

         // helper function that returns a new paragraph node containing
         // a unique id and the given text
         function createNewNode( text )
         {
            var newNode = document.createElement( "p" );
            nodeId = "new" + idcount;
            ++idcount;
            newNode.id = nodeId;
            text = "[" + nodeId + "] " + text;
            newNode.appendChild(document.createTextNode( text ) );
            return newNode;
         } // end function createNewNode

         // helper function that switches to a new currentNode
         function switchTo( newNode )
         {
            currentNode.className = ""; // remove old highlighting
            currentNode = newNode;
            currentNode.className = "highlighted"; // highlight new node
            document.getElementById( "gbi" ).value = currentNode.id;
         } // end function switchTo
         // -->
      </script>
   </head>
   <body onload = "currentNode = document.getElementById( 'bigheading' )">
      <h1 id = "bigheading" class = "highlighted">
         [bigheading] DHTML Object Model</h1>
      <h3 id = "smallheading">[smallheading] Element Functionality</h3>
      <p id = "para1">[para1] The Document Object Model (DOM) allows for
         quick, dynamic access to all elements in an XHTML document for
         manipulation with JavaScript.</p>
      <p id = "para2">[para2] For more information, check out the
         "JavaScript and the DOM" section of Deitel's
         <a id = "link" href = "http://www.deitel.com/javascript">
            [link] JavaScript Resource Center.</a></p>
      <p id = "para3">[para3] The buttons below demonstrate:(list)</p>
      <ul id = "list">
         <li id = "item1">[item1] getElementById and parentNode</li>
         <li id = "item2">[item2] insertBefore and appendChild</li>
         <li id = "item3">[item3] replaceChild and removeChild </li>
      </ul>
      <div id = "nav" class = "nav">
         <form onsubmit = "return false" action = "">
            <table>
               <tr>
                  <td><input type = "text" id = "gbi"
                     value = "bigheading" /></td>
```

Fig. 12.2 | Basic DOM functionality. (Part 3 of 8.)

```
                  <td><input type = "submit" value = "Get By id"
                     onclick = "byId()" class = "submit" /></td>
               </tr><tr>
                  <td><input type = "text" id = "ins" /></td>
                  <td><input type = "submit" value = "Insert Before"
                     onclick = "insert()" class = "submit" /></td>
               </tr><tr>
                  <td><input type = "text" id = "append" /></td>
                  <td><input type = "submit" value = "Append Child"
                     onclick = "appendNode()" class = "submit" /></td>
               </tr><tr>
                  <td><input type = "text" id = "replace" /></td>
                  <td><input type = "submit" value = "Replace Current"
                     onclick = "replaceCurrent()" class = "submit" /></td>
               </tr><tr><td />
                  <td><input type = "submit" value = "Remove Current"
                     onclick = "remove()" class = "submit" /></td>
               </tr><tr><td />
                  <td><input type = "submit" value = "Get Parent"
                     onclick = "parent()" class = "submit" /></td>
               </tr>
            </table>
         </form>
      </div>
   </body>
</html>
```

a) This is the page when it first loads. It begins with the large heading highlighted.


Basic DOM Functionality - Windows Internet Explorer
C:\examples\ch12\dom.html
Google
Basic DOM Functionality
Page
Tools
Done
My Computer
100%

[bigheading] DHTML Object Model

[smallheading] Element Functionality

[para1] The Document Object Model (DOM) allows for quick, dynamic access to all elements in an XHTML document for manipulation with JavaScript.

[para2] For more information, check out the "JavaScript and the DOM" section of Deitel's [link] JavaScript Resource Center.

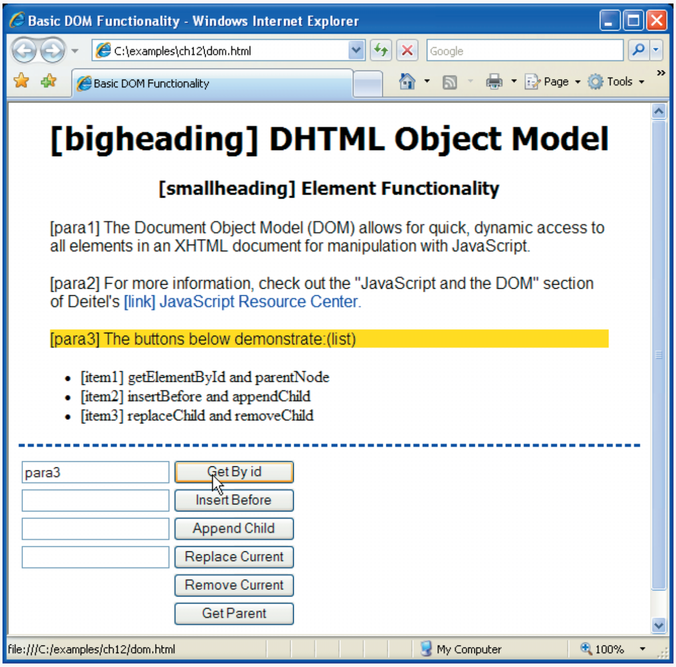
[para3] The buttons below demonstrate:(list)

- [item1] getElementById and parentNode
- [item2] insertBefore and appendChild
- [item3] replaceChild and removeChild

bigheading	Get By id
	Insert Before
	Append Child
	Replace Current
	Remove Current
	Get Parent

Fig. 12.2 | Basic DOM functionality. (Part 4 of 8.)

b) This is the document after using the Get By id button to select para3.



[bigheading] DHTML Object Model

[smallheading] Element Functionality

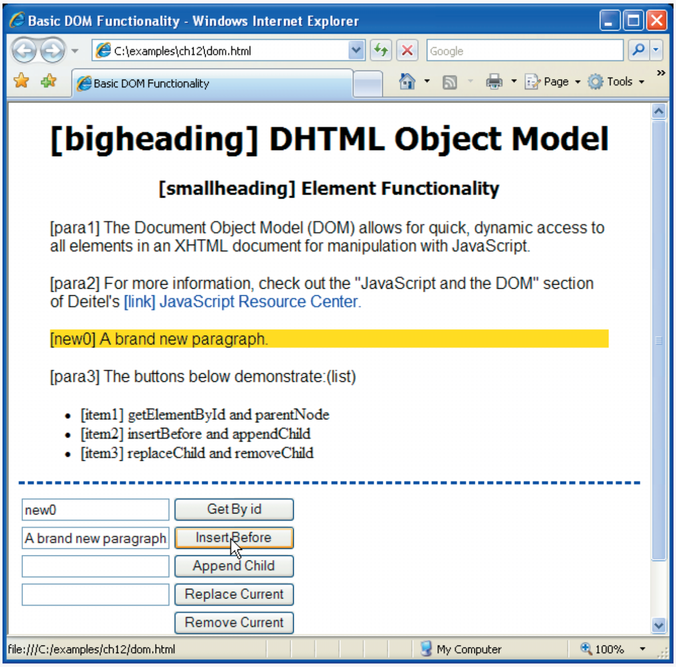
[para1] The Document Object Model (DOM) allows for quick, dynamic access to all elements in an XHTML document for manipulation with JavaScript.

[para2] For more information, check out the "JavaScript and the DOM" section of Deitel's [link] JavaScript Resource Center.

[para3] The buttons below demonstrate:(list)

- [item1] getElementById and parentNode
- [item2] insertBefore and appendChild
- [item3] replaceChild and removeChild

c) This is the document after inserting a new paragraph before the selected one.



[bigheading] DHTML Object Model

[smallheading] Element Functionality

[para1] The Document Object Model (DOM) allows for quick, dynamic access to all elements in an XHTML document for manipulation with JavaScript.

[para2] For more information, check out the "JavaScript and the DOM" section of Deitel's [link] JavaScript Resource Center.

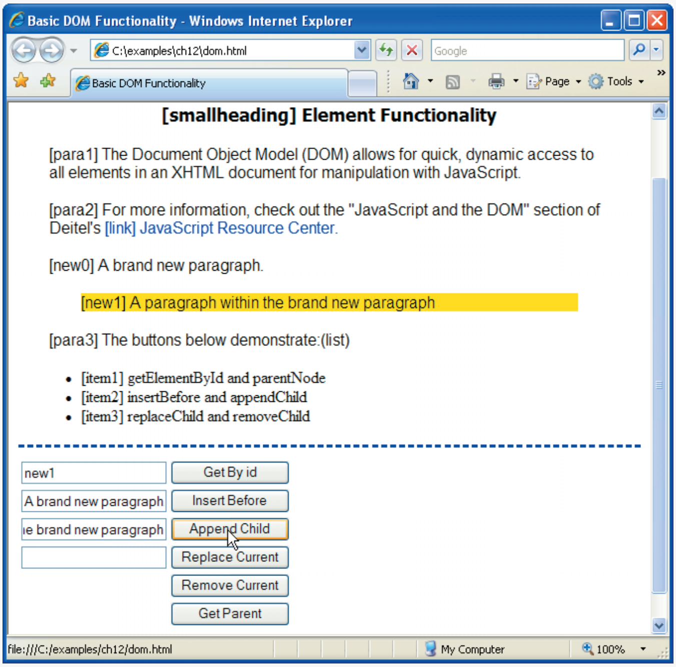
[new0] A brand new paragraph.

[para3] The buttons below demonstrate:(list)

- [item1] getElementById and parentNode
- [item2] insertBefore and appendChild
- [item3] replaceChild and removeChild

Fig. 12.2 | Basic DOM functionality. (Part 5 of 8.)

d) Using the Append Child button, a child paragraph is created.



[smallheading] Element Functionality

[para1] The Document Object Model (DOM) allows for quick, dynamic access to all elements in an XHTML document for manipulation with JavaScript.

[para2] For more information, check out the "JavaScript and the DOM" section of Deitel's [link] JavaScript Resource Center.

[new0] A brand new paragraph.

[new1] A paragraph within the brand new paragraph

[para3] The buttons below demonstrate:(list)

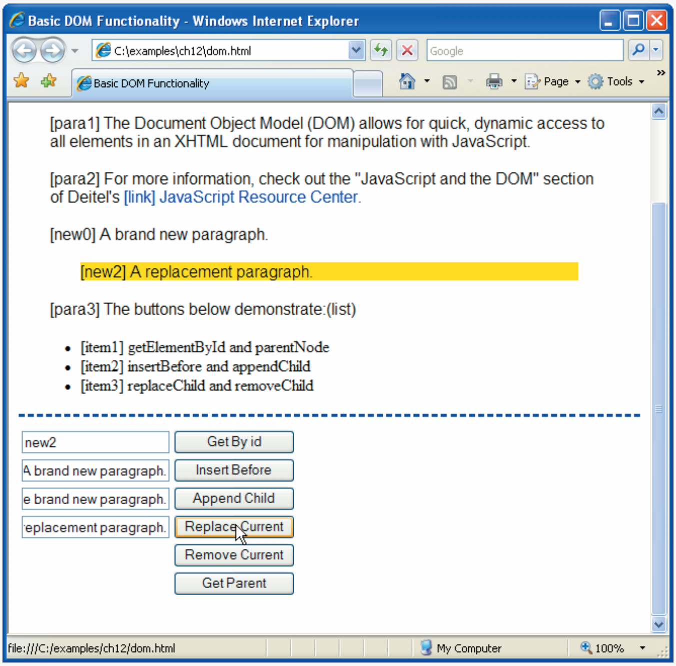
- [item1] getElementById and parentNode
- [item2] insertBefore and appendChild
- [item3] replaceChild and removeChild

new1

A brand new paragraph

ie brand new paragraph

e) The selected paragraph is replaced with a new one.



[para1] The Document Object Model (DOM) allows for quick, dynamic access to all elements in an XHTML document for manipulation with JavaScript.

[para2] For more information, check out the "JavaScript and the DOM" section of Deitel's [link] JavaScript Resource Center.

[new0] A brand new paragraph.

[new2] A replacement paragraph.

[para3] The buttons below demonstrate:(list)

- [item1] getElementById and parentNode
- [item2] insertBefore and appendChild
- [item3] replaceChild and removeChild

new2

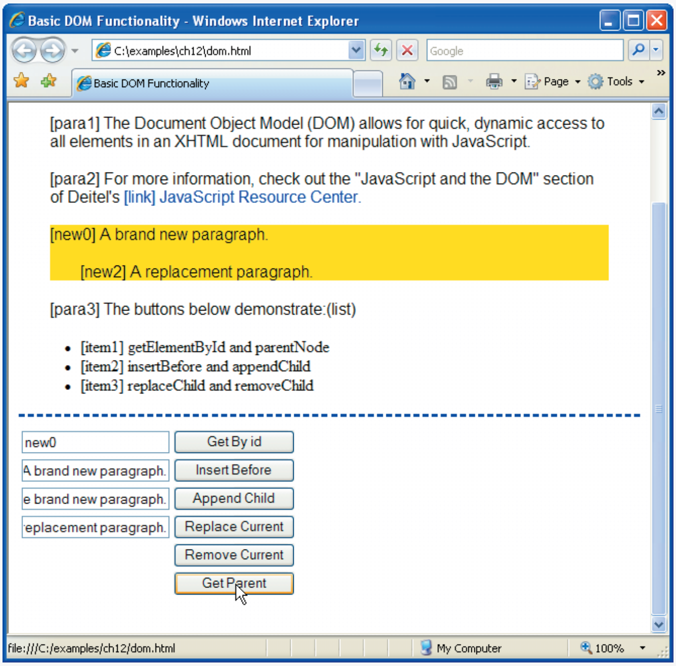
A brand new paragraph.

e brand new paragraph.

eplacement paragraph.

Fig. 12.2 | Basic DOM functionality. (Part 6 of 8.)

f) The Get Parent button gets the parent of the selected node.



[para1] The Document Object Model (DOM) allows for quick, dynamic access to all elements in an XHTML document for manipulation with JavaScript.

[para2] For more information, check out the "JavaScript and the DOM" section of Deitel's [link] JavaScript Resource Center.

[new0] A brand new paragraph.

[new2] A replacement paragraph.

[para3] The buttons below demonstrate:(list)

- [item1] getElementById and parentNode
- [item2] insertBefore and appendChild
- [item3] replaceChild and removeChild

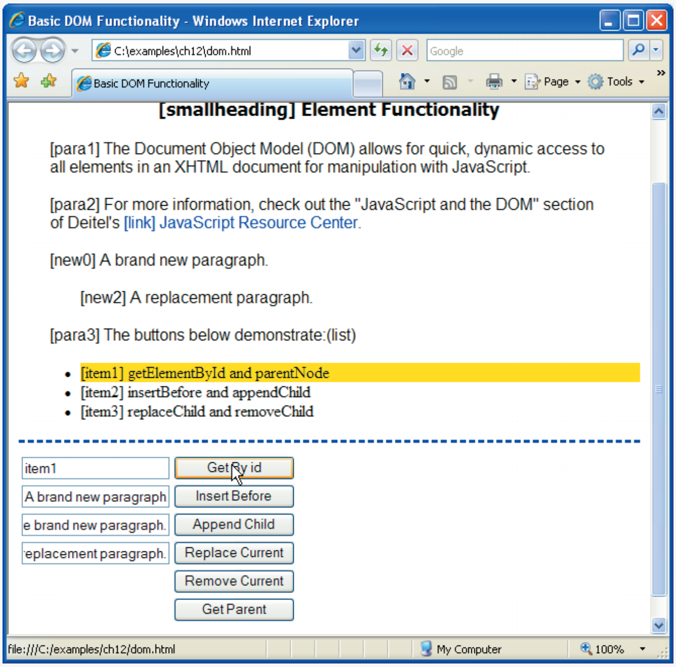
new0

A brand new paragraph.

e brand new paragraph.

eplacement paragraph.

g) Now we select the first list item.



[smallheading] Element Functionality

[para1] The Document Object Model (DOM) allows for quick, dynamic access to all elements in an XHTML document for manipulation with JavaScript.

[para2] For more information, check out the "JavaScript and the DOM" section of Deitel's [link] JavaScript Resource Center.

[new0] A brand new paragraph.

[new2] A replacement paragraph.

[para3] The buttons below demonstrate:(list)

- [item1] getElementById and parentNode
- [item2] insertBefore and appendChild
- [item3] replaceChild and removeChild

item1

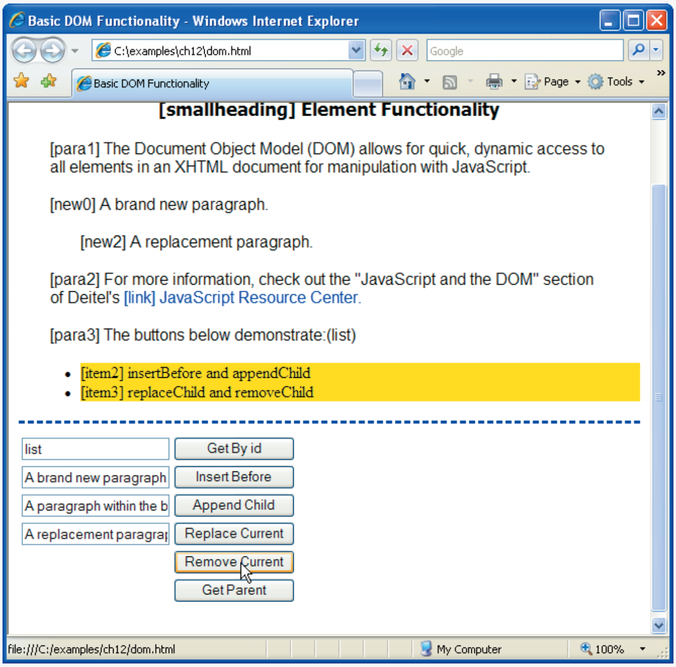
A brand new paragraph

e brand new paragraph.

eplacement paragraph.

Fig. 12.2 | Basic DOM functionality. (Part 7 of 8.)

h) The Remove Current button removes the current node and selects its parent.



[smallheading] Element Functionality

[para1] The Document Object Model (DOM) allows for quick, dynamic access to all elements in an XHTML document for manipulation with JavaScript.

[new0] A brand new paragraph.

[new2] A replacement paragraph.

[para2] For more information, check out the "JavaScript and the DOM" section of Deitel's [link] JavaScript Resource Center.

[para3] The buttons below demonstrate:(list)

- [item2] insertBefore and appendChild
- [item3] replaceChild and removeChild

list

A brand new paragraph

A paragraph within the b

A replacement paragrap

Fig. 12.2 | Basic DOM functionality. (Part 8 of 8.)

A `table` (lines 135–160) contains the controls for modifying and manipulating the elements on the page. Each of the six buttons calls its own event-handling function to perform the action described by its `value`.

The JavaScript code begins by declaring two variables. The variable `currentNode` (line 27) keeps track of the currently highlighted node, because the functionality of the buttons depends on which node is currently selected. The `body`'s `onload` attribute (line 116) initializes `currentNode` to the `h1` element with `id` `bigheading`. Variable `idcount` (line 28) is used to assign a unique `id` to any new elements that are created. The remainder of the JavaScript code contains event handling functions for the XHTML buttons and two helper functions that are called by the event handlers. We now discuss each button and its corresponding event handler in detail.

Finding and Highlighting an Element Using `getElementById` *and* `className`

The first row of the table (lines 136-141) allows the user to enter the `id` of an element into the text field (lines 137–138) and click the `Get By Id` button (lines 139–140) to find and highlight the element, as shown in Fig. 12.2(b) and (g). The `onclick` attribute sets the button's event handler to function `byId`.

The `byId` function is defined in lines 31–38. Line 33 uses `getElementById` to assign the contents of the text field to variable `id`. Line 34 uses `getElementById` again to find the element whose `id` attribute matches the contents of variable `id`, and assign it to variable `target`. If an element is found with the given `id`, `getElementById` returns an object representing that element. If no element is found, `getElementById` returns `null`. Line 36 checks whether `target` is an object—recall that any object used as a boolean expression is `true`, while `null` is `false`. If `target` evaluates to `true`, line 37 calls the `switchTo` function with `target` as its argument.

The `switchTo` function, defined in lines 106–112, is used throughout the program to highlight a new element in the page. The current element is given a yellow background using the style class `highlighted`, defined in line 22. Line 108 sets the current node's `className` property to the empty string. The **`className` property** allows you to change an XHTML element's `class` attribute. In this case, we clear the `class` attribute in order to remove the `highlighted` class from the `currentNode` before we highlight the new one.

Line 109 assigns the `newNode` object (passed into the function as a parameter) to variable `currentNode`. Line 110 adds the `highlighted` style class to the new `currentNode` using the `className` property.

Finally, line 111 uses the **`id` property** to assign the current node's `id` to the input field's value property. Just as `className` allows access to an element's `class` attribute, the `id` property controls an element's `id` attribute. While this isn't necessary when `switchTo` is called by `byId`, we will see shortly that other functions call `switchTo`. This line makes sure that the text field's value is consistent with the currently selected node's `id`. Having found the new element, removed the highlighting from the old element, updated the `currentNode` variable and highlighted the new element, the program has finished selecting a new node by a user-entered `id`.

Creating and Inserting Elements Using `insertBefore` and `appendChild`

The next two table rows allow the user to create a new element and insert it before the current node or as a child of the current node. The second row (lines 141–145) allows the user to enter text into the text field and click the `Insert Before` button. The text is placed in a new paragraph element, which is then inserted into the document before the currently selected element, as in Fig. 12.2(c). The button in lines 143–144 calls the `insert` function, defined in lines 42–48.

Lines 44–45 call the function `createNewNode`, passing it the value of the input field (whose `id` is `ins`) as an argument. Function `createNewNode`, defined in lines 94–103, creates a paragraph node containing the text passed to it. Line 96 creates a `p` element using the `document` object's **`createElement` method**. The `createElement` method creates a new DOM node, taking the tag name as an argument. Note that while `createElement` *creates* an element, it does not *insert* the element on the page.

Line 97 creates a unique `id` for the new element by concatenating `"new"` and the value of `idcount` before incrementing `idcount` in line 98. Line 99 assigns the `id` to the new element. Line 100 concatenates the element's `id` in square brackets to the beginning of `text` (the parameter containing the paragraph's text).

Line 101 introduces two new methods. The `document`'s **`createTextNode` method** creates a node that can contain only text. Given a string argument, `createTextNode` inserts the string into the text node. In line 101, we create a new text node containing the contents of variable `text`. This new node is then used (still in line 101) as the argument to the **`appendChild` method**, which is called on the paragraph node. Method `appendChild` is called on a parent node to insert a child node (passed as an argument) after any existing children.

After the `p` element is created, line 102 returns the node to the calling function `insert`, where it is assigned to variable `newNode` in lines 44–45. Line 46 inserts the newly created node before the currently selected node. The **`parentNode` property** of any DOM node contains the node's parent. In line 46, we use the `parentNode` property of `currentNode` to get its parent.

We call the `insertBefore` method (line 46) on the parent with `newNode` and `currentNode` as its arguments to insert `newNode` as a child of the parent directly before `currentNode`. The general syntax of the **`insertBefore` method** is

parent`.insertBefore(` *newChild*`,` *existingChild* `);`

The method is called on a parent with the new child and an existing child as arguments. The node *newChild* is inserted as a child of *parent* directly before *existingChild*. Line 47 uses the `switchTo` function (discussed earlier in this section) to update the `currentNode` to the newly inserted node and highlight it in the XHTML page.

The third table row (lines 145–149) allows the user to append a new paragraph node as a child of the current element, demonstrated in Fig. 12.2(d). This features uses a similar procedure to the `insertBefore` functionality. Lines 53–54 in function `appendNode` create a new node, line 55 inserts it as a child of the current node, and line 56 uses `switchTo` to update `currentNode` and highlight the new node.

Replacing and Removing Elements Using `replaceChild` *and* `removeChild`

The next two table rows (lines 149–156) allow the user to replace the current element with a new `p` element or simply remove the current element. Lines 150–152 contain a text field and a button that replaces the currently highlighted element with a new paragraph node containing the text in the text field. This feature is demonstrated in Fig. 12.2(e).

The button in lines 151–152 calls function `replaceCurrent`, defined in lines 60–66. Lines 62–63 call `createNewNode`, in the same way as in `insert` and `appendNode`, getting the text from the correct input field. Line 64 gets the parent of `currentNode`, then calls the `replaceChild` method on the parent. The **`replaceChild` method** works as follows:

parent`.replaceChild(` *newChild*`,` *oldChild* `);`

The *parent*'s `replaceChild` method inserts *newChild* into its list of children in place of *oldChild*.

The `Remove Current` feature, shown in Fig. 12.2(h), removes the current element entirely and highlights the parent. No text field is required because a new element is not being created. The button in lines 154-155 calls the `remove` function, defined in lines 69–79. If the node's parent is the body element, line 72 alerts an error—the program does not allow the entire body element to be selected. Otherwise, lines 75–77 remove the current element. Line 75 stores the old `currentNode` in variable `oldNode`. We do this to maintain a reference to the node to be removed after we've changed the value of `currentNode`. Line 76 calls `switchTo` to highlight the parent node.

Line 77 uses the **`removeChild` method** to remove the `oldNode` (a child of the new `currentNode`) from its place in the XHTML document. In general,

parent`.removeChild(` *child* `);`

looks in *parent*'s list of children for *child* and removes it.

The final button (lines 157–158) selects and highlights the parent element of the currently highlighted element by calling the parent function, defined in lines 82–90. Function parent simply gets the parent node (line 84), makes sure it is not the body element, (line 86) and calls switchTo to highlight it (line 87). Line 89 alerts an error if the parent node is the body element. This feature is shown in Fig. 12.2(f).

This section introduced the basics of DOM tree traversal and manipulation. Next, we introduce the concept of collections, which give you access to multiple elements in a page.

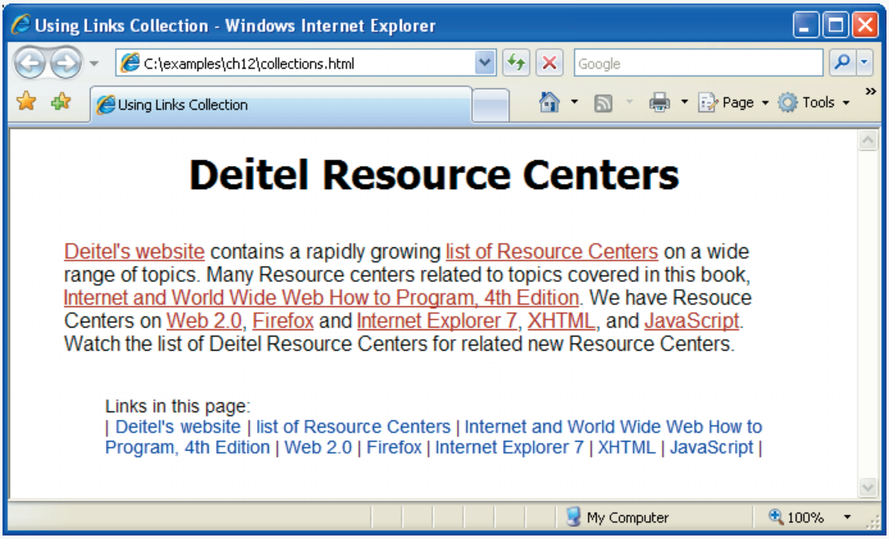
12.4 DOM Collections

Included in the Document Object Model is the notion of **collections**, which are groups of related objects on a page. DOM collections are accessed as properties of DOM objects such as the document object or a DOM node. The document object has properties containing the **images collection**, **links collection**, **forms collection** and **anchors collection**. These collections contain all the elements of the corresponding type on the page. Figure 12.3 gives an example that uses the links collection to extract all of the links on a page and display them together at the bottom of the page.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 12.3: collections.html -->
<!-- Using the links collection. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Using Links Collection</title>
      <style type = "text/css">
         body           { font-family: arial, helvetica, sans-serif }
         h1             { font-family: tahoma, geneva, sans-serif;
                          text-align: center }
         p              { margin: 5% }
         p a            { color: #aa0000 }
         .links         { font-size: 14px;
                          text-align: justify;
                          margin-left: 10%;
                          margin-right: 10% }
         .link a        { text-decoration: none }
         .link a:hover  { text-decoration: underline }
      </style>
      <script type = "text/javascript">
         <!--
         function processlinks()
         {
            var linkslist = document.links; // get the document's links
            var contents = "Links in this page:\n<br />| ";

            // concatenate each link to contents
            for ( var i = 0; i < linkslist.length; i++ )
            {
```

Fig. 12.3 | Using the links collection. (Part 1 of 2.)

```
               var currentLink = linkslist[ i ];
               contents += "<span class = 'link'>" +
                  currentLink.innerHTML.link( currentLink.href ) +
                  "</span> | ";
            } // end for

            document.getElementById( "links" ).innerHTML = contents;
         } // end function processlinks
         // -->
      </script>
   </head>
   <body onload = "processlinks()">
      <h1>Deitel Resource Centers</h1>
      <p><a href = "http://www.deitel.com/">Deitel's website</a> contains
         a rapidly growing
         <a href = "http://www.deitel.com/ResourceCenters.html">list of
         Resource Centers</a> on a wide range of topics. Many Resource
         centers related to topics covered in this book,
         <a href = "http://www.deitel.com/iw3htp4">Internet and World Wide
         Web How to Program, 4th Edition</a>. We have Resouce Centers on
         <a href = "http://www.deitel.com/Web2.0">Web 2.0</a>,
         <a href = "http://www.deitel.com/Firefox">Firefox</a> and
         <a href = "http://www.deitel.com/IE7">Internet Explorer 7</a>,
         <a href = "http://www.deitel.com/XHTML">XHTML</a>, and
         <a href = "http://www.deitel.com/JavaScript">JavaScript</a>.
         Watch the list of Deitel Resource Centers for related new
         Resource Centers.</p>
      <div id = "links" class = "links"></div>
   </body>
</html>
```


Using Links Collection - Windows Internet Explorer
C:\examples\ch12\collections.html
Google
Using Links Collection
Page
Tools
My Computer
100%

Deitel Resource Centers

Deitel's website contains a rapidly growing list of Resource Centers on a wide range of topics. Many Resource centers related to topics covered in this book, Internet and World Wide Web How to Program, 4th Edition. We have Resouce Centers on Web 2.0, Firefox and Internet Explorer 7, XHTML, and JavaScript. Watch the list of Deitel Resource Centers for related new Resource Centers.

Links in this page:
| Deitel's website | list of Resource Centers | Internet and World Wide Web How to Program, 4th Edition | Web 2.0 | Firefox | Internet Explorer 7 | XHTML | JavaScript |

Fig. 12.3 | Using the `links` collection. (Part 2 of 2.)

The XHTML body contains a paragraph (lines 46–59) with links at various places in the text and an empty `div` (line 60) with `id links`. The `body`'s `onload` attribute specifies that the `processlinks` method is called when the `body` finishes loading.

Method `processlinks` declares variable `linkslist` (line 27) to store the `document`'s `links` collection, which is accessed as the `links` property of the `document` object. Line 28 creates the string (`contents`) that will contain all the document's links, to be inserted into the `links div` later. Line 31 begins a `for` statement to iterate through each link. To find the number of elements in the collection, we use the collection's **`length` property**.

Line 33 inside the `for` statement creates a variable (`currentlink`) that stores the current link. Note that we can access the collection stored in `linkslist` using indices in square brackets, just as we did with arrays. DOM collections are stored in objects which have only one property and two methods—the `length` property, the **`item` method** and the **`namedItem` method**. The `item` method—an alternative to the square bracketed indices—can be used to access specific elements in a collection by taking an index as an argument. The `namedItem` method takes a name as a parameter and finds the element in the collection, if any, whose `id` attribute or `name` attribute matches it.

Lines 34–36 add a `span` element to the `contents` string containing the current link. Recall that the `link` method of a string object returns the string as a link to the URL passed to the method. Line 35 uses the `link` method to create an `a` (anchor) element containing the proper text and `href` attribute.

Notice that variable `currentLink` (a DOM node representing an `a` element) has a specialized **`href` property** to refer to the link's `href` attribute. Many types of XHTML elements are represented by special types of nodes that extend the functionality of a basic DOM node. Line 39 inserts the contents into the empty `div` with `id "links"` (line 60) in order to show all the links on the page in one location.

Collections allow easy access to all elements of a single type in a page. This is useful for gathering elements into one place and for applying changes across an entire page. For example, the forms collection could be used to disable all form inputs after a submit button has been pressed to avoid multiple submissions while the next page loads. The next section discusses how to dynamically modify CSS styles using JavaScript and DOM nodes.

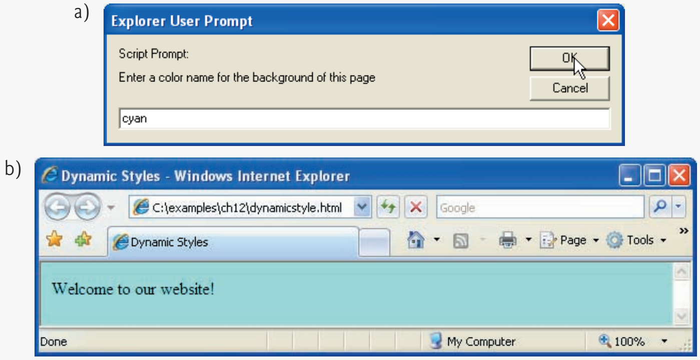
12.5 Dynamic Styles

An element's style can be changed dynamically. Often such a change is made in response to user events, which we discuss in Chapter 13. Such style changes can create many effects, including mouse hover effects, interactive menus, and animations. Figure 12.4 is a simple example that changes the `background-color` style property in response to user input.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 12.4: dynamicstyle.html -->
<!-- Dynamic styles. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
```

Fig. 12.4 | Dynamic styles. (Part 1 of 2.)

```
 9          <title>Dynamic Styles</title>
10          <script type = "text/javascript">
11             <!--
12             function start()
13             {
14                var inputColor = prompt( "Enter a color name for the " +
15                   "background of this page", "" );
16                document.body.style.backgroundColor = inputColor;
17             } // end function start
18             // -->
19          </script>
20       </head>
21       <body id = "body" onload = "start()">
22          <p>Welcome to our website!</p>
23       </body>
24    </html>
```



Script Prompt:

Enter a color name for the background of this page

cyan

C:\examples\ch12\dynamicstyle.html

Dynamic Styles

Welcome to our website!

Fig. 12.4 | Dynamic styles. (Part 2 of 2.)

Function `start` (lines 12–17) `prompts` the user to enter a color name, then sets the background color to that value. [*Note:* An error occurs if the value entered is not a valid color. See Appendix B, XHTML Colors, for further information.] We refer to the background color as `document.body.style.backgroundColor`—the **body property** of the `document` object refers to the `body` element. We then use the `style` property (a property of most XHTML elements) to set the `background-color` CSS property. This is referred to as `backgroundColor` in JavaScript—the hyphen is removed to avoid confusion with the subtraction (-) operator. This naming convention is consistent for most CSS properties. For example, `borderWidth` correlates to the `border-width` CSS property, and `fontFamily` correlates to the `font-family` CSS property. In general, CSS properties are accessed in the format *node*`.style.`*styleproperty*.

Figure 12.5 introduces the `setInterval` and `clearInterval` methods of the `window` object, combining them with dynamic styles to create animated effects. This example is a basic image viewer that allows you to select a Deitel book cover and view it in a larger size. When one of the thumbnail images on the right is clicked, the larger version grows from the top-left corner of the main image area.

The `body` (lines 66–85) contains two `div` elements, both floated `left` using styles defined in lines 14 and 17 in order to present them side by side. The left `div` contains the full-size image iw3htp4.jpg, the cover of this book, which appears when the page loads. The right div contains six thumbnail images which respond to the click event by calling the display method and passing it the filename of the corresponding full-size image.

The display function (lines 46–62) dynamically updates the image in the left div to the one corresponding to the user's click. Lines 48–49 prevent the rest of the function from executing if interval is defined (i.e., an animation is in progress.) Line 51 gets the left div by its id, imgCover. Line 52 creates a new img element. Lines 53–55 set its id to imgCover, set its src to the correct image file in the fullsize directory, and set its required alt attribute. Lines 56–59 do some additional initialization before beginning the animation in line 61. To create the growing animation effect, lines 57–58 set the image width and height to 0. Line 59 replaces the current bigImage node with newNode (created in line 52), and line 60 sets count, the variable that controls the animation, to 0.

Line 61 introduces the window object's **setInterval method**, which starts the animation. This method takes two parameters—a statement to execute repeatedly, and an integer specifying how often to execute it, in milliseconds. We use setInterval to call

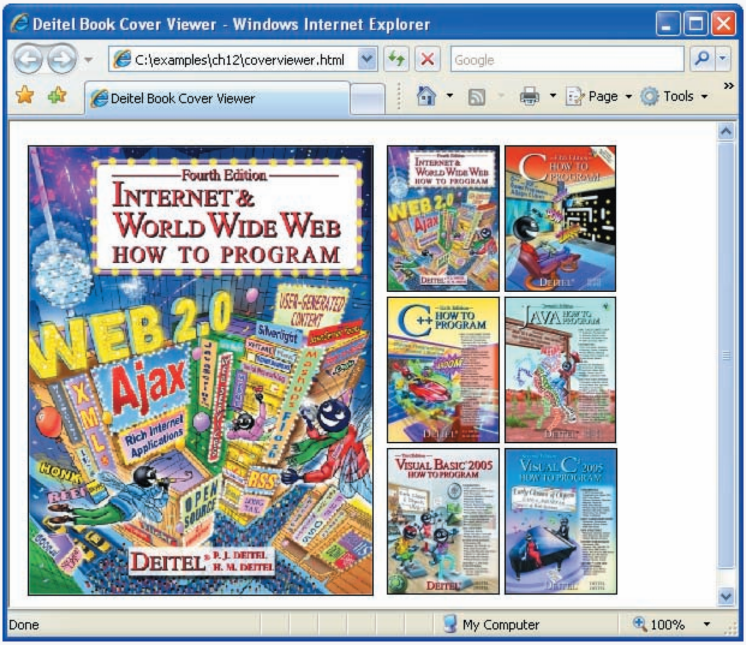
```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 12.5: coverviewer.html -->
 6  <!-- Dynamic styles used for animation. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Deitel Book Cover Viewer</title>
10        <style type = "text/css">
11           .thumbs   { width: 192px;
12                       height: 370px;
13                       padding: 5px;
14                       float: left }
15           .mainimg  { width: 289px;
16                       padding: 5px;
17                       float: left }
18           .imgCover { height: 373px }
19           img       { border: 1px solid black }
20        </style>
21        <script type = "text/javascript">
22           <!--
23           var interval = null; // keeps track of the interval
24           var speed = 6; // determines the speed of the animation
25           var count = 0; // size of the image during the animation
26
27           // called repeatedly to animate the book cover
28           function run()
29           {
30              count += speed;
31
32              // stop the animation when the image is large enough
33              if ( count >= 375 )
34              {
```

Fig. 12.5 | Dynamic styles used for animation. (Part 1 of 4.)

```
               window.clearInterval( interval );
               interval = null;
            } // end if

            var bigImage = document.getElementById( "imgCover" );
            bigImage.style.width = .7656 * count + "px";
            bigImage.style.height = count + "px";
         } // end function run

         // inserts the proper image into the main image area and
         // begins the animation
         function display( imgfile )
         {
            if ( interval )
               return;

            var bigImage = document.getElementById( "imgCover" );
            var newNode = document.createElement( "img" );
            newNode.id = "imgCover";
            newNode.src = "fullsize/" + imgfile;
            newNode.alt = "Large image";
            newNode.className = "imgCover";
            newNode.style.width = "0px";
            newNode.style.height = "0px";
            bigImage.parentNode.replaceChild( newNode, bigImage );
            count = 0; // start the image at size 0
            interval = window.setInterval( "run()", 10 ); // animate
         } // end function display
         // -->
      </script>
   </head>
   <body>
      <div id = "mainimg" class = "mainimg">
         <img id = "imgCover" src = "fullsize/iw3htp4.jpg"
            alt = "Full cover image" class = "imgCover" />
      </div>
      <div id = "thumbs" class = "thumbs" >
         <img src = "thumbs/iw3htp4.jpg" alt = "iw3htp4"
            onclick = "display( 'iw3htp4.jpg' )" />
         <img src = "thumbs/chtp5.jpg" alt = "chtp5"
            onclick = "display( 'chtp5.jpg' )" />
         <img src = "thumbs/cpphtp6.jpg" alt = "cpphtp6"
            onclick = "display( 'cpphtp6.jpg' )" />
         <img src = "thumbs/jhtp7.jpg" alt = "jhtp7"
            onclick = "display( 'jhtp7.jpg' )" />
         <img src = "thumbs/vbhtp3.jpg" alt = "vbhtp3"
            onclick = "display( 'vbhtp3.jpg' )" />
         <img src = "thumbs/vcsharphtp2.jpg" alt = "vcsharphtp2"
            onclick = "display( 'vcsharphtp2.jpg' )" />
      </div>
   </body>
</html>
```

Fig. 12.5 | Dynamic styles used for animation. (Part 2 of 4.)

a) The cover viewer page loads with the cover of this book.


Deitel Book Cover Viewer - Windows Internet Explorer
C:\examples\ch12\coverviewer.html
Google
Deitel Book Cover Viewer
Page
Tools
Done
My Computer
100%

b) When the user clicks the thumbnail of C How to Program, the full-size image begins growing from the top-left corner of the window.

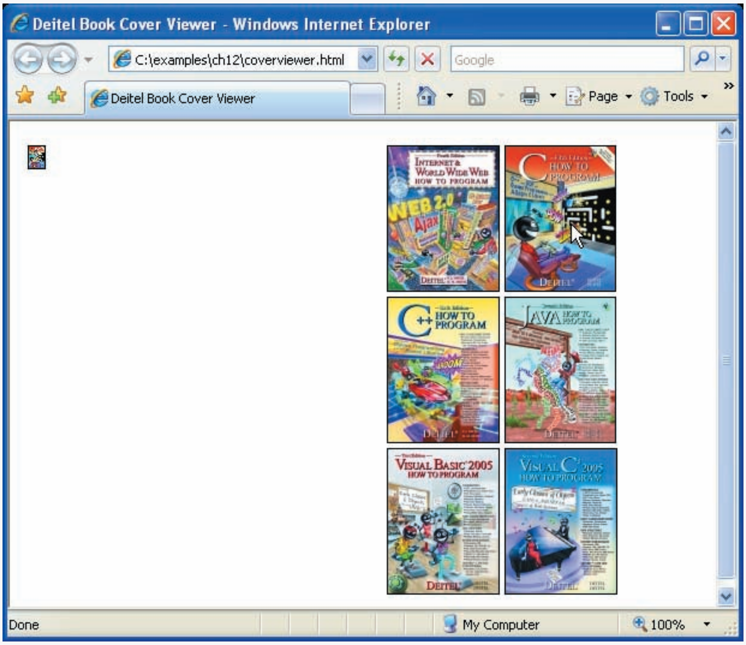
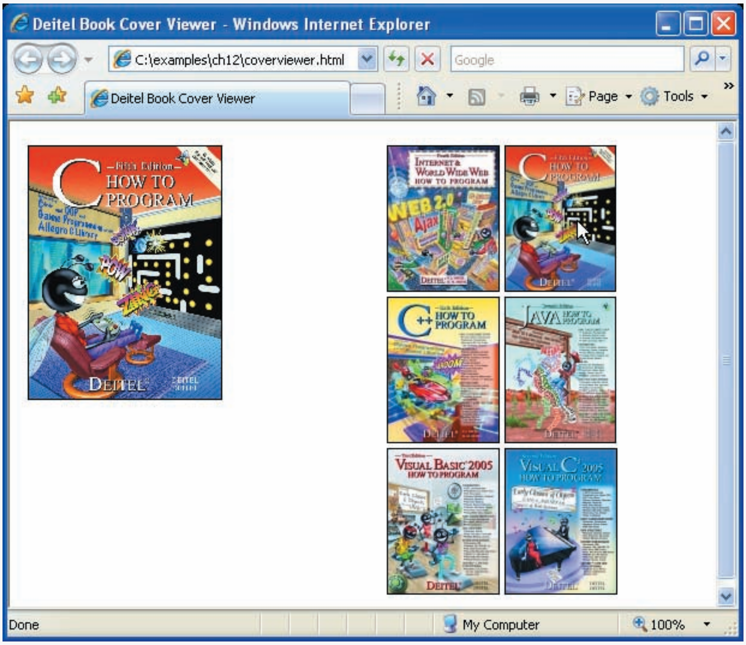

Deitel Book Cover Viewer - Windows Internet Explorer
C:\examples\ch12\coverviewer.html
Google
Deitel Book Cover Viewer
Page
Tools
Done
My Computer
100%

Fig. 12.5 | Dynamic styles used for animation. (Part 3 of 4.)

c) The cover continues to grow.


Deitel Book Cover Viewer - Windows Internet Explorer
C:\examples\ch12\coverviewer.html
Google
Deitel Book Cover Viewer
Page
Tools
Done
My Computer
100%

d) The animation finishes when the cover reaches its full size.

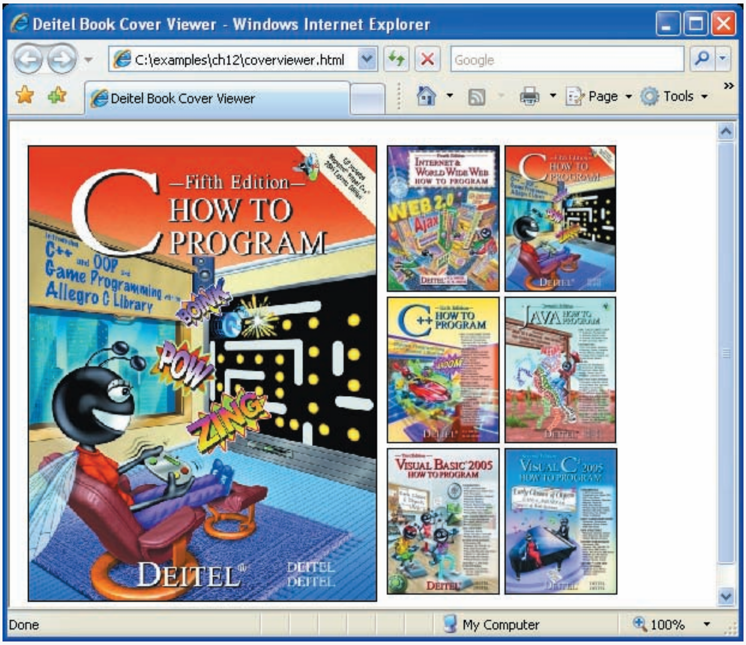

Deitel Book Cover Viewer - Windows Internet Explorer
C:\examples\ch12\coverviewer.html
Google
Deitel Book Cover Viewer
Page
Tools
Done
My Computer
100%

Fig. 12.5 | Dynamic styles used for animation. (Part 4 of 4.)

function run every 10 milliseconds. The setInterval method returns a unique identifier to keep track of that particular interval—we assign this identifier to the variable interval. We use this identifier to stop the animation when the image has finished growing.

The run function, defined in lines 28–42, increases the height of the image by the value of speed and updates its width accordingly to keep the aspect ratio consistent. Because the run function is called every 10 milliseconds, this increase happens repeatedly to create an animated growing effect. Line 30 adds the value of speed (declared and initialized to 6 in line 24) to count, which keeps track of the animation's progress and dictates the current size of the image. If the image has grown to its full height (375), line 35 uses the window's **clearInterval method** to stop the repetitive calls of the run method. We pass to clearInterval the interval identifier (stored in interval) that setInterval created in line 61. Although it seems unnecessary in this script, this identifier allows the script to keep track of multiple intervals running at the same time and to choose which interval to stop when calling clearInterval.

Line 39 gets the image and lines 40–41 set its width and height CSS properties. Note that line 40 multiplies count by a scaling factor of .7656 in order to keep the ratio of the image's dimensions consistent with the actual dimensions of the image. Run the code example and click on a thumbnail image to see the full animation effect.

This section demonstrated the concept of dynamically changing CSS styles using JavaScript and the DOM. We also discussed the basics of how to create scripted animations using setInterval and clearInterval.

12.6 Summary of the DOM Objects and Collections

As you have seen in the preceding sections, the objects and collections in the W3C DOM give you flexibility in manipulating the elements of a web page. We have shown how to access the objects in a page, how to access the objects in a collection, and how to change element styles dynamically.

The W3C DOM allows you to access every element in an XHTML document. Each element in a document is represented by a separate object. The diagram in Fig. 12.6 shows many of the important objects and collections provided by the W3C DOM. Figure 12.7 provides a brief description of each object and collection in Fig. 12.6.

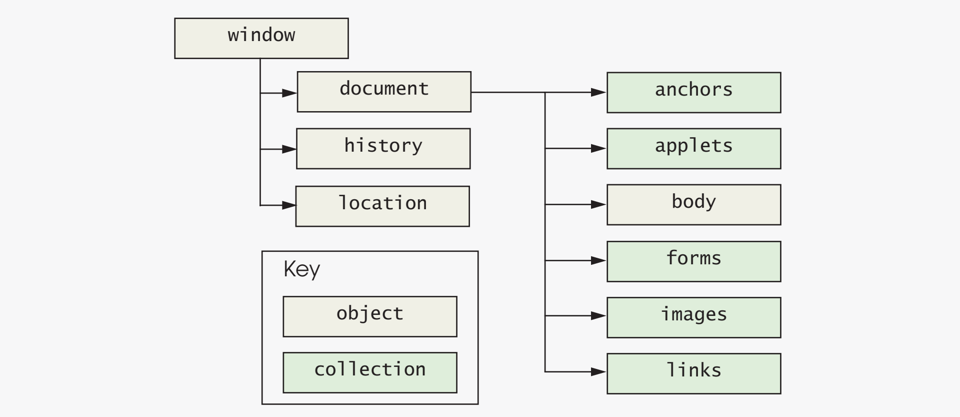


Fig. 12.6 | W3C Document Object Model.

Object or collection	Description
Objects	
window	Represents the browser window and provides access to the document object contained in the window. Also contains history and location objects.
document	Represents the XHTML document rendered in a window. The document object provides access to every element in the XHTML document and allows dynamic modification of the XHTML document. Contains several collections for accessing all elements of a given type.
body	Provides access to the body element of an XHTML document.
history	Keeps track of the sites visited by the browser user. The object provides a script programmer with the ability to move forward and backward through the visited sites.
location	Contains the URL of the rendered document. When this object is set to a new URL, the browser immediately navigates to the new location.
Collections	
anchors	Collection contains all the anchor elements (a) that have a name or id attribute. The elements appear in the collection in the order in which they were defined in the XHTML document.
forms	Contains all the form elements in the XHTML document. The elements appear in the collection in the order in which they were defined in the XHTML document.
images	Contains all the img elements in the XHTML document. The elements appear in the collection in the order in which they were defined in the XHTML document.
links	Contains all the anchor elements (a) with an href property. The elements appear in the collection in the order in which they were defined in the XHTML document.

Fig. 12.7 | Objects and collections in the W3C Document Object Model.

For a complete reference on the W3C Document Object Model, see the DOM Level 3 recommendation from the W3C at http://www.w3.org/TR/DOM-Level-3-Core/. The DOM Level 2 HTML Specification (the most recent HTML DOM standard), available at http://www.w3.org/TR/DOM-Level-2-HTML/, describes additional DOM functionality specific to HTML, such as objects for various types of XHTML elements. Keep in mind that not all web browsers implement all features included in the specification.

12.7 Wrap-Up

This chapter discussed the Document Object Model, which provides access to a web page's elements. We described the child-parent relationships that exist between a document's nodes, and provided visual representations of a DOM tree to illustrate how elements in a web page are related. We introduced several properties and methods that allow us to manipulate nodes, and described groups of related objects on a page, called collections. Finally, we discussed how to dynamically modify content in a web page, adding interactivity and animation to the existing site. In the next chapter, we discuss JavaScript events, which enable JavaScript programs to process user interactions with a web page.

12.8 Web Resources

`www.deitel.com/javascript/`

The Deitel JavaScript Resource Center contains links to some of the best JavaScript resources on the web. There you'll find categorized links to JavaScript tools, code generators, forums, books, libraries, frameworks, tutorials and more. Check out the section specifically dedicated to the Document Object Model. Be sure to visit the related Resource Centers on XHTML (`www.deitel.com/xhtml/`) and CSS 2.1 (`www.deitel.com/css21/`).

Summary

Section 12.1 Introduction

- The Document Object Model gives you access to all the elements on a web page. Using JavaScript, you can create, modify and remove elements in the page dynamically.

Section 12.2 Modeling a Document: DOM Nodes and Trees

- The `getElementById` method returns objects called DOM nodes. Every element in an XHTML page is modeled in the web browser by a DOM node.
- All the nodes in a document make up the page's DOM tree, which describes the relationships among elements.
- Nodes are related to each other through child-parent relationships. An XHTML element inside another element is said to be a child of the containing element. The containing element is known as the parent. A node may have multiple children, but only one parent. Nodes with the same parent node are referred to as siblings.
- Firefox's DOM Inspector and the IE Web Developer Toolbar allow you to see a visual representation of a document's DOM tree and information about each node.
- The document node in a DOM tree is called the root node, because it has no parent.

Section 12.3 Traversing and Modifying a DOM Tree

- The `className` property of a DOM node allows you to change an XHTML element's `class` attribute.
- The `id` property of a DOM node controls an element's `id` attribute.
- The `document` object's `createElement` method creates a new DOM node, taking the tag name as an argument. Note that while `createElement` *creates* an element, it does not *insert* the element on the page.
- The `document`'s `createTextNode` method creates a DOM node that can contain only text. Given a string argument, `createTextNode` inserts the string into the text node.
- Method `appendChild` is called on a parent node to insert a child node (passed as an argument) after any existing children.

- The parentNode property of any DOM node contains the node's parent.
- The insertBefore method is called on a parent with a new child and an existing child as arguments. The new child is inserted as a child of the parent directly before the existing child.
- The replaceChild method is called on a parent, taking a new child and an existing child as arguments. The method inserts the new child into its list of children in place of the existing child.
- The removeChild method is called on a parent with a child to be removed as an argument.

Section 12.4 DOM Collections

- Included in the Document Object Model is the notion of collections, which are groups of related objects on a page. DOM collections are accessed as properties of DOM objects such as the document object or a DOM node.
- The document object has properties containing the images collection, links collection, forms collection and anchors collection. These collections contain all the elements of the corresponding type on the page.
- To find the number of elements in the collection, use the collection's length property.
- To access items in a collection, use square brackets the same way you would with an array, or use the item method. The item method of a DOM collection is used to access specific elements in a collection, taking an index as an argument. The namedItem method takes a name as a parameter and finds the element in the collection, if any, whose id attribute or name attribute matches it.
- The href property of a DOM link node refers to the link's href attribute. Many types of XHTML elements are represented by special types of nodes that extend the functionality of a basic DOM node.
- Collections allow easy access to all elements of a single type in a page. This is useful for gathering elements into one place and for applying changes across an entire page.

Section 12.5 Dynamic Styles

- An element's style can be changed dynamically. Often such a change is made in response to user events, which are discussed in the next chapter. Such style changes can create many effects, including mouse hover effects, interactive menus, and animations.
- The body property of the document object refers to the body element in the XHTML page.
- The style property can access a CSS property in the format *node*.style.*styleproperty*.
- A CSS property with a hyphen (-), such as background-color, is referred to as backgroundColor in JavaScript, to avoid confusion with the subtraction (-) operator. Removing the hyphen and capitalizing the first letter of the following word is the convention for most CSS properties.
- The setInterval method of the window object repeatedly executes a statement on a certain interval. It takes two parameters—a statement to execute repeatedly, and an integer specifying how often to execute it, in milliseconds. The setInterval method returns a unique identifier to keep track of that particular interval.
- The window object's clearInterval method stops the repetitive calls of object's setInterval method. We pass to clearInterval the interval identifier that setInterval returned.

Section 12.6 Summary of the DOM Objects and Collections

- The objects and collections in the W3C DOM give you flexibility in manipulating the elements of a web page.
- The W3C DOM allows you to access every element in an XHTML document. Each element in a document is represented by a separate object.

- For a reference on the W3C Document Object Model, see the DOM Level 3 recommendation from the W3C at `http://www.w3.org/TR/DOM-Level-3-Core/`. The DOM Level 2 HTML Specification, available at `http://www.w3.org/TR/DOM-Level-2-HTML/`, describes additional DOM functionality specific to HTML, such as objects for various types of XHTML elements.
- Not all web browsers implement all features included in the DOM specification.

Terminology

`anchors` collection of the `document` object
`body` property of the `document` object
`appendChild` method of a DOM node
child
`className` property of a DOM node
`clearInterval` method of the `window` object
collection
`createElement` method of the `document` object
`createTextNode` method of the `document` object
`document` object
Document Object Model
DOM collection
DOM Inspector
DOM node
DOM tree
dynamic style
`forms` collection of the `document` object
`href` property of an `a` (anchor) node
Internet Explorer Web Developer Toolbar
`id` property of a DOM node
`images` collection
`innerHTML` property of a DOM node
`insertBefore` method of a DOM node
`item` method of a DOM collection
`length` property of a DOM collection
`links` collection of the `document` object
`namedItem` method of a DOM collection
object hierarchy
parent
`removeChild` method of a DOM node
`replaceChild` method of a DOM node
root node
`setInterval` method of the `window` object
sibling
`style` property of a DOM node
W3C Document Object Model

Self-Review Exercises

12.1 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Every XHTML element in a page is represented by a DOM tree.
b) A text node cannot have child nodes.
c) The `document` node in a DOM tree cannot have child nodes.
d) You can change an element's style class dynamically with the `style` property.
e) The `createElement` method creates a new node and inserts it into the document.
f) The `setInterval` method calls a function repeatedly at a set time interval.
g) The `insertBefore` method is called on the document object, taking a new node and an existing one to insert the new one before.
h) The most recently started interval is stopped when the `clearInterval` method is called.
i) The collection `links` contains all the links in a document with specified `name` or `id` attributes.

12.2 Fill in the blanks for each of the following statements.

a) The ________ property refers to the text inside an element, including XHTML tags.
b) A document's DOM ________ represents all of the nodes in a document, as well as their relationships to each other.
c) The ________ property contains the number of elements in a collection.
d) The ________ method allows access to an individual element in a collection.
e) The ________ collection contains all the `img` elements on a page.
f) The ________ object contains information about the sites that a user previously visited.
g) CSS properties may be accessed using the ________ object.

Answers to Self-Review Exercises

12.1 a) False. Every element is represented by a DOM *node*. Each node is a member of the document's DOM tree. b) True. c) False. The `document` is the root node, therefore has no parent node. d) False. The style class is changed with the `className` property. e) False. The `createElement` method creates a node, but does not insert it into the DOM tree. f) True. g) False. `insertBefore` is called on the parent. h) False. `clearInterval` takes an interval identifier as an argument to determine which interval to end. i) False. The `links` collection contains all links in a document.

12.2 a) `innerHTML`. b) tree. c) `length`. d) `item`. e) `images`. f) `history`. g) `style`.

Exercises

12.3 Modify Fig. 12.3 to use a background color to highlight all the links in the page instead of displaying them in a box at the bottom.

12.4 Use the Firefox DOM Inspector or the IE Web Developer Toolbar to view the DOM tree of the document in Fig. 12.2. Look at the document tree of your favorite website. Notice the information these tools give you in the right panel(s) about an element when you click it.

12.5 Write a script that contains a button and a counter in a `div`. The button should increment the counter each time it is clicked.

12.6 Write a script that prints out the length of all the JavaScript collections on a page.

12.7 Create a web page in which users are allowed to select their favorite layout and formatting through the use of the `className` property.

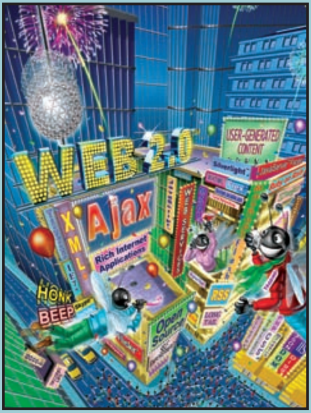
12.8 *(15 Puzzle)* Write a web page that enables the user to play the game of 15. There is a 4-by-4 board (implemented as an XHTML table) for a total of 16 slots. One of the slots is empty. The other slots are occupied by 15 tiles, randomly numbered from 1 through 15. Any tile next to the currently empty slot can be moved into the currently empty slot by clicking on the tile. Your program should create the board with the tiles out of order. The user's goal is to arrange the tiles in sequential order row by row. Using the DOM and the `onclick` event, write a script that allows the user to swap the positions of the open position and an adjacent tile. [*Hint:* The `onclick` event should be specified for each table cell.]

12.9 Modify your solution to Exercise 12.8 to determine when the game is over, then prompt the user to determine whether to play again. If so, scramble the numbers using the `Math.random` method.

12.10 Modify your solution to Exercise 12.9 to use an image that is split into 16 equally sized pieces. Discard one of the pieces and randomly place the other 15 pieces in the XHTML table.

13

JavaScript: Events



The wisest prophets make sure of the event first.

—Horace Walpole

Do you think I can listen all day to such stuff?

—Lewis Carroll

The user should feel in control of the computer; not the other way around. This is achieved in applications that embody three qualities: responsiveness, permissiveness, and consistency.

—*Inside Macintosh, Volume 1*
Apple Computer, Inc., 1985

We are responsible for actions performed in response to circumstances for which we are not responsible.

—Allan Massie

OBJECTIVES

In this chapter you will learn:

- The concepts of events, event handlers and event bubbling.
- To create and register event handlers that respond to mouse and keyboard events.
- To use the `event` object to get information about an event.
- To recognize and respond to many common events.

13.1 Introduction

We've seen that XHTML pages can be controlled via scripting, and we've already used a few events to trigger scripts, such as the `onclick` and `onsubmit` events. This chapter goes into more detail on **JavaScript events**, which allow scripts to respond to user interactions and modify the page accordingly. Events allow scripts to respond to a user who is moving the mouse, entering form data or pressing keys. Events and event handling help make web applications more responsive, dynamic and interactive.

In this chapter, we discuss how to set up functions to react when an event **fires** (occurs). We give examples of event handling for nine common events, including mouse events and form-processing events. A the end of the chapter, we provide a table of the events covered in this chapter and other useful events.

13.2 Registering Event Handlers

Functions that handle events are called **event handlers**. Assigning an event handler to an event on a DOM node is called **registering an event handler**. Previously, we have registered event handlers using the **inline model**, treating events as attributes of XHTML elements (e.g., `<p onclick = "myfunction()">`). Another model, known as the **traditional model**, for registering event handlers is demonstrated alongside the inline model in Fig. 13.1.

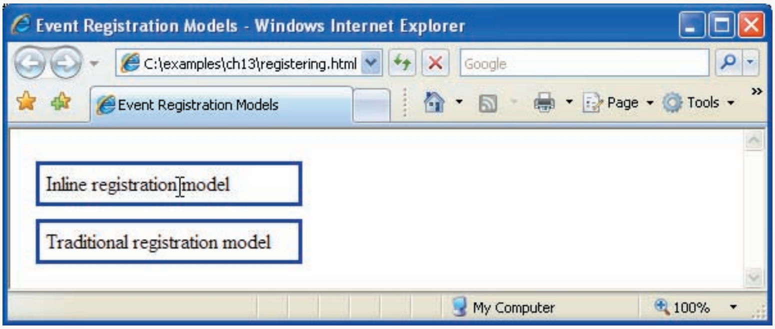
In the earliest event-capable browsers, the inline model was the only way to handle events. Later, Netscape developed the traditional model and Internet Explorer adopted it. Since then, both Netscape and Microsoft have developed separate (incompatible)

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

```

Fig. 13.1 | Event registration models. (Part 1 of 3.)

```
<!-- Fig. 13.1: registering.html -->
<!-- Event registration models. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Event Registration Models</title>
      <style type = "text/css">
         div { padding: 5px;
               margin: 10px;
               border: 3px solid #0000BB;
               width: 12em }
      </style>
      <script type = "text/javascript">
         <!--
         // handle the onclick event regardless of how it was registered
         function handleEvent()
         {
            alert( "The event was successfully handled." );
         } // end function handleEvent

         // register the handler using the traditional model
         function registerHandler()
         {
            var traditional = document.getElementById( "traditional" );
            traditional.onclick = handleEvent;
         } // end function registerHandler
         // -->
      </script>
   </head>
   <body onload = "registerHandler()">
      <!-- The event handler is registered inline -->
      <div id = "inline" onclick = "handleEvent()">
         Inline registration model</div>

      <!-- The event handler is registered by function registerHandler -->
      <div id = "traditional">Traditional registration model</div>
   </body>
</html>
```

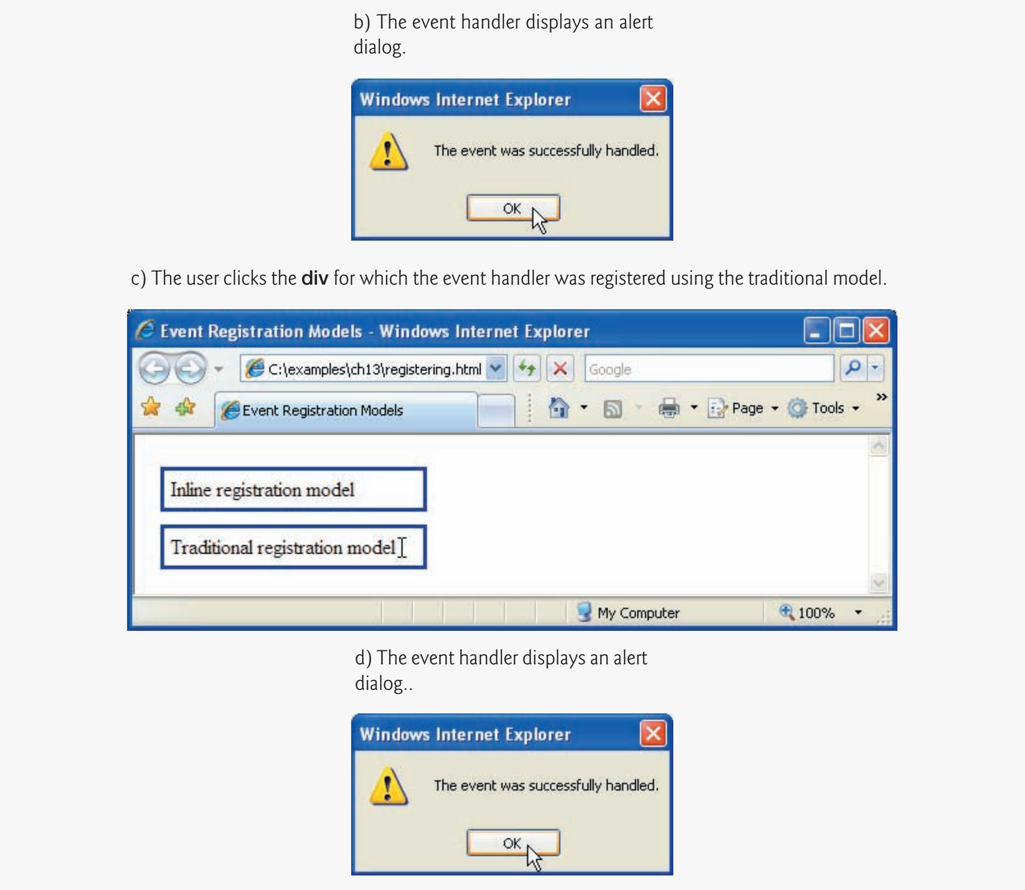
a) The user clicks the **div** for which the event handler was registered using the inline model.


Event Registration Models - Windows Internet Explorer
C:\examples\ch13\registering.html
Google
Event Registration Models
Page
Tools
My Computer
100%

Inline registration model

Traditional registration model

Fig. 13.1 | Event registration models. (Part 2 of 3.)



b) The event handler displays an alert dialog.

Windows Internet Explorer

The event was successfully handled.

c) The user clicks the **div** for which the event handler was registered using the traditional model.

Event Registration Models - Windows Internet Explorer

C:\examples\ch13\registering.html

Inline registration model

Traditional registration model

d) The event handler displays an alert dialog..

Windows Internet Explorer

The event was successfully handled.

Fig. 13.1 | Event registration models. (Part 3 of 3.)

advanced event models with more functionality than either the inline or the traditional model. Netscape's advanced model was adapted by the W3C to create a DOM Events Specification. Most browsers support the W3C model, but Internet Explorer 7 does not. This means that to create cross-browser websites, we are mostly limited to the traditional and inline event models. While the advanced models provide more convenience and functionality, most of the features can be implemented with the traditional model.

Line 35 assigns `"handleEvent()"` to the `onclick` attribute of the `div` in lines 35–36. This is the inline model for event registration we've seen in previous examples. The `div` in line 39 is assigned an event handler using the traditional model. When the `body` element (lines 33–40) loads, the `registerHandler` function is called.

Function `registerHandler` (lines 25–29) uses JavaScript to register the function `handleEvent` as the event handler for the `onclick` event of the `div` with the `id` `"traditional"`. Line 27 gets the `div`, and line 28 assigns the function `handleEvent` to the `div`'s `onclick` property.

Notice that in line 28, we do not put `handleEvent` in quotes or include parentheses at the end of the function name, as we do in the inline model in line 35. In the inline model, the value of the XHTML attribute is a *JavaScript statement* to execute when the event occurs. The value of the onclick property of a DOM node is not an executable statement, but the name of a *function* to be called when the event occurs. Recall that JavaScript functions can be treated as data (i.e., passed into methods, assigned to variables, etc.).

Common Programming Error 13.1

Putting quotes around the function name when registering it using the inline model would assign a string to the onclick property of the node—a string cannot be called.

Common Programming Error 13.2

Putting parentheses after the function name when registering it using the inline model would call the function immediately and assign its return value to the onclick property.

Once the event handler is registered in line 28, the div in line 39 has the same behavior as the div in lines 35–36, because handleEvent (lines 19–22) is set to handle the onclick event for both divs. When either div is clicked, an alert will display "The event was successfully handled."

The traditional model allows us to register event handlers in JavaScript code. This has important implications for what we can do with JavaScript events. For example, traditional event-handler registration allows us to assign event handlers to many elements quickly and easily using repetition statements, instead of adding an inline event handler to each XHTML element. In the remaining examples in this chapter, we use both the inline and traditional registration models depending on which is more convenient.

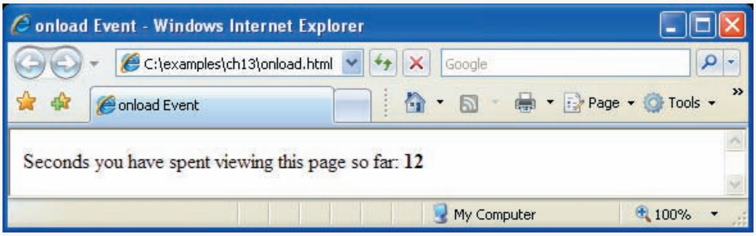
13.3 Event onload

The onload event fires whenever an element finishes loading successfully (i.e., all its children are loaded). Frequently, this event is used in the body element to initiate a script after the page loads in the client's browser. Figure 13.2 uses the onload event for this purpose. The script called by the onload event updates a timer that indicates how many seconds have elapsed since the document was loaded.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 13.2: onload.html -->
<!-- Demonstrating the onload event. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>onload Event</title>
      <script type = "text/javascript">
         <!--
         var seconds = 0;

         // called when the page loads to begin the timer
         function startTimer()
         {
```

Fig. 13.2 | Demonstrating the onload event. (Part 1 of 2.)

```
17                 // 1000 milliseconds = 1 second
18                 window.setInterval( "updateTime()", 1000 );
19              } // end function startTimer
20
21              // called every 1000 ms to update the timer
22              function updateTime()
23              {
24                 ++seconds;
25                 document.getElementById( "soFar" ).innerHTML = seconds;
26              } // end function updateTime
27              // -->
28           </script>
29        </head>
30        <body onload = "startTimer()">
31           <p>Seconds you have spent viewing this page so far:
32           <strong id = "soFar">0</strong></p>
33        </body>
34     </html>
```


onload Event - Windows Internet Explorer
C:\examples\ch13\onload.html
Google
onload Event
Page
Tools
My Computer
100%

Seconds you have spent viewing this page so far: 12

Fig. 13.2 | Demonstrating the `onload` event. (Part 2 of 2.)

Our use of the `onload` event occurs in line 30. After the `body` section loads, the browser triggers the `onload` event. This calls function `startTimer` (lines 15–19), which in turn uses method `window.setInterval` to specify that function `updateTime` (lines 22–26) should be called every `1000` milliseconds. The `updateTime` function increments variable `seconds` and updates the counter on the page.

Note that we could not have created this program without the onload event, because elements in the XHTML page cannot be accessed until the page has loaded. If a script in the `head` attempts to get a DOM node for an XHTML element in the `body`, `getElementById` returns `null` because the `body` has not yet loaded. Other uses of the `onload` event include opening a pop-up window once a page has loaded and triggering a script when an image or Java applet loads.

Common Programming Error 13.3

Trying to get an element in a page before the page has loaded is a common error. Avoid this by putting your script in a function using the `onload` event to call the function.

13.4 Event onmousemove, the event Object and this

This section introduces the `onmousemove` event, which fires repeatedly whenever the user moves the mouse over the web page. We also discuss the `event` object and the keyword this, which permit more advanced event-handling capabilities. Figure 13.3 uses onmousemove and this to create a simple drawing program that allows the user to draw inside a box in red or blue by holding down the *Shift* or *Ctrl* keys.

The XHTML body has a table with a tbody containing one row that gives the user instructions on how to use the program. The body's onload attribute (line 61) calls function createCanvas, which initializes the program by filling in the table.

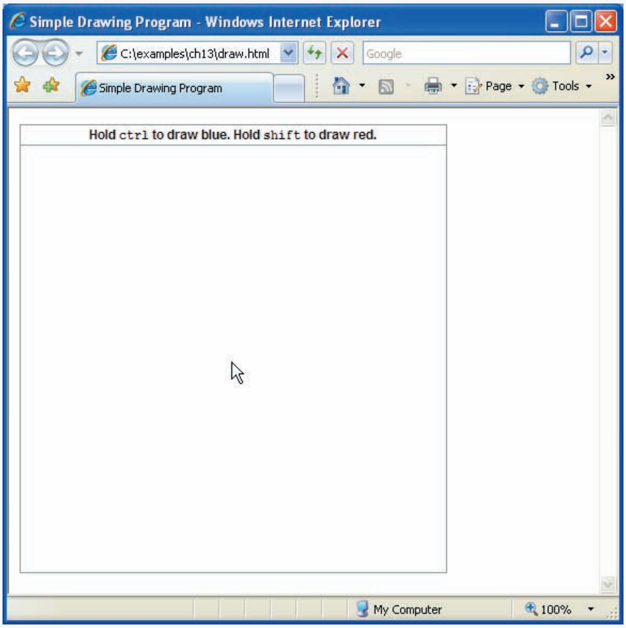
The createCanvas function (lines 23–41) fills in the table with a grid of cells. The CSS rule in lines 14–15 sets the width and height of every td element to 4px. Line 11

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 13.3: draw.html -->
 6  <!-- A simple drawing program. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>Simple Drawing Program</title>
10        <style type = "text/css">
11           #canvas { width: 400px;
12                     border: 1px solid #999999;
13                     border-collapse: collapse }
14           td      { width: 4px;
15                     height: 4px }
16           th.key  { font-family: arial, helvetica, sans-serif;
17                     font-size: 12px;
18                     border-bottom: 1px solid #999999 }
19        </style>
20        <script type = "text/javascript">
21           <!--
22           //initialization function to insert cells into the table
23           function createCanvas ()
24           {
25              var side = 100;
26              var tbody = document.getElementById( "tablebody" );
27
28              for ( var i = 0; i < side; i++ )
29              {
30                 var row = document.createElement( "tr" );
31
32                 for ( var j = 0; j < side; j++ )
33                 {
34                    var cell = document.createElement( "td" );
35                    cell.onmousemove = processMouseMove;
36                    row.appendChild( cell );
37                 } // end for
38
39                 tbody.appendChild( row );
40              } // end for
41           } // end function createCanvas
42
```

Fig. 13.3 | Simple drawing program. (Part 1 of 3.)

```
         // processes the onmousemove event
         function processMouseMove( e )
         {
            // get the event object from IE
            if ( !e )
               var e = window.event;

            // turn the cell blue if the Ctrl key is pressed
            if ( e.ctrlKey )
               this.style.backgroundColor = "blue";

            // turn the cell red if the Shift key is pressed
            if ( e.shiftKey )
               this.style.backgroundColor = "red";
         } // end function processMouseMove
         // -->
      </script>
   </head>
   <body onload = "createCanvas()">
      <table id = "canvas" class = "canvas"><tbody id = "tablebody">
      <tr><th class = "key" colspan = "100">Hold <tt>ctrl</tt>
         to draw blue. Hold <tt>shift</tt> to draw red.</th></tr>
      </tbody></table>
   </body>
</html>
```

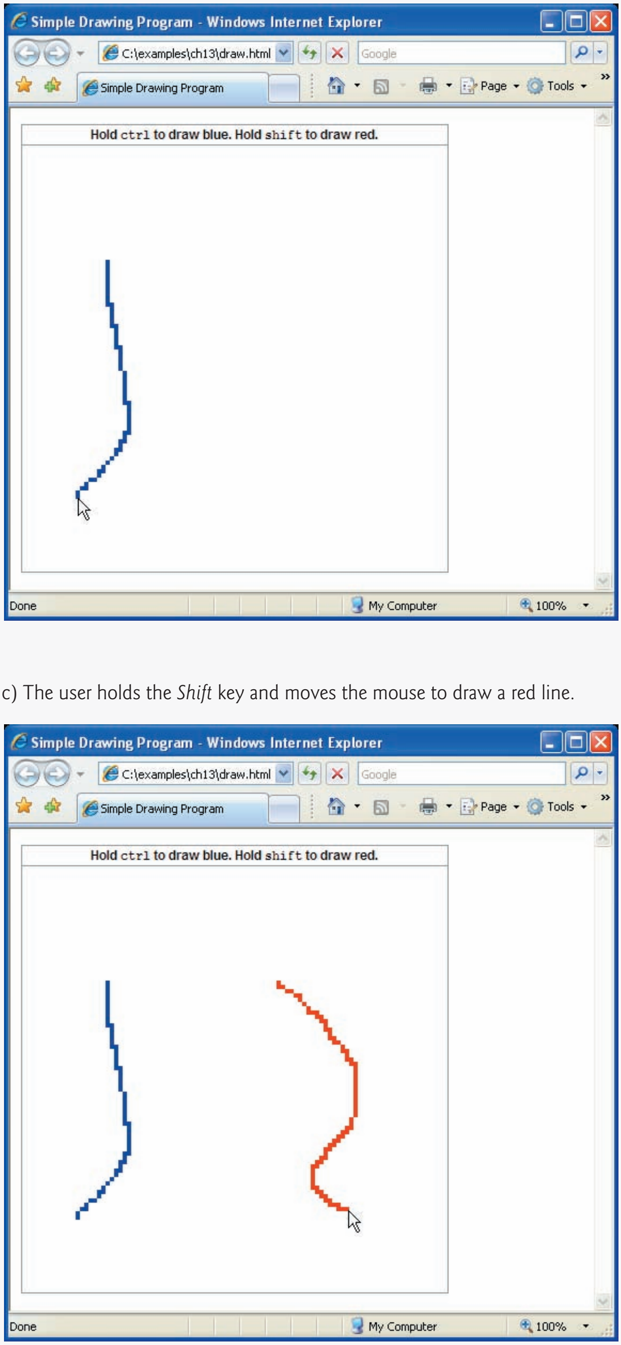
a) The page loads and fills with white cells. With no keys held down, moving the mouse does not draw anything.


Simple Drawing Program - Windows Internet Explorer
C:\examples\ch13\draw.html
Google
Simple Drawing Program
Page
Tools
My Computer
100%

Hold ctrl to draw blue. Hold shift to draw red.

Fig. 13.3 | Simple drawing program. (Part 2 of 3.)

b) The user holds the *Ctrl* key and moves the mouse to draw a blue line.



Hold ctrl to draw blue. Hold shift to draw red.

Simple Drawing Program - Windows Internet Explorer
C:\examples\ch13\draw.html
Google
Simple Drawing Program
Page
Tools
Done
My Computer
100%

c) The user holds the *Shift* key and moves the mouse to draw a red line.

Hold ctrl to draw blue. Hold shift to draw red.

Simple Drawing Program - Windows Internet Explorer
C:\examples\ch13\draw.html
Google
Simple Drawing Program
Page
Tools
Done
My Computer
100%

Fig. 13.3 | Simple drawing program. (Part 3 of 3.)

dictates that the table is `400px` wide. Line 13 uses the `border-collapse` CSS property to eliminate space between the table cells.

Line 25 defines variable `side`, which determines the number of cells in each row and the number of rows created by the nested `for` statements in lines 28–40. We set `side` to `100` in order to fill the table with 10,000 `4px` cells. Line 26 stores the `tbody` element so that we can append rows to it as they are generated.

Common Programming Error 13.4

Although you can omit the `tbody` element in an XHTML table, without it you cannot append `tr` elements as children of a `table` using JavaScript. While Firefox treats appended rows as members of the table body, Internet Explorer will not render any table cells that are dynamically added to a table outside a `thead`, `tbody` or `tfoot` element.

The nested `for` statements in lines 28–40 fill the table with a 100 × 100 grid of cells. The outer loop creates each table row, while the inner loop creates each cell. The inner loop uses the `createElement` method to create a table cell, assigns function `processMouseMove` as the event handler for the cell's `onmousemove` event and appends the cell as a child of the row. The **`onmousemove` event** of an element fires whenever the user moves the mouse over that element.

At this point, the program is initialized and simply calls `processMouseMove` whenever the mouse moves over any table cell. The function `processMouseMove` (lines 44–57) colors the cell the mouse moves over, depending on the key that is pressed when the event occurs. Lines 44–48 get the **`event` object**, which stores information about the event that called the event-handling function.

Internet Explorer and Firefox do not implement the same event models, so we need to account for some differences in how the `event` object can be handled and used. Firefox and other W3C-compliant browsers (e.g., Safari, Opera) pass the `event` object as an argument to the event-handling function. Internet Explorer, on the other hand, stores the event object in the `event` property of the `window` object. To get the event object regardless of the browser, we use a two-step process. Function `processMouseMove` takes the parameter `e` in line 44 to get the `event` object from Firefox. Then, if `e` is undefined (i.e., if the client is Internet Explorer), we assign the object in `window.event` to `e` in line 48.

In addition to providing different ways to access the `event` object, Firefox and Internet Explorer also implement different functionality in the `event` object itself. However, there are several `event` properties that both browsers implement with the same name, and some that both browsers implement with different names. In this book, we use properties that are implemented in both event models, or we write our code to use the correct property depending on the browser—all of our code runs properly in IE7 and Firefox 2.

Once `e` contains the `event` object, we can use it to get information about the event. Lines 51–56 do the actual drawing. The event object's **`ctrlKey` property** contains a boolean which reflects whether the *Ctrl* key was pressed during the event. If `ctrlKey` is true, line 52 executes, changing the color of a table cell.

To determine which table cell to color, we introduce the **`this` keyword**. The meaning of `this` depends on its context. In an event-handling function, `this` refers to the DOM object on which the event occurred. Our function uses `this` to refer to the table cell over which the mouse moved. The `this` keyword allows us to use one event handler to apply a change to one of many DOM elements, depending on which one received the event.

Lines 51–52 change the background color of this table cell to blue if the *Ctrl* key is pressed during the event. Similarly, lines 55–56 color the cell red if the *Shift* key is pressed. To determine this, we use the **shiftKey property** of the event object. This simple function allows the user to draw inside the table on the page in red and blue. You'll add more functionality to this example in the exercises at the end of this chapter.

This example demonstrated the ctrlKey and shiftKey properties of the event object. Figure 13.4 provides a table of some important cross-browser properties of the event object.

This section introduced the event onmousemove and the keyword this. We also discussed more advanced event handling using the event object to get information about the event. The next section continues our introduction of events with the onmouseover and onmouseout events.

Property	Description
altKey	This value is true if the *Alt* key was pressed when the event fired.
cancelBubble	Set to true to prevent the event from bubbling. Defaults to false. (See Section 14.9, Event Bubbling.)
clientX and clientY	The coordinates of the mouse cursor inside the client area (i.e., the active area where the web page is displayed, excluding scrollbars, navigation buttons, etc.).
ctrlKey	This value is true if the *Ctrl* key was pressed when the event fired.
keyCode	The ASCII code of the key pressed in a keyboard event. See Appendix D for more information on the ASCII character set.
screenX and screenY	The coordinates of the mouse cursor on the screen coordinate system.
shiftKey	This value is true if the *Shift* key was pressed when the event fired.
type	The name of the event that fired, without the prefix "on".

Fig. 13.4 | Some event object properties.

13.5 Rollovers with onmouseover and onmouseout

Two more events fired by mouse movements are onmouseover and onmouseout. When the mouse cursor moves into an element, an **onmouseover event** occurs for that element. When the cursor leaves the element, an **onmouseout event** occurs. Figure 13.5 uses these events to achieve a **rollover effect** that updates text when the mouse cursor moves over it. We also introduce a technique for creating rollover images.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

```

Fig. 13.5 | Events onmouseover and onmouseout. (Part 1 of 5.)

```
<!-- Fig. 13.5: onmouseoverout.html -->
<!-- Events onmouseover and onmouseout. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Events onmouseover and onmouseout</title>
      <style type = "text/css">
         body  { background-color: wheat }
         table { border-style: groove;
                 text-align: center;
                 font-family: monospace;
                 font-weight: bold }
         td    { width: 6em }
      </style>
      <script type = "text/javascript">
         <!--
         image1 = new Image();
         image1.src = "heading1.gif";
         image2 = new Image();
         image2.src = "heading2.gif";

         function mouseOver( e )
         {
            if ( !e )
               var e = window.event;

            var target = getTarget( e );

            // swap the image when the mouse moves over it
            if ( target.id == "heading" )
            {
               target.src = image2.src;
               return;
            } // end if

            // if an element's id is defined, assign the id to its color
            // to turn hex code's text the corresponding color
            if ( target.id )
                target.style.color = target.id;
         } // end function mouseOver

         function mouseOut( e )
         {
            if ( !e )
               var e = window.event;

            var target = getTarget( e );

            // put the original image back when the mouse moves away
            if ( target.id == "heading" )
            {
               target.src = image1.src;
               return;
            } // end if
```

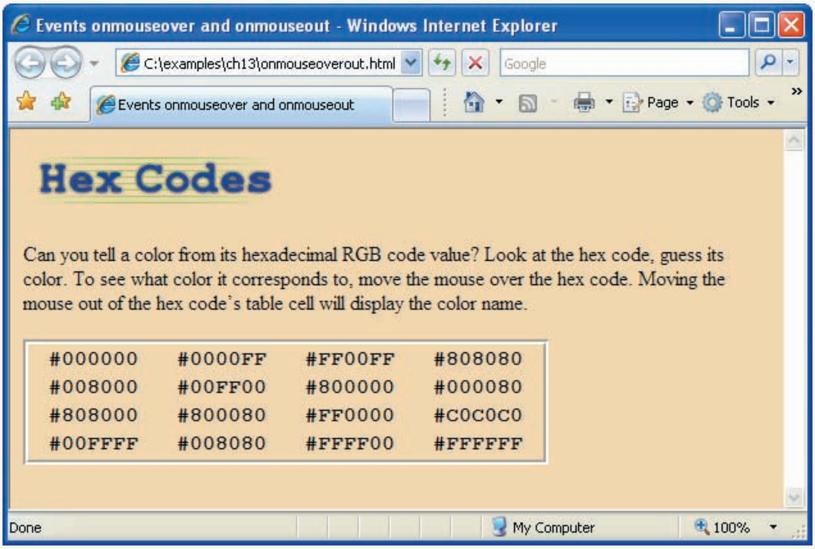
Fig. 13.5 | Events onmouseover and onmouseout. (Part 2 of 5.)

```

            // if an element's id is defined, assign id to innerHTML
            // to display the color name
            if ( target.id )
               target.innerHTML = target.id;
         } // end function mouseOut

         // return either e.srcElement or e.target, whichever exists
         function getTarget( e )
         {
            if ( e.srcElement )
               return e.srcElement;
            else
               return e.target;
         } // end function getTarget

         document.onmouseover = mouseOver;
         document.onmouseout = mouseOut;
         // -->
      </script>
   </head>
   <body>
      <img src = "heading1.gif" id = "heading" alt = "Heading Image" />
      <p>Can you tell a color from its hexadecimal RGB code
      value? Look at the hex code, guess its color. To see
      what color it corresponds to, move the mouse over the
      hex code. Moving the mouse out of the hex code’s table
      cell will display the color name.</p>
      <table>
         <tr>
            <td id = "Black">#000000</td>
            <td id = "Blue">#0000FF</td>
            <td id = "Magenta">#FF00FF</td>
            <td id = "Gray">#808080</td>
         </tr>
         <tr>
            <td id = "Green">#008000</td>
            <td id = "Lime">#00FF00</td>
            <td id = "Maroon">#800000</td>
            <td id = "Navy">#000080</td>
         </tr>
         <tr>
            <td id = "Olive">#808000</td>
            <td id = "Purple">#800080</td>
            <td id = "Red">#FF0000</td>
            <td id = "Silver">#C0C0C0</td>
         </tr>
         <tr>
            <td id = "Cyan">#00FFFF</td>
            <td id = "Teal">#008080</td>
            <td id = "Yellow">#FFFF00</td>
            <td id = "White">#FFFFFF</td>
         </tr>
```

Fig. 13.5 | Events onmouseover and onmouseout. (Part 3 of 5.)

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

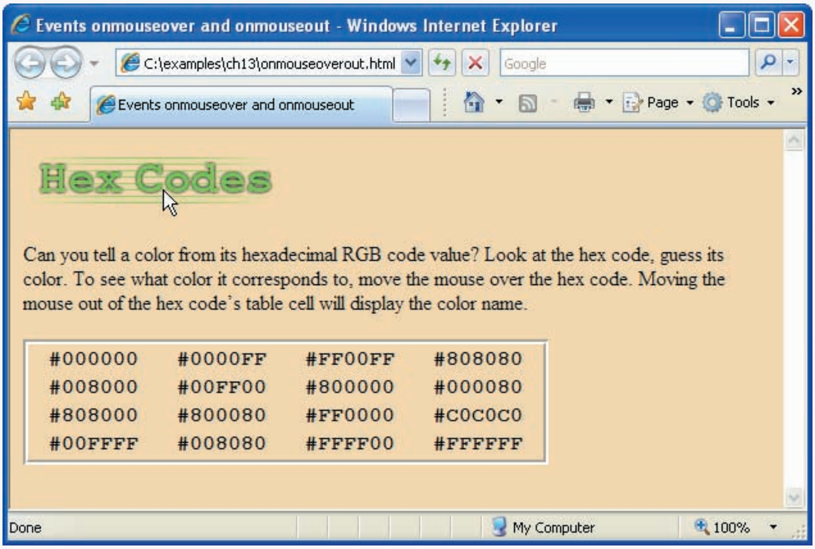
a) The page loads with the blue heading image and all the hex codes in black.


Events onmouseover and onmouseout - Windows Internet Explorer
C:\examples\ch13\onmouseoverout.html
Google
Events onmouseover and onmouseout
Page
Tools
Hex Codes
Done
My Computer
100%

Can you tell a color from its hexadecimal RGB code value? Look at the hex code, guess its color. To see what color it corresponds to, move the mouse over the hex code. Moving the mouse out of the hex code's table cell will display the color name.

#000000	#0000FF	#FF00FF	#808080
#008000	#00FF00	#800000	#000080
#808000	#800080	#FF0000	#C0C0C0
#00FFFF	#008080	#FFFF00	#FFFFFF

b) The heading image switches to an image with green text when the mouse rolls over it.

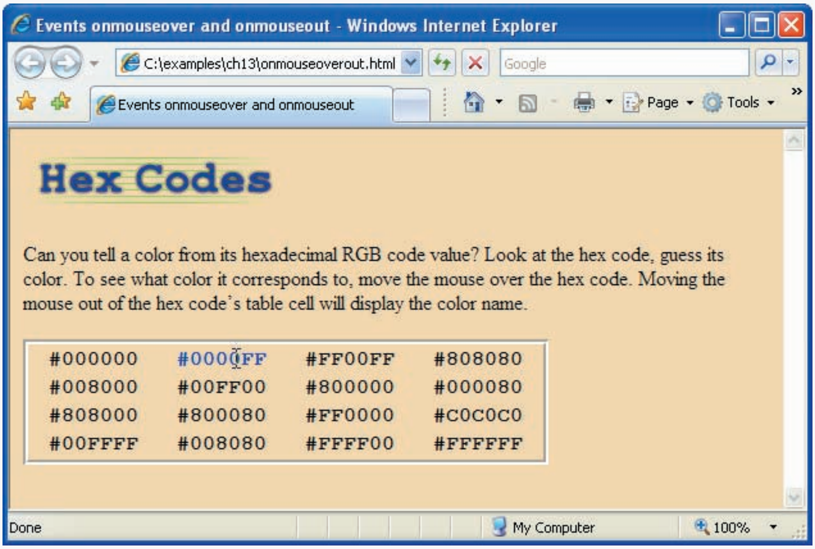

Events onmouseover and onmouseout - Windows Internet Explorer
C:\examples\ch13\onmouseoverout.html
Google
Events onmouseover and onmouseout
Page
Tools
Hex Codes
Done
My Computer
100%

Can you tell a color from its hexadecimal RGB code value? Look at the hex code, guess its color. To see what color it corresponds to, move the mouse over the hex code. Moving the mouse out of the hex code's table cell will display the color name.

#000000	#0000FF	#FF00FF	#808080
#008000	#00FF00	#800000	#000080
#808000	#800080	#FF0000	#C0C0C0
#00FFFF	#008080	#FFFF00	#FFFFFF

Fig. 13.5 | Events `onmouseover` and `onmouseout`. (Part 4 of 5.)

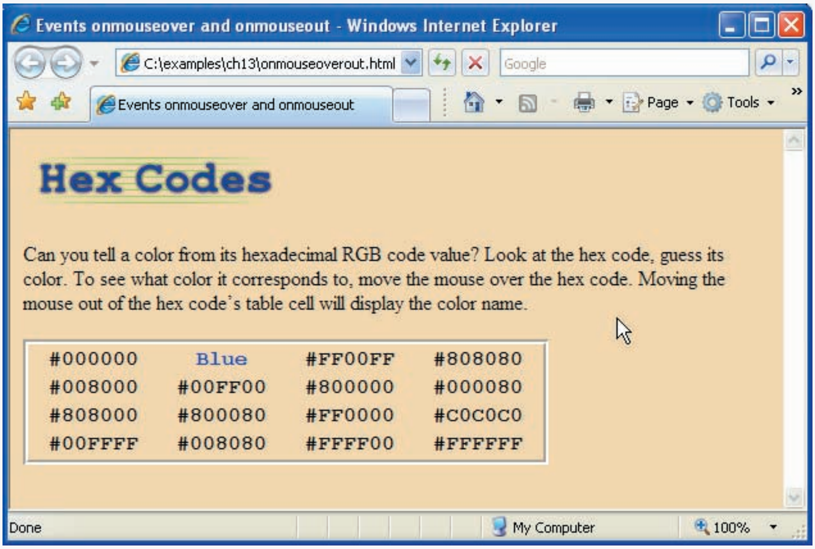
c) When mouse rolls over a hex code, the text color changes to the color represented by the hex code. Notice that the heading image has become blue again because the mouse is no longer over it.


Events onmouseover and onmouseout - Windows Internet Explorer
C:\examples\ch13\onmouseoverout.html
Google
Events onmouseover and onmouseout
Page
Tools
Hex Codes
Done
My Computer
100%

Can you tell a color from its hexadecimal RGB code value? Look at the hex code, guess its color. To see what color it corresponds to, move the mouse over the hex code. Moving the mouse out of the hex code's table cell will display the color name.

#000000	#0000FF	#FF00FF	#808080
#008000	#00FF00	#800000	#000080
#808000	#800080	#FF0000	#C0C0C0
#00FFFF	#008080	#FFFF00	#FFFFFF

d) When the mouse leaves the hex code's table cell, the text changes to the name of the color.


Events onmouseover and onmouseout - Windows Internet Explorer
C:\examples\ch13\onmouseoverout.html
Google
Events onmouseover and onmouseout
Page
Tools
Hex Codes
Done
My Computer
100%

Can you tell a color from its hexadecimal RGB code value? Look at the hex code, guess its color. To see what color it corresponds to, move the mouse over the hex code. Moving the mouse out of the hex code's table cell will display the color name.

#000000	Blue	#FF00FF	#808080
#008000	#00FF00	#800000	#000080
#808000	#800080	#FF0000	#C0C0C0
#00FFFF	#008080	#FFFF00	#FFFFFF

Fig. 13.5 | Events onmouseover and onmouseout. (Part 5 of 5.)

To create a rollover effect for the image in the heading, lines 20–23 create two new JavaScript Image objects—image1 and image2. Image image2 displays when the mouse hovers over the image. Image image1 displays when the mouse is outside the image. The script sets the src properties of each Image in lines 21 and 23, respectively. Creating Image objects preloads the images (i.e., loads the images in advance), so the browser does not need to download the rollover image the first time the script displays the image. If the image is large or the connection is slow, downloading would cause a noticeable delay in the image update.

Performance Tip 13.1

Preloading images used in rollover effects prevents a delay the first time an image is displayed.

Functions `mouseOver` and `mouseOut` are set to process the `onmouseover` and `onmouseout` events, respectively, in lines 74–75. Both functions begin (lines 25–28 and 45–48) by getting the `event` object and using function `getTarget` to find the element that received the action. Because of browser event model differences, we need `getTarget` (defined in lines 66–72) to return the DOM node targeted by the action. In Internet Explorer, this node is stored in the `event` object's **`srcElement` property**. In Firefox, it is stored in the `event` object's **`target` property**. Lines 68–71 return the node using the correct property to hide the browser differences from the rest of our program. We must use function `getTarget` instead of `this` because we do not define an event handler for each specific element in the `document`. In this case, using `this` would return the entire document. In both `mouseOver` and `mouseOut`, we assign the return value of `getTarget` to variable `target` (lines 30 and 50).

Lines 33–37 in the `mouseOver` function handle the `onmouseover` event for the heading image by setting its `src` attribute (`target.src`) to the `src` property of the appropriate `Image` object (`image2.src`). The same task occurs with `image1` in the `mouseOut` function (lines 53–57).

The script handles the `onmouseover` event for the table cells in lines 41–42. This code tests whether an `id` is specified, which is true only for our hex code table cells and the heading image in this example. If the element receiving the action has an `id`, the code changes the color of the element to match the color name stored in the `id`. As you can see in the code for the `table` (lines 86–111), each `td` element containing a color code has an `id` attribute set to one of the 16 basic XHTML colors. Lines 61–62 handle the `onmouseout` event by changing the text in the table cell the mouse cursor just left to match the color that it represents.

13.6 Form Processing with `onfocus` and `onblur`

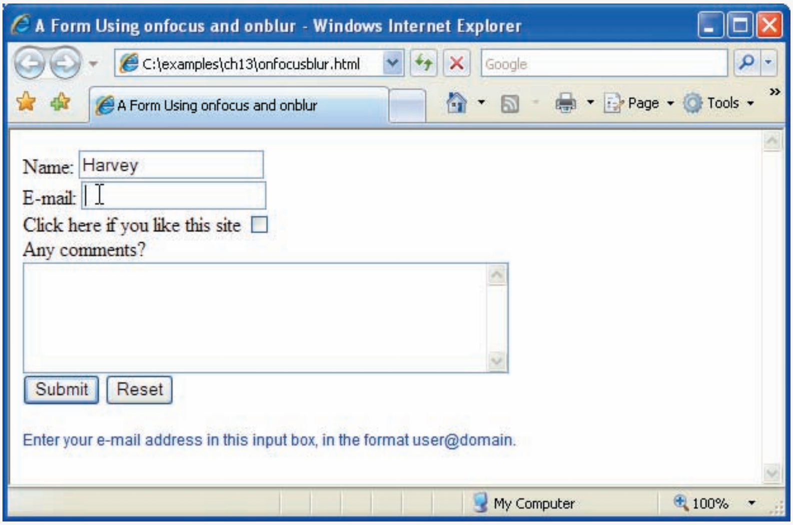
The `onfocus` and `onblur` events are particularly useful when dealing with form elements that allow user input (Fig. 13.6). The `onfocus` event fires when an element gains focus (i.e., when the user clicks a form field or uses the *Tab* key to move between form elements), and `onblur` fires when an element loses focus, which occurs when another control gains the focus. In lines 31–32, the script changes the text inside the `div` below the form (line 58) based on the `messageNum` passed to function `helpText` (lines 29–33). Each of the elements of the form, such as the `name` input in lines 40–41, passes a different value to the `helpText` function when it gains focus (and its `onfocus` event fires). These values are used as indices for `helpArray`, which is declared and initialized in lines 17–27 and stores help messages. When elements lose focus, they all pass the value `6` to `helpText` to clear the `tip` `div` (note that the empty string `""` is stored in the last element of the array).

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 13.6: onfocusblur.html -->
<!-- Demonstrating the onfocus and onblur events. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>A Form Using onfocus and onblur</title>
      <style type = "text/css">
         .tip { font-family: sans-serif;
                color: blue;
                font-size: 12px }
      </style>
      <script type = "text/javascript">
         <!--
         var helpArray =
            [ "Enter your name in this input box.", // element 0
              "Enter your e-mail address in this input box, " +
              "in the format user@domain.", // element 1
              "Check this box if you liked our site.", // element 2
              "In this box, enter any comments you would " +
              "like us to read.", // element 3
              "This button submits the form to the " +
              "server-side script.", // element 4
              "This button clears the form.", // element 5
              "" ]; // element 6

         function helpText( messageNum )
         {
            document.getElementById( "tip" ).innerHTML =
                helpArray[ messageNum ];
         } // end function helpText
         // -->
      </script>
   </head>
   <body>
      <form id = "myForm" action = "">
         <div>
         Name: <input type = "text" name = "name"
            onfocus = "helpText(0)" onblur = "helpText(6)" /><br />
         E-mail: <input type = "text" name = "e-mail"
            onfocus = "helpText(1)" onblur = "helpText(6)" /><br />
         Click here if you like this site
         <input type = "checkbox" name = "like" onfocus =
            "helpText(2)" onblur = "helpText(6)" /><br /><hr />

         Any comments?<br />
         <textarea name = "comments" rows = "5" cols = "45"
            onfocus = "helpText(3)" onblur = "helpText(6)"></textarea>
         <br />
         <input type = "submit" value = "Submit" onfocus =
            "helpText(4)" onblur = "helpText(6)" />
```

Fig. 13.6 | Demonstrating the onfocus and onblur events. (Part 1 of 2.)

```
            <input type = "reset" value = "Reset" onfocus =
               "helpText(5)" onblur = "helpText(6)" />
            </div>
         </form>
         <div id = "tip" class = "tip"></div>
      </body>
   </html>
```

a) The blue message at the bottom of the page instructs the user to enter an e-mail when the e-mail field has focus.



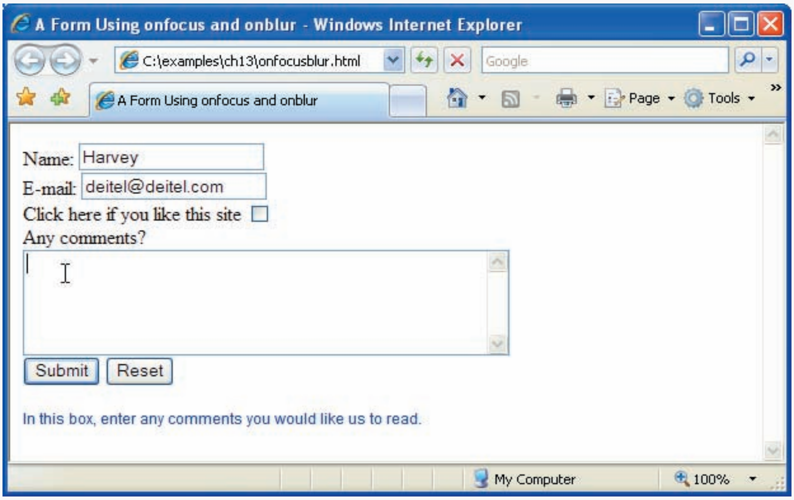
A Form Using onfocus and onblur - Windows Internet Explorer
C:\examples\ch13\onfocusblur.html
Google
A Form Using onfocus and onblur
Page
Tools
My Computer
100%

Name: Harvey
E-mail:
Click here if you like this site □
Any comments?

Submit Reset

Enter your e-mail address in this input box, in the format user@domain.

b) The message changes depending on which field has focus. Now it gives instructions for the comments box.



A Form Using onfocus and onblur - Windows Internet Explorer
C:\examples\ch13\onfocusblur.html
Google
A Form Using onfocus and onblur
Page
Tools
My Computer
100%

Name: Harvey
E-mail: deitel@deitel.com
Click here if you like this site □
Any comments?

Submit Reset

In this box, enter any comments you would like us to read.

Fig. 13.6 | Demonstrating the `onfocus` and `onblur` events. (Part 2 of 2.)

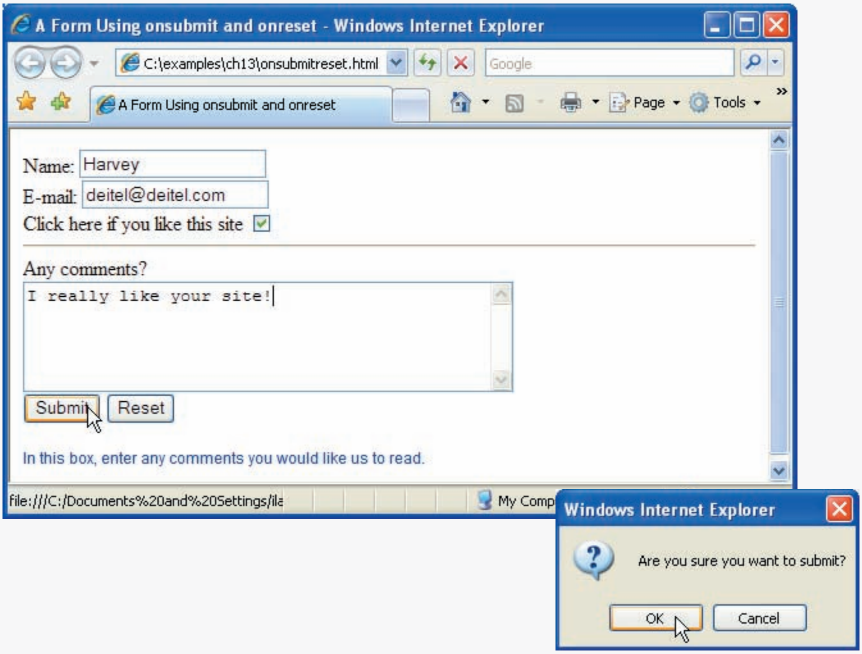
13.7 More Form Processing with onsubmit and onreset

Two more useful events for processing forms are onsubmit and onreset. These events fire when a form is submitted or reset, respectively (Fig. 13.7). Function registerEvents (lines 35–46) registers the event handlers for the form after the body has loaded.

```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 13.7: onsubmitreset.html -->
 6  <!-- Demonstrating the onsubmit and onreset events. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8     <head>
 9        <title>A Form Using onsubmit and onreset</title>
10        <style type = "text/css">
11           .tip { font-family: sans-serif;
12                  color: blue;
13                  font-size: 12px }
14        </style>
15        <script type = "text/javascript">
16           <!--
17           var helpArray =
18              [ "Enter your name in this input box.",
19                "Enter your e-mail address in this input box, " +
20                "in the format user@domain.",
21                "Check this box if you liked our site.",
22                "In this box, enter any comments you would " +
23                "like us to read.",
24                "This button submits the form to the " +
25                "server-side script.",
26                "This button clears the form.",
27                "" ];
28
29           function helpText( messageNum )
30           {
31              document.getElementById( "tip" ).innerHTML =
32                  helpArray[ messageNum ];
33           } // end function helpText
34
35           function registerEvents()
36           {
37              document.getElementById( "myForm" ).onsubmit = function()
38              {
39                 return confirm( "Are you sure you want to submit?" );
40              } // end anonymous function
41
42              document.getElementById( "myForm" ).onreset = function()
43              {
44                 return confirm( "Are you sure you want to reset?" );
45              } // end anonymous function
46           } // end function registerEvents
```

Fig. 13.7 | Demonstrating the onsubmit and onreset events. (Part 1 of 2.)

```
         // -->
      </script>
   </head>
   <body onload = "registerEvents()">
      <form id = "myForm" action = "">
         <div>
         Name: <input type = "text" name = "name"
            onfocus = "helpText(0)" onblur = "helpText(6)" /><br />
         E-mail: <input type = "text" name = "e-mail"
            onfocus = "helpText(1)" onblur = "helpText(6)" /><br />
         Click here if you like this site
         <input type = "checkbox" name = "like" onfocus =
            "helpText(2)" onblur = "helpText(6)" /><br /><hr />

         Any comments?<br />
         <textarea name = "comments" rows = "5" cols = "45"
            onfocus = "helpText(3)" onblur = "helpText(6)"></textarea>
         <br />
         <input type = "submit" value = "Submit" onfocus =
            "helpText(4)" onblur = "helpText(6)" />
         <input type = "reset" value = "Reset" onfocus =
            "helpText(5)" onblur = "helpText(6)" />
         </div>
      </form>
      <div id = "tip" class = "tip"></div>
   </body>
</html>
```


A Form Using onsubmit and onreset - Windows Internet Explorer
C:\examples\ch13\onsubmitreset.html
Google
A Form Using onsubmit and onreset
Page
Tools
file:///C:/Documents%20and%20Settings/ila
My Comp
Windows Internet Explorer
OK
Cancel
Submit
Reset

Name: Harvey
E-mail: deitel@deitel.com
Click here if you like this site ☑

Any comments?
I really like your site!

In this box, enter any comments you would like us to read.

Are you sure you want to submit?

Fig. 13.7 | Demonstrating the `onsubmit` and `onreset` events. (Part 2 of 2.)

Lines 37–40 and 42–45 introduce several new concepts. Line 37 gets the form element ("myForm", lines 51–70), then lines 37–40 assign an anonymous function to its onsubmit property. An anonymous function is defined with no name—it is created in nearly the same way as any other function, but with no identifier after the keyword function. This notation is useful when creating a function for the sole purpose of assigning it to an event handler. We never call the function ourselves, so we don't need to give it a name, and it's more concise to create the function and register it as an event handler at the same time.

The anonymous function (lines 37–40) assigned to the onsubmit property of myForm executes in response to the user submitting the form (i.e., clicking the **Submit** button or pressing the *Enter* key). Line 39 introduces the confirm method of the window object. As with alert, we do not need to prefix the call with the object name window and the dot (.) operator. The confirm dialog asks the users a question, presenting them with an **OK** button and a **Cancel** button. If the user clicks **OK**, confirm returns true; otherwise, confirm returns false.

Our event handlers for the form's onsubmit and onreset events simply return the value of the confirm dialog, which asks the users if they are sure they want to submit or reset (lines 39 and 44, respectively). By returning either true or false, the event handlers dictate whether the default action for the event—in this case submitting or resetting the form—is taken. (Recall that we also returned false from some event-handling functions to prevent forms from submitting in Chapter 12.) Other default actions, such as following a hyperlink, can be prevented by returning false from an onclick event handler on the link. If an event handler returns true or does not return a value, the default action is taken once the event handler finishes executing.

13.8 Event Bubbling

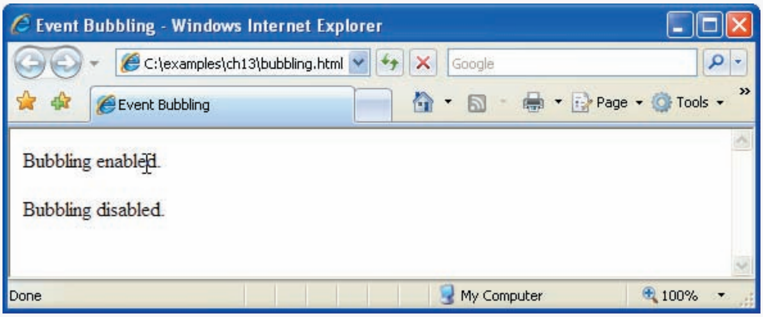
Event bubbling is the process by which events fired in child elements "bubble" up to their parent elements. When an event is fired on an element, it is first delivered to the element's event handler (if any), then to the parent element's event handler (if any). This might result in event handling that was not intended. If you intend to handle an event in a child element alone, you should cancel the bubbling of the event in the child element's event-handling code by using the cancelBubble property of the event object, as shown in Fig. 13.8.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 13.8: bubbling.html -->
<!-- Canceling event bubbling. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Event Bubbling</title>
      <script type = "text/javascript">
         <!--
```

Fig. 13.8 | Canceling event bubbling. (Part 1 of 3.)

```
            function documentClick()
            {
               alert( "You clicked in the document." );
            } // end function documentClick

            function bubble( e )
            {
               if ( !e )
                  var e = window.event;

               alert( "This will bubble." );
               e.cancelBubble = false;
            } // end function bubble

            function noBubble( e )
            {
               if ( !e )
                  var e = window.event;

               alert( "This will not bubble." );
               e.cancelBubble = true;
            } // end function noBubble

            function registerEvents()
            {
               document.onclick = documentClick;
               document.getElementById( "bubble" ).onclick = bubble;
               document.getElementById( "noBubble" ).onclick = noBubble;
            } // end function registerEvents
            // -->
         </script>
      </head>
      <body onload = "registerEvents()">
         <p id = "bubble">Bubbling enabled.</p>
         <p id = "noBubble">Bubbling disabled.</p>
      </body>
   </html>
```

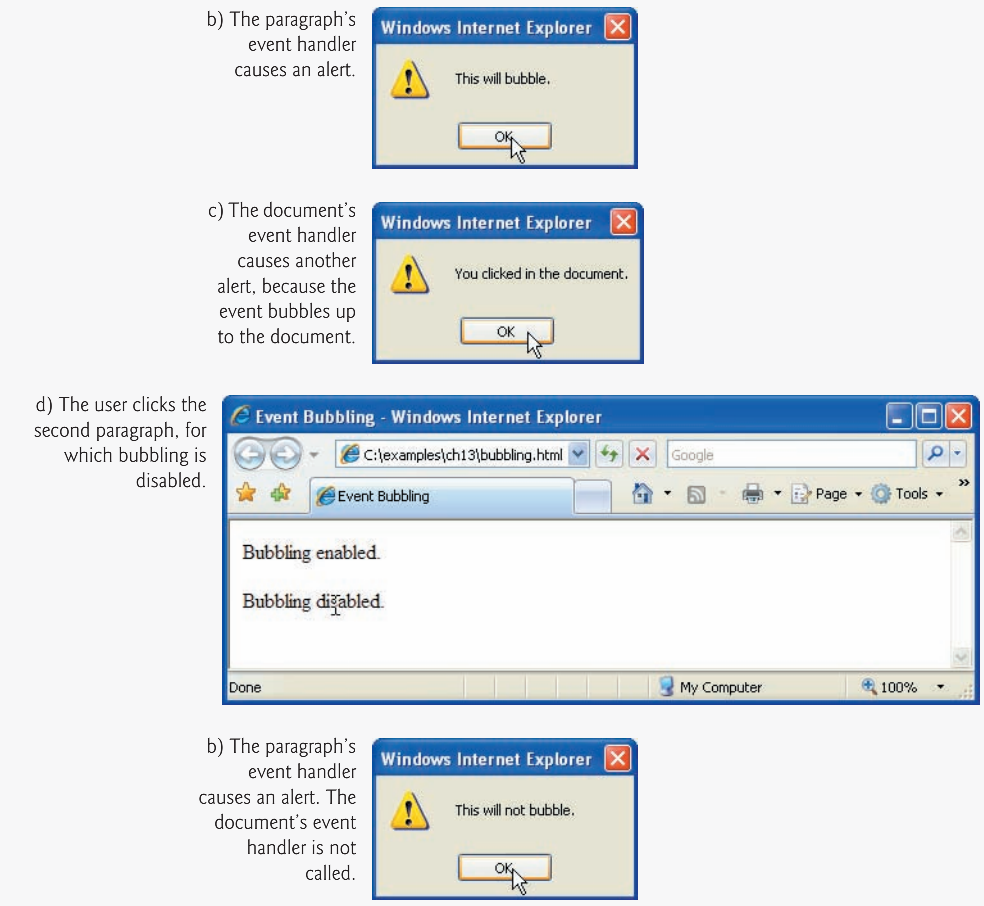
a) The user clicks the first paragraph, for which bubbling is enabled.


Event Bubbling - Windows Internet Explorer
C:\examples\ch13\bubbling.html
Google
Event Bubbling
Page
Tools
Done
My Computer
100%

Bubbling enabled.

Bubbling disabled.

Fig. 13.8 | Canceling event bubbling. (Part 2 of 3.)



b) The paragraph's event handler causes an alert.

Windows Internet Explorer

This will bubble.

OK

c) The document's event handler causes another alert, because the event bubbles up to the document.

Windows Internet Explorer

You clicked in the document.

OK

d) The user clicks the second paragraph, for which bubbling is disabled.

Event Bubbling - Windows Internet Explorer
C:\examples\ch13\bubbling.html
Google
Event Bubbling
Page
Tools
Done
My Computer
100%

Bubbling enabled.

Bubbling disabled.

b) The paragraph's event handler causes an alert. The document's event handler is not called.

Windows Internet Explorer

This will not bubble.

OK

Fig. 13.8 | Canceling event bubbling. (Part 3 of 3.)

Clicking the first p element (line 45) triggers a call to `bubble`. Then, because line 37 registers the `document`'s `onclick` event, `documentClick` is also called. This occurs because the `onclick` event bubbles up to the `document`. This is probably not the desired result. Clicking the second p element (line 46) calls `noBubble`, which disables the event bubbling for this event by setting the `cancelBubble` property of the `event` object to `true`. [*Note:* The default value of `cancelBubble` is `false`, so the statement in line 23 is unnecessary.]

Common Programming Error 13.5

Forgetting to cancel event bubbling when necessary may cause unexpected results in your scripts.

13.9 More Events

The events we covered in this chapter are among the most commonly used. A list of some events supported by both Firefox and Internet Explorer is given with descriptions in Fig. 13.9.

Event	Description
`onabort`	Fires when image transfer has been interrupted by user.
`onchange`	Fires when a new choice is made in a `select` element, or when a text input is changed and the element loses focus.
`onclick`	Fires when the user clicks using the mouse.
`ondblclick`	Fires when the mouse is double clicked.
`onfocus`	Fires when a form element gains focus.
`onkeydown`	Fires when the user pushes down a key.
`onkeypress`	Fires when the user presses then releases a key.
`onkeyup`	Fires when the user releases a key.
`onload`	Fires when an element and all its children have loaded.
`onmousedown`	Fires when a mouse button is pressed down.
`onmousemove`	Fires when the mouse moves.
`onmouseout`	Fires when the mouse leaves an element.
`onmouseover`	Fires when the mouse enters an element.
`onmouseup`	Fires when a mouse button is released.
`onreset`	Fires when a form resets (i.e., the user clicks a reset button).
`onresize`	Fires when the size of an object changes (i.e., the user resizes a window or frame).
`onselect`	Fires when a text selection begins (applies to `input` or `textarea`).
`onsubmit`	Fires when a form is submitted.
`onunload`	Fires when a page is about to unload.

Fig. 13.9 | Cross-browser events.

13.10 Wrap-Up

This chapter introduced JavaScript events, which allow scripts to respond to user interactions and make web pages more dynamic. We described event handlers and how to register them to specific events on DOM nodes in the script. We introduced the event object and the keyword `this`, which allows us to use one event handler to apply a change to the recipient of an event. We discussed various events that fire from mouse actions, as well as the `onfocus`, `onblur`, `onsubmit` and `onreset` events. Finally, we learned about event bubbling, which can produce unexpected effects in scripts that use events. The next chapter introduces XML, an open technology used for data exchange.

13.11 Web Resources

`http://www.quirksmode.org/js/introevents.html`

An introduction and reference site for JavaScript events. Includes comprehensive information on history of events, the different event models, and making events work across multiple browsers.

`wsabstract.com/dhtmltutors/domevent1.shtml`

This *JavaScript Kit* tutorial introduces event handling and discusses the W3C DOM advanced event model.

`http://www.w3schools.com/jsref/jsref_events.asp`

The W3 School's JavaScript Event Reference site has a comprehensive list of JavaScript events, a description of their usage and their browser compatibilities.

`http://www.brainjar.com/dhtml/events/`

BrainJar.com's DOM Event Model site provides a comprehensive introduction to the DOM event model, and has example code to demonstrate several different ways of assigning and using events.

Summary

Section 13.1 Introduction

- JavaScript events allow scripts to respond to user interactions and modify the page accordingly.
- Events and event handling help make web applications more responsive, dynamic and interactive.

Section 13.2 Registering Event Handlers

- Functions that handle events are called event handlers. Assigning an event handler to an event on a DOM node is called registering an event handler.
- We discuss two models for registering event handlers. The inline model treats events as attributes of XHTML elements.
- To register events using the traditional model, we assign the name of the function to the event property of a DOM node.
- In the inline model, the value of the XHTML attribute is a *JavaScript statement* to be executed when the event occurs.
- In the traditional model, the value of the event property of a DOM node is the name of a *function* to be called when the event occurs.
- Traditional registration of event handlers allows us to quickly and easily assign event handlers to many elements using repetition statements, instead of adding an inline event handler to each XHTML element.

Section 13.3 Event `onload`

- The `onload` event fires whenever an element finishes loading successfully.
- If a script in the `head` attempts to get a DOM node for an XHTML element in the `body`, `getElementById` returns `null` because the `body` has not yet loaded.

Section 13.4 Event `onmousemove`, the `event` Object and `this`

- The `onmousemove` event fires whenever the user moves the mouse.
- The `event` object stores information about the event that called the event-handling function.
- The `event` object's `ctrlKey` property contains a boolean which reflects whether the *Ctrl* key was pressed during the event.

- The `event` object's `shiftKey` property reflects whether the *Shift* key was pressed during the event.
- In an event-handling function, `this` refers to the DOM object on which the event occurred.
- The `this` keyword allows us to use one event handler to apply a change to one of many DOM elements, depending on which one received the event.

Section 13.5 Rollovers with `onmouseover` *and* `onmouseout`

- When the mouse cursor enters an element, an `onmouseover` event occurs for that element. When the mouse cursor leaves the element, an `onmouseout` event occurs for that element.
- Creating an `Image` object and setting its `src` property preloads the image.
- The `event` object stores the node on which the action occurred. In Internet Explorer, this node is stored in the `event` object's `srcElement` property. In Firefox, it is stored in the `event` object's `target` property.

Section 13.6 Form Processing with `onfocus` *and* `onblur`

- The `onfocus` event fires when an element gains focus (i.e., when the user clicks a form field or uses the *Tab* key to move between form elements).
- `onblur` fires when an element loses focus, which occurs when another control gains the focus.

Section 13.7 More Form Processing with `onsubmit` *and* `onreset`

- The `onsubmit` and `onreset` events fire when a form is submitted or reset, respectively.
- An anonymous function is a function that is defined with no name—it is created in nearly the same way as any other function, but with no identifier after the keyword `function`.
- Anonymous functions are useful when creating a function for the sole purpose of assigning it to an event handler.
- The `confirm` method asks the users a question, presenting them with an **OK** button and a **Cancel** button. If the user clicks **OK**, `confirm` returns `true`; otherwise, `confirm` returns `false`.
- By returning either `true` or `false`, event handlers dictate whether the default action for the event is taken.
- If an event handler returns `true` or does not return a value, the default action is taken once the event handler finishes executing.

Section 13.8 Event Bubbling

- Event bubbling is the process whereby events fired in child elements "bubble" up to their parent elements. When an event is fired on an element, it is first delivered to the element's event handler (if any), then to the parent element's event handler (if any).
- If you intend to handle an event in a child element alone, you should cancel the bubbling of the event in the child element's event-handling code by using the `cancelBubble` property of the `event` object.

Terminology

`altKey` property of `event` object
anonymous function
`cancelBubble` property of `event` object
`clientX` property of `event` object
`clientY` property of `event` object
`confirm` method of `window` object
`ctrlKey` property of `event` object
default action for an event
event bubbling
event handler
event models
`event` object
event registration models
events in JavaScript

- fire an event
- inline model of event registration
- keyboard event
- `keyCode` property of an `event` object
- mouse event
- `onabort` event
- `onblur` event
- `onchange` event
- `onclick` event
- `ondblclick` event
- `onfocus` event
- `onkeydown` event
- `onkeypress` event
- `onkeyup` event
- `onload` event
- `onmousedown` event
- `onmousemove` event
- `onmouseout` event
- `onmouseover` event
- `onmouseup` event
- `onreset` event
- `onresize` event
- `onselect` event
- `onsubmit` event
- `onunload` event
- registering an event handler
- return value of an event handler
- rollover effect
- `screenX` property of `event` object
- `screenY` property of `event` object
- `setInterval` method of `window` object
- `shiftkey` property of `event` object
- `srcElement` property of `event` object
- *Tab* key to switch between fields on a form
- `target` property of `event` object
- `this` keyword
- traditional model of event registration
- trigger an event
- `type` property of `event` object

Self-Review Exercises

13.1 Fill in the blanks in each of the following statements:

a) Event handlers can be registered in XHTML using the ________ model or in JavaScript using the ________ model.

b) The state of three keys can be retrieved by using the `event` object. These keys are ________, ________ and ________.

c) If a child element does not handle an event, ________ lets the event rise through the object hierarchy.

d) The ________ of an event-handling function specifies whether to perform the default action for the event.

e) In an event handler, the reference for the `id` of an element that fired an event is ________ in Firefox and ________ in Internet Explorer.

f) Three events that fire when the user clicks the mouse are ________, ________ and ________.

13.2 State whether each of the following is *true* or *false*. If the statement is *false*, explain why.

a) The `onload` event fires whenever an element starts loading.

b) The `onclick` event fires when the user clicks the mouse on an element.

c) The `onfocus` event fires when an element loses focus.

d) When using the rollover effect with images, it is a good practice to create `Image` objects that preload the desired images.

e) Returning `true` in an event handler on an a (anchor) element prevents the browser from following the link when the event handler finishes.

Answers to Self-Review Exercises

13.1 a) inline, traditional. b) *Ctrl*, *Alt* and *Shift*. c) event bubbling. d) return value. e) `event.target.id`, `event.srcElement.id`. f) `onclick`, `onmousedown`, `onmouseup`.

13.2 a) False. The `onload` event fires when an element *finishes* loading. b) True. c) False. It fires when an element gains focus. d) True. e) False. Returning `false` prevents the default action.

Exercises

13.3 Add an erase feature to the drawing program in Fig. 13.3. Try setting the background color of the table cell over which the mouse moved to `white` when the *Alt* key is pressed.

13.4 Add a button to your program from Exercise 13.3 to erase the entire drawing window.

13.5 You have a server-side script that cannot handle any ampersands (&) in the form data. Write a function that converts all ampersands in a form field to `" and "` when the field loses focus (`onblur`).

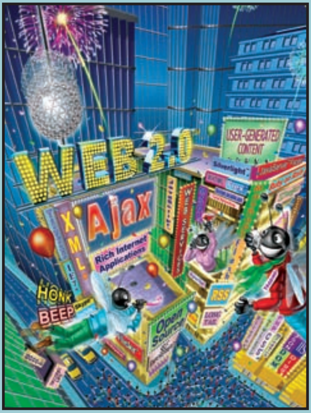
13.6 Write a function that responds to a click anywhere on the page by displaying an `alert` dialog. Display the event name if the user held *Shift* during the mouse click. Display the element name that triggered the event if the user held *Ctrl* during the mouse click.

13.7 Use CSS absolute positioning, `onmousedown`, `onmousemove`, `onmouseup` and the `clientX`/`clientY` properties of the `event` object to create a program that allows you to drag and drop an image. When the user clicks the image, it should follow the cursor until the mouse button is released.

13.8 Modify Exercise 13.7 to allow multiple images to be dragged and dropped in the same page.

14

XML and RSS



Knowing trees, I understand the meaning of patience. Knowing grass, I can appreciate persistence.
—Hal Borland

Like everything metaphysical, the harmony between thought and reality is to be found in the grammar of the language.
—Ludwig Wittgenstein

I played with an idea, and grew willful; tossed it into the air; transformed it; let it escape and recaptured it; made it iridescent with fancy, and winged it with paradox.
—Oscar Wilde

OBJECTIVES

In this chapter you will learn:

- To mark up data using XML.
- How XML namespaces help provide unique XML element and attribute names.
- To create DTDs and schemas for specifying and validating the structure of an XML document.
- To create and use simple XSL style sheets to render XML document data.
- To retrieve and manipulate XML data programmatically using JavaScript.
- RSS and how to programmatically apply an XSL transformation to an RSS document using JavaScript.

Outline

14.1 Introduction

The **Extensible Markup Language (XML)** was developed in 1996 by the **World Wide Web Consortium's (W3C's)** XML Working Group. XML is a widely supported **open technology** (i.e., nonproprietary technology) for describing data that has become the standard format for data exchanged between applications over the Internet.

Web applications use XML extensively and web browsers provide many XML-related capabilities. Sections 14.2–14.7 introduce XML and XML-related technologies—XML namespaces for providing unique XML element and attribute names, and Document Type Definitions (DTDs) and XML Schemas for validating XML documents. These sections support the use of XML in many subsequent chapters. Sections 14.8–14.9 present additional XML technologies and key JavaScript capabilities for loading and manipulating XML documents programmatically—this material is optional but is recommended if you plan to use XML in your own applications. Finally, Section 14.10 introduces RSS—an XML format used to syndicate simple website content—and shows how to format RSS elements using JavaScript and other technologies presented in this chapter.

14.2 XML Basics

XML permits document authors to create **markup** (i.e., a text-based notation for describing data) for virtually any type of information. This enables document authors to create entirely new markup languages for describing any type of data, such as mathematical formulas, software-configuration instructions, chemical molecular structures, music, news, recipes and financial reports. XML describes data in a way that both human beings and computers can understand.

Figure 14.1 is a simple XML document that describes information for a baseball player. We focus on lines 5–9 to introduce basic XML syntax. You will learn about the other elements of this document in Section 14.3.

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 14.1: player.xml -->
4  <!-- Baseball player structured with XML -->
5  <player>
6     <firstName>John</firstName>
7     <lastName>Doe</lastName>
8     <battingAverage>0.375</battingAverage>
9  </player>
```

Fig. 14.1 | XML that describes a baseball player's information.

XML documents contain text that represents content (i.e., data), such as `John` (line 6 of Fig. 14.1), and **elements** that specify the document's structure, such as `firstName` (line 6 of Fig. 14.1). XML documents delimit elements with **start tags** and **end tags**. A start tag consists of the element name in **angle brackets** (e.g., `<player>` and `<firstName>` in lines 5 and 6, respectively). An end tag consists of the element name preceded by a **forward slash (/)** in angle brackets (e.g., `</firstName>` and `</player>` in lines 6 and 9, respectively). An element's start and end tags enclose text that represents a piece of data (e.g., the `player`'s `firstName`—`John`—in line 6, which is enclosed by the `<firstName>` start tag and `</firstName>` end tag). Every XML document must have exactly one **root element** that contains all the other elements. In Fig. 14.1, the root element is `player` (lines 5–9).

XML-based markup languages—called XML **vocabularies**—provide a means for describing particular types of data in standardized, structured ways. Some XML vocabularies include XHTML (Extensible HyperText Markup Language), MathML (for mathematics), VoiceXML™ (for speech), CML (Chemical Markup Language—for chemistry), XBRL (Extensible Business Reporting Language—for financial data exchange) and others that we discuss in Section 14.7.

Massive amounts of data are currently stored on the Internet in many formats (e.g., databases, web pages, text files). Much of this data, especially that which is passed between systems, will soon take the form of XML. Organizations see XML as the future of data encoding. Information technology groups are planning ways to integrate XML into their systems. Industry groups are developing custom XML vocabularies for most major industries that will allow business applications to communicate in common languages. For example, many web services allow web-based applications to exchange data seamlessly through standard protocols based on XML. We discuss web services in Chapter 28.

The next generation of the web is being built on an XML foundation, enabling you to develop more sophisticated web-based applications. XML allows you to assign meaning to what would otherwise be random pieces of data. As a result, programs can "understand" the data they manipulate. For example, a web browser might view a street address in a simple web page as a string of characters without any real meaning. In an XML document, however, this data can be clearly identified (i.e., marked up) as an address. A program that uses the document can recognize this data as an address and provide links to a map of that location, driving directions from that location or other location-specific information. Likewise, an application can recognize names of people, dates, ISBN numbers and any other type of XML-encoded data. The application can then present users with other related information, providing a richer, more meaningful user experience.

Viewing and Modifying XML Documents

XML documents are highly portable. Viewing or modifying an XML document—which is a text file that usually ends with the `.xml` filename extension—does not require special software, although many software tools exist, and new ones are frequently released that make it more convenient to develop XML-based applications. Any text editor that supports ASCII/Unicode characters can open XML documents for viewing and editing. Also, most web browsers can display XML documents in a formatted manner that shows the XML's structure. Section 14.3 demonstrates this in Internet Explorer and Firefox. An important characteristic of XML is that it is both human and machine readable.

Processing XML Documents

Processing an XML document requires software called an **XML parser** (or **XML processor**). A parser makes the document's data available to applications. While reading an XML document's contents, a parser checks that the document follows the syntax rules specified by the W3C's XML Recommendation (`www.w3.org/XML`). XML syntax requires a single root element, a start tag and end tag for each element, and properly nested tags (i.e., the end tag for a nested element must appear before the end tag of the enclosing element). Furthermore, XML is case sensitive, so the proper capitalization must be used in elements. A document that conforms to this syntax is a **well-formed XML document** and is syntactically correct. We present fundamental XML syntax in Section 14.3. If an XML parser can process an XML document successfully, that XML document is well-formed. Parsers can provide access to XML-encoded data in well-formed documents only.

Often, XML parsers are built into software or available for download over the Internet. Some popular parsers include **Microsoft XML Core Services (MSXML)**—which is included with Internet Explorer, the Apache Software Foundation's **Xerces** (`xml.apache.org`) and the open-source **Expat XML Parser** (`expat.sourceforge.net`).

Validating XML Documents

An XML document can reference a **Document Type Definition (DTD)** or a **schema** that defines the proper structure of the XML document. When an XML document references a DTD or a schema, some parsers (called **validating parsers**) can read the DTD/schema and check that the XML document follows the structure defined by the DTD/schema. If the XML document conforms to the DTD/schema (i.e., the document has the appropriate structure), the XML document is **valid**. For example, if in Fig. 14.1 we were referencing a DTD that specified that a `player` element must have `firstName`, `lastName` and `battingAverage` elements, then omitting the `lastName` element (line 7 in Fig. 14.1) would invalidate the XML document `player.xml`. However, the XML document would still be well-formed, because it follows proper XML syntax (i.e., it has one root element, each element has a start tag and an end tag, and the elements are nested properly). By definition, a valid XML document is well-formed. Parsers that cannot check for document conformity against DTDs/schemas are **nonvalidating parsers**—they determine only whether an XML document is well-formed, not whether it is valid.

We discuss validation, DTDs and schemas, as well as the key differences between these two types of structural specifications, in Sections 14.5–14.6. For now, note that schemas are XML documents themselves, whereas DTDs are not. As you will learn in Section 14.6, this difference presents several advantages in using schemas over DTDs.

Software Engineering Observation 14.1

DTDs and schemas are essential for business-to-business (B2B) transactions and mission-critical systems. Validating XML documents ensures that disparate systems can manipulate data structured in standardized ways and prevents errors caused by missing or malformed data.

Formatting and Manipulating XML Documents

Most XML documents contain only data, not formatting instructions, so applications that process XML documents must decide how to manipulate or display the data. For example, a PDA (personal digital assistant) may render an XML document differently than a wireless phone or a desktop computer. You can use Extensible Stylesheet Language (XSL) to specify rendering instructions for different platforms. We discuss XSL in Section 14.8.

XML-processing programs can also search, sort and manipulate XML data using XSL. Some other XML-related technologies are XPath (XML Path Language—a language for accessing parts of an XML document), XSL-FO (XSL Formatting Objects—an XML vocabulary used to describe document formatting) and XSLT (XSL Transformations—a language for transforming XML documents into other documents). We present XSLT and XPath in Section 14.8.

14.3 Structuring Data

In this section and throughout this chapter, we create our own XML markup. XML allows you to describe data precisely in a well-structured format.

XML Markup for an Article

In Fig. 14.2, we present an XML document that marks up a simple article using XML. The line numbers shown are for reference only and are not part of the XML document.

This document begins with an XML declaration (line 1), which identifies the document as an XML document. The **`version` attribute** specifies the XML version to which the document conforms. The current XML standard is version 1.0. Though the W3C released a version 1.1 specification in February 2004, this newer version is not yet widely supported. The W3C may continue to release new versions as XML evolves to meet the requirements of different fields.

```
1   <?xml version = "1.0"?>
2
3   <!-- Fig. 14.2: article.xml -->
4   <!-- Article structured with XML -->
5   <article>
6      <title>Simple XML</title>
7      <date>July 4, 2007</date>
8      <author>
9         <firstName>John</firstName>
10        <lastName>Doe</lastName>
11     </author>
12     <summary>XML is pretty easy.</summary>
13     <content>This chapter presents examples that use XML.</content>
14  </article>
```

Fig. 14.2 | XML used to mark up an article.

Portability Tip 14.1

Documents should include the XML declaration to identify the version of XML used. A document that lacks an XML declaration might be assumed to conform to the latest version of XML—when it does not, errors could result.

As in most markup languages, blank lines (line 2), white spaces and indentation help improve readability. Blank lines are normally ignored by XML parsers. XML comments (lines 3–4), which begin with `<!--` and end with `-->`, can be placed almost anywhere in an XML document and can span multiple lines. There must be exactly one end marker (`-->`) for each begin marker (`<!--`).

Common Programming Error 14.1

Placing any characters, including white space, before the XML declaration is an error.

Common Programming Error 14.2

In an XML document, each start tag must have a matching end tag; omitting either tag is an error. Soon, you will learn how such errors are detected.

Common Programming Error 14.3

XML is case sensitive. Using different cases for the start tag and end tag names for the same element is a syntax error.

In Fig. 14.2, `article` (lines 5–14) is the root element. The lines that precede the root element (lines 1–4) are the XML prolog. In an XML prolog, the XML declaration must appear before the comments and any other markup.

The elements we use in the example do not come from any specific markup language. Instead, we chose the element names and markup structure that best describe our particular data. You can invent elements to mark up your data. For example, element `title` (line 6) contains text that describes the article's title (e.g., `Simple XML`). Similarly, `date` (line 7), `author` (lines 8–11), `firstName` (line 9), `lastName` (line 10), `summary` (line 12) and `content` (line 13) contain text that describes the date, author, the author's first name, the author's last name, a summary and the content of the document, respectively. XML element names can be of any length and may contain letters, digits, underscores, hyphens and periods. However, they must begin with either a letter or an underscore, and they should not begin with "`xml`" in any combination of uppercase and lowercase letters (e.g., `XML`, `Xml`, `xMl`), as this is reserved for use in the XML standards.

Common Programming Error 14.4

Using a white-space character in an XML element name is an error.

Good Programming Practice 14.1

XML element names should be meaningful to humans and should not use abbreviations.

XML elements are nested to form hierarchies—with the root element at the top of the hierarchy. This allows document authors to create parent/child relationships between data. For example, elements `title`, `date`, `author`, `summary` and `content` are nested within article. Elements firstName and lastName are nested within author. We discuss the hierarchy of Fig. 14.2 later in this chapter (Fig. 14.25).

Common Programming Error 14.5

Nesting XML tags improperly is a syntax error. For example, `<x><y>hello</x></y>` *is an error, because the* `</y>` *tag must precede the* `</x>` *tag.*

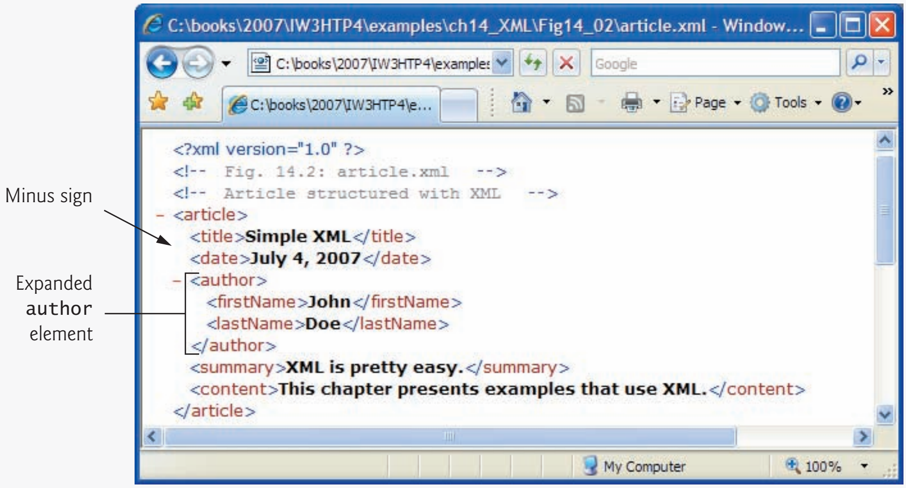
Any element that contains other elements (e.g., article or author) is a container element. Container elements also are called parent elements. Elements nested inside a container element are child elements (or children) of that container element. If those child elements are at the same nesting level, they are siblings of one another.

Viewing an XML Document in Internet Explorer and Firefox

The XML document in Fig. 14.2 is simply a text file named article.xml. This document does not contain formatting information for the article. This is because XML is a technology for describing the structure of data. Formatting and displaying data from an XML document are application-specific issues. For example, when the user loads article.xml in Internet Explorer, MSXML (Microsoft XML Core Services) parses and displays the document's data. Firefox has a similar capability. Each browser has a built-in style sheet to format the data. Note that the resulting format of the data (Fig. 14.3) is similar to the format of the listing in Fig. 14.2. In Section 14.8, we show how to create style sheets to transform your XML data into various formats suitable for display.

Note the minus sign (–) and plus sign (+) in the screen shots of Fig. 14.3. Although these symbols are not part of the XML document, both browsers place them next to every container element. A minus sign indicates that the browser is displaying the container element's child elements. Clicking the minus sign next to an element collapses that element

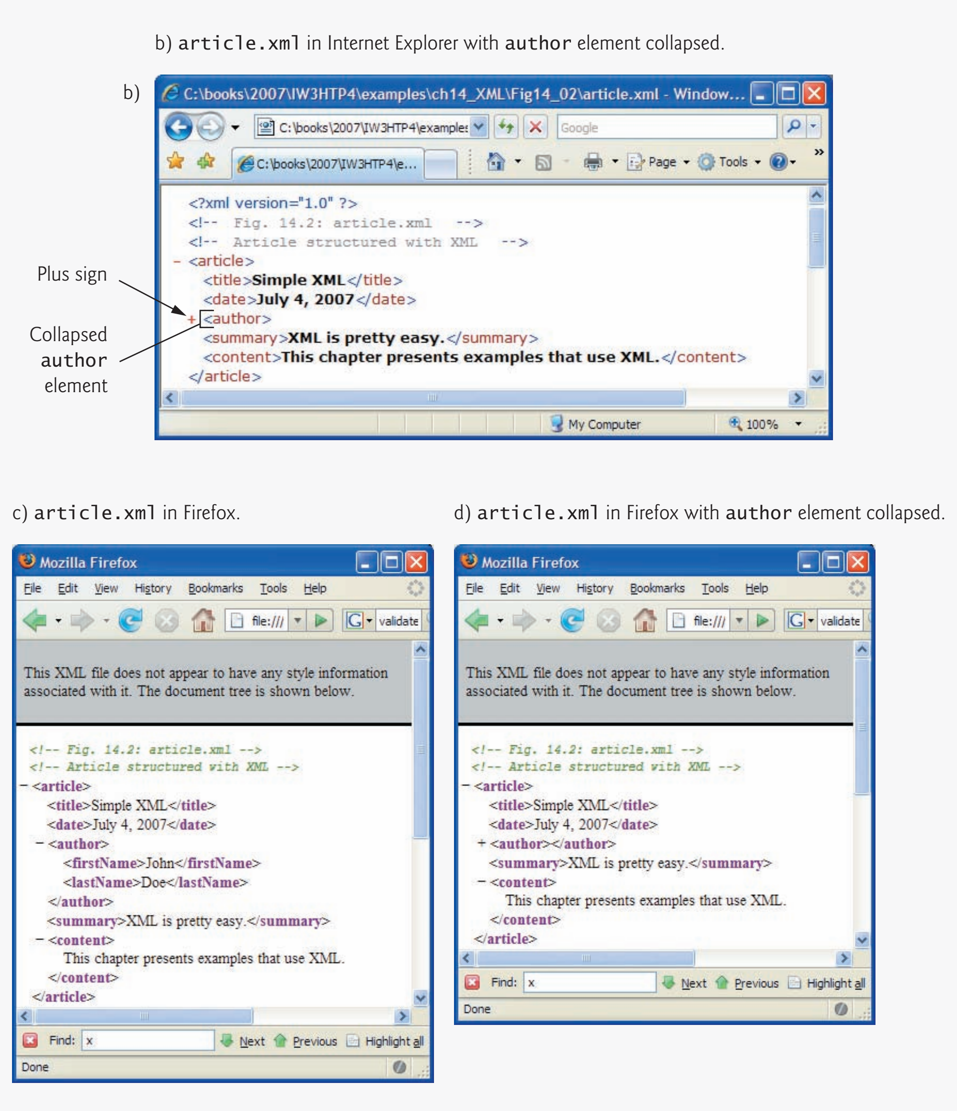
a) article.xml in Internet Explorer.



C:\books\2007\IW3HTP4\examples\ch14_XML\Fig14_02\article.xml - Window...
C:\books\2007\IW3HTP4\examples
Google
C:\books\2007\IW3HTP4\e...
Page
Tools
My Computer
100%
Minus sign
Expanded author element

```
<?xml version="1.0" ?>
<!-- Fig. 14.2: article.xml  -->
<!-- Article structured with XML  -->
- <article>
    <title>Simple XML</title>
    <date>July 4, 2007</date>
  - <author>
      <firstName>John</firstName>
      <lastName>Doe</lastName>
    </author>
    <summary>XML is pretty easy.</summary>
    <content>This chapter presents examples that use XML.</content>
  </article>
```

Fig. 14.3 | article.xml displayed by Internet Explorer 7 and Firefox 2. (Part 1 of 2.)



b) `article.xml` in Internet Explorer with `author` element collapsed.

b)

C:\books\2007\IW3HTP4\examples\ch14_XML\Fig14_02\article.xml - Window...
C:\books\2007\IW3HTP4\examples
Google
C:\books\2007\IW3HTP4\e...
Page
Tools
My Computer
100%

```
<?xml version="1.0" ?>
<!-- Fig. 14.2: article.xml  -->
<!-- Article structured with XML  -->
- <article>
    <title>Simple XML</title>
    <date>July 4, 2007</date>
  + <author>
    <summary>XML is pretty easy.</summary>
    <content>This chapter presents examples that use XML.</content>
  </article>
```

Plus sign

Collapsed `author` element

c) `article.xml` in Firefox.

Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///
validate
Find: x
Next
Previous
Highlight all
Done

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<!-- Fig. 14.2: article.xml -->
<!-- Article structured with XML -->
- <article>
    <title>Simple XML</title>
    <date>July 4, 2007</date>
  - <author>
      <firstName>John</firstName>
      <lastName>Doe</lastName>
    </author>
    <summary>XML is pretty easy.</summary>
  - <content>
      This chapter presents examples that use XML.
    </content>
  </article>
```

d) `article.xml` in Firefox with `author` element collapsed.

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<!-- Fig. 14.2: article.xml -->
<!-- Article structured with XML -->
- <article>
    <title>Simple XML</title>
    <date>July 4, 2007</date>
  + <author></author>
    <summary>XML is pretty easy.</summary>
  - <content>
      This chapter presents examples that use XML.
    </content>
  </article>
```

Fig. 14.3 | `article.xml` displayed by Internet Explorer 7 and Firefox 2. (Part 2 of 2.)

(i.e., causes the browser to hide the container element's children and replace the minus sign with a plus sign). Conversely, clicking the plus sign next to an element expands that element (i.e., causes the browser to display the container element's children and replace the plus sign with a minus sign). This behavior is similar to viewing the directory structure on your system in Windows Explorer or another similar directory viewer. In fact, a directory structure often is modeled as a series of tree structures, in which the **root** of a tree represents a disk drive (e.g., `C:`), and **nodes** in the tree represent directories. Parsers often store XML data as tree structures to facilitate efficient manipulation, as discussed in Section 14.9.

[*Note:* In Windows XP Service Pack 2 and Windows Vista, by default Internet Explorer displays all the XML elements in expanded view, and clicking the minus sign (Fig. 14.3(a)) does not do anything. To enable collapsing and expanding, right click the *Information Bar* that appears just below the **Address** field and select **Allow Blocked Content…**. Then click **Yes** in the pop-up window that appears.]

XML Markup for a Business Letter

Now that you've seen a simple XML document, let's examine a more complex XML document that marks up a business letter (Fig. 14.4). Again, we begin the document with the XML declaration (line 1) that states the XML version to which the document conforms.

```
 1  <?xml version = "1.0"?>
 2  
 3  <!-- Fig. 14.4: letter.xml -->
 4  <!-- Business letter marked up as XML -->
 5  <!DOCTYPE letter SYSTEM "letter.dtd">
 6  
 7  <letter>
 8     <contact type = "sender">
 9        <name>Jane Doe</name>
10        <address1>Box 12345</address1>
11        <address2>15 Any Ave.</address2>
12        <city>Othertown</city>
13        <state>Otherstate</state>
14        <zip>67890</zip>
15        <phone>555-4321</phone>
16        <flag gender = "F" />
17     </contact>
18  
19     <contact type = "receiver">
20        <name>John Doe</name>
21        <address1>123 Main St.</address1>
22        <address2></address2>
23        <city>Anytown</city>
24        <state>Anystate</state>
25        <zip>12345</zip>
26        <phone>555-1234</phone>
27        <flag gender = "M" />
28     </contact>
29  
30     <salutation>Dear Sir:</salutation>
31  
32     <paragraph>It is our privilege to inform you about our new database
33        managed with XML. This new system allows you to reduce the
34        load on your inventory list server by having the client machine
35        perform the work of sorting and filtering the data.
36     </paragraph>
37  
38     <paragraph>Please visit our website for availability and pricing.
39     </paragraph>
40  
```

Fig. 14.4 | Business letter marked up as XML. (Part 1 of 2.)

```
41      <closing>Sincerely,</closing>
42      <signature>Ms. Jane Doe</signature>
43   </letter>
```

Fig. 14.4 | Business letter marked up as XML. (Part 2 of 2.)

Line 5 specifies that this XML document references a DTD. Recall from Section 14.2 that DTDs define the structure of the data for an XML document. For example, a DTD specifies the elements and parent/child relationships between elements permitted in an XML document.

Error-Prevention Tip 14.1

An XML document is not required to reference a DTD, but validating XML parsers can use a DTD to ensure that the document has the proper structure.

Portability Tip 14.2

Validating an XML document helps guarantee that independent developers will exchange data in a standardized form that conforms to the DTD.

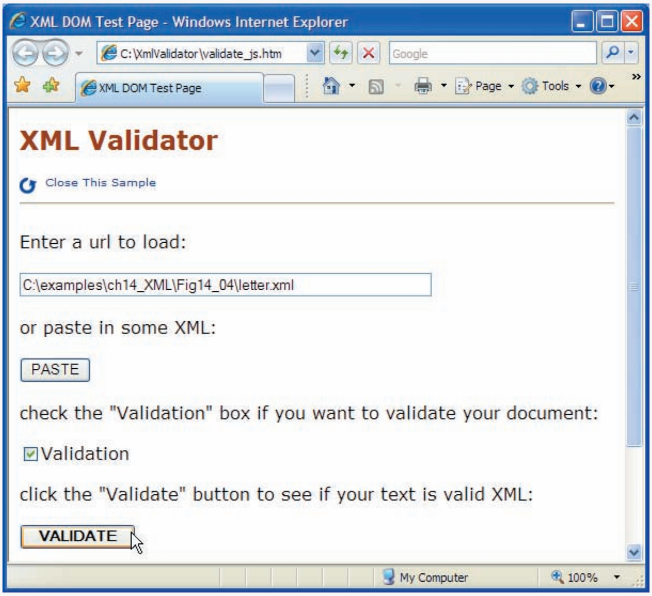
The `DOCTYPE` reference (line 5) contains three items, the name of the root element that the DTD specifies (`letter`); the keyword **`SYSTEM`** (which denotes an **external DTD**—a DTD declared in a separate file, as opposed to a DTD declared locally in the same file); and the DTD's name and location (i.e., `letter.dtd` in the current directory; this could also be a fully qualified URL). DTD document filenames typically end with the **`.dtd`** extension. We discuss DTDs and `letter.dtd` in detail in Section 14.5.

Several tools (many of which are free) validate documents against DTDs (discussed in Section 14.5) and schemas (discussed in Section 14.6). Microsoft's **XML Validator** is available free of charge from the **Download sample** link at

```
msdn.microsoft.com/archive/en-us/samples/internet/
xml/xml_validator/default.asp
```

This validator can validate XML documents against both DTDs and schemas. To install it, run the downloaded executable file `xml_validator.exe` and follow the steps to complete the installation. Once the installation is successful, open the `validate_js.htm` file located in your XML Validator installation directory in IE to validate your XML documents. We installed the XML Validator at `C:\XMLValidator` (Fig. 14.5). The output (Fig. 14.6) shows the results of validating the document using Microsoft's XML Validator. You can click a node to expand it and see its contents. Visit `www.w3.org/XML/Schema` for a list of additional validation tools.

Root element `letter` (lines 7–43 of Fig. 14.4) contains the child elements `contact`, `contact`, `salutation`, `paragraph`, `paragraph`, `closing` and `signature`. Data can be placed between an elements' tags or as **attributes**—name/value pairs that appear within the angle brackets of an element's start tag. Elements can have any number of attributes (separated by spaces) in their start tags. The first `contact` element (lines 8–17) has an attribute named `type` with **attribute value** `"sender"`, which indicates that this `contact` element identifies the letter's sender. The second `contact` element (lines 19–28) has attribute `type` with value `"receiver"`, which indicates that this `contact` element identifies the letter's recipient. Like element names, attribute names are case sensitive, can be any


XML DOM Test Page - Windows Internet Explorer
C:\XmlValidator\validate_js.htm
Google
XML DOM Test Page
Page
Tools
My Computer
100%

XML Validator

Close This Sample

Enter a url to load:

C:\examples\ch14_XML\Fig14_04\letter.xml

or paste in some XML:

PASTE

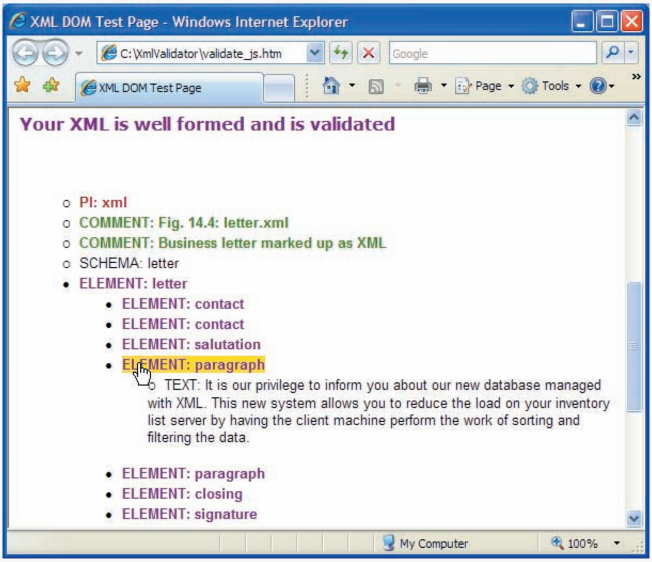
check the "Validation" box if you want to validate your document:

☑ Validation

click the "Validate" button to see if your text is valid XML:

VALIDATE

Fig. 14.5 | Validating an XML document with Microsoft's XML Validator.


XML DOM Test Page - Windows Internet Explorer
C:\XmlValidator\validate_js.htm
Google
XML DOM Test Page
Page
Tools
My Computer
100%

Your XML is well formed and is validated

- PI: xml
- COMMENT: Fig. 14.4: letter.xml
- COMMENT: Business letter marked up as XML
- SCHEMA: letter
- ELEMENT: letter
  - ELEMENT: contact
  - ELEMENT: contact
  - ELEMENT: salutation
  - ELEMENT: paragraph
    - TEXT: It is our privilege to inform you about our new database managed with XML. This new system allows you to reduce the load on your inventory list server by having the client machine perform the work of sorting and filtering the data.
  - ELEMENT: paragraph
  - ELEMENT: closing
  - ELEMENT: signature

Fig. 14.6 | Validation result using Microsoft's XML Validator.

length, may contain letters, digits, underscores, hyphens and periods, and must begin with either a letter or an underscore character. A `contact` element stores various items of information about a contact, such as the contact's name (represented by element `name`), address (represented by elements `address1`, `address2`, `city`, `state` and `zip`), phone number (represented by element `phone`) and gender (represented by attribute `gender` of element `flag`). Element `salutation` (line 30) marks up the letter's salutation. Lines 32–39 mark up the letter's body using two `paragraph` elements. Elements `closing` (line 41) and `signature` (line 42) mark up the closing sentence and the author's "signature," respectively.

Common Programming Error 14.6

Failure to enclose attribute values in double (`""`) or single (`''`) quotes is a syntax error.

Line 16 introduces the empty element `flag`. An empty element is one that does not have any content. Instead, an empty element sometimes places data in attributes. Empty element `flag` has one attribute that indicates the gender of the contact (represented by the parent `contact` element). Document authors can close an empty element either by placing a slash immediately preceding the right angle bracket, as shown in line 16, or by explicitly writing an end tag, as in line 22

```
<address2></address2>
```

Note that the `address2` element in line 22 is empty because there is no second part to this contact's address. However, we must include this element to conform to the structural rules specified in the XML document's DTD—`letter.dtd` (which we present in Section 14.5). This DTD specifies that each `contact` element must have an `address2` child element (even if it is empty). In Section 14.5, you will learn how DTDs indicate required and optional elements.

14.4 XML Namespaces

XML allows document authors to create custom elements. This extensibility can result in naming collisions among elements in an XML document that each have the same name. For example, we may use the element `book` to mark up data about a Deitel publication. A stamp collector may use the element `book` to mark up data about a book of stamps. Using both of these elements in the same document could create a naming collision, making it difficult to determine which kind of data each element contains.

An XML namespace is a collection of element and attribute names. XML namespaces provide a means for document authors to unambiguously refer to elements with the same name (i.e., prevent collisions). For example,

```
<subject>Geometry</subject>
```

and

```
<subject>Cardiology</subject>
```

use element `subject` to mark up data. In the first case, the subject is something one studies in school, whereas in the second case, the subject is a field of medicine. Namespaces can differentiate these two `subject` elements—for example:

```
<highschool:subject>Geometry</highschool:subject>
```

and

```
<medicalschool:subject>Cardiology</medicalschool:subject>
```

Both highschool and medicalschool are namespace prefixes. A document author places a namespace prefix and colon (:) before an element name to specify the namespace to which that element belongs. Document authors can create their own namespace prefixes using virtually any name except the reserved namespace prefix xml. In the next subsections, we demonstrate how document authors ensure that namespaces are unique.

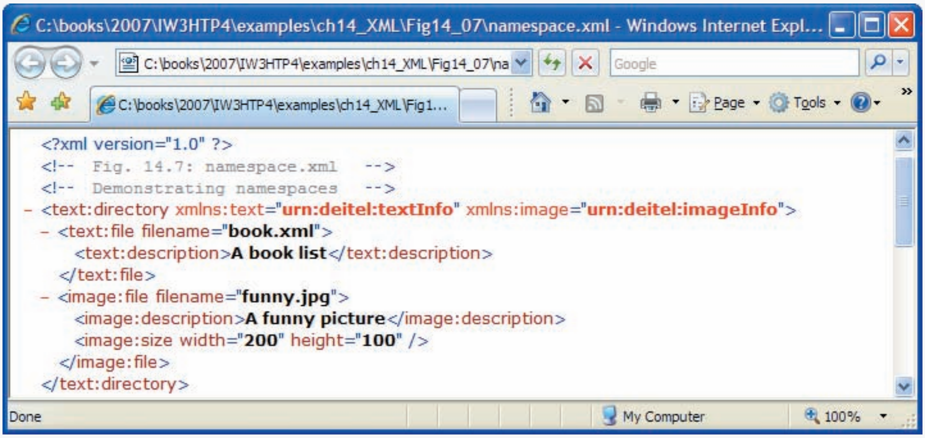
Common Programming Error 14.7

Attempting to create a namespace prefix named xml *in any mixture of uppercase and lowercase letters is a syntax error—the* xml *namespace prefix is reserved for internal use by XML itself.*

Differentiating Elements with Namespaces

Figure 14.7 demonstrates namespaces. In this document, namespaces differentiate two distinct elements—the file element related to a text file and the file document related to an image file.

```
<?xml version = "1.0"?>

<!-- Fig. 14.7: namespace.xml -->
<!-- Demonstrating namespaces -->
<text:directory
   xmlns:text = "urn:deitel:textInfo"
   xmlns:image = "urn:deitel:imageInfo">

   <text:file filename = "book.xml">
      <text:description>A book list</text:description>
   </text:file>

   <image:file filename = "funny.jpg">
      <image:description>A funny picture</image:description>
      <image:size width = "200" height = "100" />
   </image:file>
</text:directory>
```


C:\books\2007\IW3HTP4\examples\ch14_XML\Fig14_07\namespace.xml - Windows Internet Expl...
C:\books\2007\IW3HTP4\examples\ch14_XML\Fig14_07\na
Google
C:\books\2007\IW3HTP4\examples\ch14_XML\Fig1...
Page
Tools
Done
My Computer
100%

```
  <?xml version="1.0" ?>
  <!-- Fig. 14.7: namespace.xml  -->
  <!-- Demonstrating namespaces  -->
- <text:directory xmlns:text="urn:deitel:textInfo" xmlns:image="urn:deitel:imageInfo">
  - <text:file filename="book.xml">
      <text:description>A book list</text:description>
    </text:file>
  - <image:file filename="funny.jpg">
      <image:description>A funny picture</image:description>
      <image:size width="200" height="100" />
    </image:file>
  </text:directory>
```

Fig. 14.7 | XML namespaces demonstration.

Lines 6–7 use the XML-namespace reserved attribute **xmlns** to create two namespace prefixes—text and image. Each namespace prefix is bound to a series of characters called a **Uniform Resource Identifier (URI)** that uniquely identifies the namespace. Document authors create their own namespace prefixes and URIs. A URI is a way to identifying a resource, typically on the Internet. Two popular types of URI are **Uniform Resource Name (URN)** and **Uniform Resource Locator (URL)**.

To ensure that namespaces are unique, document authors must provide unique URIs. In this example, we use the text urn:deitel:textInfo and urn:deitel:imageInfo as URIs. These URIs employ the URN scheme frequently used to identify namespaces. Under this naming scheme, a URI begins with "urn:", followed by a unique series of additional names separated by colons.

Another common practice is to use URLs, which specify the location of a file or a resource on the Internet. For example, www.deitel.com is the URL that identifies the home page of the Deitel & Associates website. Using URLs guarantees that the namespaces are unique because the domain names (e.g., www.deitel.com) are guaranteed to be unique. For example, lines 5–7 could be rewritten as

```
<text:directory
   xmlns:text = "http://www.deitel.com/xmlns-text"
   xmlns:image = "http://www.deitel.com/xmlns-image">
```

where URLs related to the deitel.com domain name serve as URIs to identify the text and image namespaces. The parser does not visit these URLs, nor do these URLs need to refer to actual web pages. They each simply represent a unique series of characters used to differentiate URI names. In fact, any string can represent a namespace. For example, our image namespace URI could be hgjfkdlsa4556, in which case our prefix assignment would be

```
xmlns:image = "hgjfkdlsa4556"
```

Lines 9–11 use the text namespace prefix for elements file and description. Note that the end tags must also specify the namespace prefix text. Lines 13–16 apply namespace prefix image to the elements file, description and size. Note that attributes do not require namespace prefixes (although they can have them), because each attribute is already part of an element that specifies the namespace prefix. For example, attribute filename (line 9) is implicitly part of namespace text because its element (i.e., file) specifies the text namespace prefix.

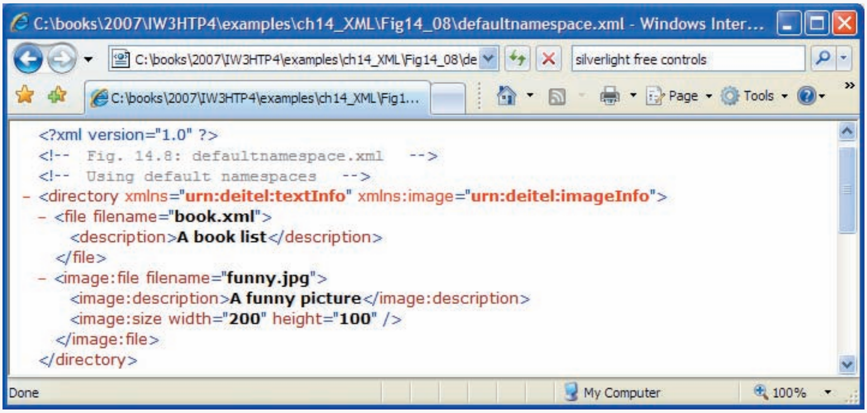
Specifying a Default Namespace

To eliminate the need to place namespace prefixes in each element, document authors may specify a **default namespace** for an element and its children. Figure 14.8 demonstrates using a default namespace (urn:deitel:textInfo) for element directory.

Line 5 defines a default namespace using attribute xmlns with no prefex specified, but with a URI as its value. Once we define this default namespace, child elements belonging to the namespace need not be qualified by a namespace prefix. Thus, element file (lines 8–10) is in the default namespace urn:deitel:textInfo. Compare this to lines 9–10 of Fig. 14.7, where we had to prefix the file and description element names with the namespace prefix text.

The default namespace applies to the directory element and all elements that are not qualified with a namespace prefix. However, we can use a namespace prefix to specify a

```
 1  <?xml version = "1.0"?>
 2
 3  <!-- Fig. 14.8: defaultnamespace.xml -->
 4  <!-- Using default namespaces -->
 5  <directory xmlns = "urn:deitel:textInfo"
 6     xmlns:image = "urn:deitel:imageInfo">
 7
 8     <file filename = "book.xml">
 9        <description>A book list</description>
10     </file>
11
12     <image:file filename = "funny.jpg">
13        <image:description>A funny picture</image:description>
14        <image:size width = "200" height = "100" />
15     </image:file>
16  </directory>
```


C:\books\2007\IW3HTP4\examples\ch14_XML\Fig14_08\defaultnamespace.xml - Windows Inter...
C:\books\2007\IW3HTP4\examples\ch14_XML\Fig14_08\de
silverlight free controls
C:\books\2007\IW3HTP4\examples\ch14_XML\Fig1...
Page
Tools
Done
My Computer
100%

```
  <?xml version="1.0" ?>
  <!--  Fig. 14.8: defaultnamespace.xml  -->
  <!--  Using default namespaces  -->
- <directory xmlns="urn:deitel:textInfo" xmlns:image="urn:deitel:imageInfo">
  - <file filename="book.xml">
      <description>A book list</description>
    </file>
  - <image:file filename="funny.jpg">
      <image:description>A funny picture</image:description>
      <image:size width="200" height="100" />
    </image:file>
  </directory>
```

Fig. 14.8 | Default namespace demonstration.

different namespace for a particular element. For example, the `file` element in lines 12–15 includes the `image` namespace prefix, indicating that this element is in the `urn:deitel:imageInfo` namespace, not the default namespace.

Namespaces in XML Vocabularies

XML-based languages, such as XML Schema (Section 14.6) and Extensible Stylesheet Language (XSL) (Section 14.8), often use namespaces to identify their elements. Each of these vocabularies defines special-purpose elements that are grouped in namespaces. These namespaces help prevent naming collisions between predefined elements and user-defined elements.

14.5 Document Type Definitions (DTDs)

Document Type Definitions (DTDs) are one of two main types of documents you can use to specify XML document structure. Section 14.6 presents W3C XML Schema documents, which provide an improved method of specifying XML document structure.

Software Engineering Observation 14.2

XML documents can have many different structures, and for this reason an application cannot be certain whether a particular document it receives is complete, ordered properly, and not missing data. DTDs and schemas (Section 14.6) solve this problem by providing an extensible way to describe XML document structure. Applications should use DTDs or schemas to confirm whether XML documents are valid.

Software Engineering Observation 14.3

Many organizations and individuals are creating DTDs and schemas for a broad range of applications. These collections—called ***repositories****—are available free for download from the web (e.g.,* `www.xml.org`*,* `www.oasis-open.org`*).*

Creating a Document Type Definition

Figure 14.4 presented a simple business letter marked up with XML. Recall that line 5 of `letter.xml` references a DTD—`letter.dtd` (Fig. 14.9). This DTD specifies the business letter's element types and attributes, and their relationships to one another.

A DTD describes the structure of an XML document and enables an XML parser to verify whether an XML document is valid (i.e., whether its elements contain the proper attributes and appear in the proper sequence). DTDs allow users to check document structure and to exchange data in a standardized format. A DTD expresses the set of rules for document structure using an EBNF (Extended Backus-Naur Form) grammar. DTDs are not themselves XML documents. [*Note:* EBNF grammars are commonly used to define programming languages. To learn more about EBNF grammars, visit `en.wikipedia.org/wiki/EBNF` or `www.garshol.priv.no/download/text/bnf.html`.]

```
<!-- Fig. 14.9: letter.dtd       -->
<!-- DTD document for letter.xml -->

<!ELEMENT letter ( contact+, salutation, paragraph+,
   closing, signature )>

<!ELEMENT contact ( name, address1, address2, city, state,
   zip, phone, flag )>
<!ATTLIST contact type CDATA #IMPLIED>

<!ELEMENT name ( #PCDATA )>
<!ELEMENT address1 ( #PCDATA )>
<!ELEMENT address2 ( #PCDATA )>
<!ELEMENT city ( #PCDATA )>
<!ELEMENT state ( #PCDATA )>
<!ELEMENT zip ( #PCDATA )>
<!ELEMENT phone ( #PCDATA )>
<!ELEMENT flag EMPTY>
<!ATTLIST flag gender (M | F) "M">

<!ELEMENT salutation ( #PCDATA )>
<!ELEMENT closing ( #PCDATA )>
<!ELEMENT paragraph ( #PCDATA )>
<!ELEMENT signature ( #PCDATA )>
```

Fig. 14.9 | Document Type Definition (DTD) for a business letter.

Common Programming Error 14.8

For documents validated with DTDs, any document that uses elements, attributes or nesting relationships not explicitly defined by a DTD is an invalid document.

Defining Elements in a DTD

The **ELEMENT element type declaration** in lines 4–5 defines the rules for element `letter`. In this case, `letter` contains one or more `contact` elements, one `salutation` element, one or more `paragraph` elements, one `closing` element and one `signature` element, in that sequence. The **plus sign (+) occurrence indicator** specifies that the DTD requires one or more occurrences of an element. Other occurence indicators include the **asterisk** (*), which indicates an optional element that can occur zero or more times, and the **question mark** (?), which indicates an optional element that can occur at most once (i.e., zero or one occurrence). If an element does not have an occurrence indicator, the DTD requires exactly one occurrence.

The `contact` element type declaration (lines 7–8) specifies that a `contact` element contains child elements `name`, `address1`, `address2`, `city`, `state`, `zip`, `phone` and `flag`—in that order. The DTD requires exactly one occurrence of each of these elements.

Defining Attributes in a DTD

Line 9 uses the **ATTLIST attribute-list declaration** to define an attribute named `type` for the `contact` element. Keyword **#IMPLIED** specifies that if the parser finds a `contact` element without a `type` attribute, the parser can choose an arbitrary value for the attribute or can ignore the attribute. Either way the document will still be valid (if the rest of the document is valid)—a missing `type` attribute will not invalidate the document. Other keywords that can be used in place of `#IMPLIED` in an `ATTLIST` declaration include **#REQUIRED** and **#FIXED**. Keyword `#REQUIRED` specifies that the attribute must be present in the element, and keyword `#FIXED` specifies that the attribute (if present) must have the given fixed value. For example,

```
<!ATTLIST address zip CDATA #FIXED "01757">
```

indicates that attribute `zip` (if present in element `address`) must have the value `01757` for the document to be valid. If the attribute is not present, then the parser, by default, uses the fixed value that the `ATTLIST` declaration specifies.

Character Data vs. Parsed Character Data

Keyword **CDATA** (line 9) specifies that attribute `type` contains **character data** (i.e., a string). A parser will pass such data to an application without modification.

Software Engineering Observation 14.4

DTD syntax cannot describe an element's or attribute's data type. For example, a DTD cannot specify that a particular element or attribute can contain only integer data.

Keyword **#PCDATA** (line 11) specifies that an element (e.g., `name`) may contain **parsed character data** (i.e., data that is processed by an XML parser). Elements with parsed character data cannot contain markup characters, such as less than (<), greater than (>) or ampersand (&). The document author should replace any markup character in a `#PCDATA` element with the character's corresponding **character entity reference**. For example, the character entity reference `&lt;` should be used in place of the less-than symbol (<), and the character entity reference `&gt;` should be used in place of the greater-than symbol (>). A document author who wishes to use a literal ampersand should use the entity reference `&amp;` instead—parsed character data can contain ampersands (&) only for inserting entities. See Appendix A, XHTML Special Characters, for a list of other character entity references.

Common Programming Error 14.9

Using markup characters (e.g., <, > and &) in parsed character data is an error. Use character entity references (e.g., `&lt;`*,* `&gt;` *and* `&amp;`*) instead.*

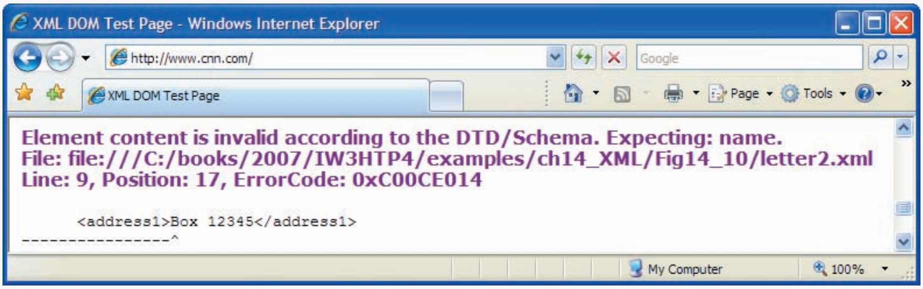
Defining Empty Elements in a DTD

Line 18 defines an empty element named `flag`. Keyword **`EMPTY`** specifies that the element does not contain any data between its start and end tags. Empty elements commonly describe data via attributes. For example, `flag`'s data appears in its `gender` attribute (line 19). Line 19 specifies that the `gender` attribute's value must be one of the enumerated values (`M` or `F`) enclosed in parentheses and delimited by a vertical bar (`|`) meaning "or." Note that line 19 also indicates that `gender` has a default value of `M`.

Well-Formed Documents vs. Valid Documents

In Section 14.3, we demonstrated how to use the Microsoft XML Validator to validate an XML document against its specified DTD. The validation revealed that the XML document `letter.xml` (Fig. 14.4) is well-formed and valid—it conforms to `letter.dtd` (Fig. 14.9). Recall that a well-formed document is syntactically correct (i.e., each start tag has a corresponding end tag, the document contains only one root element, etc.), and a valid document contains the proper elements with the proper attributes in the proper sequence. An XML document cannot be valid unless it is well-formed.

When a document fails to conform to a DTD or a schema, the Microsoft XML Validator displays an error message. For example, the DTD in Fig. 14.9 indicates that a `contact` element must contain the child element `name`. A document that omits this child element is still well-formed, but is not valid. In such a scenario, Microsoft XML Validator displays the error message shown in Fig. 14.10.



```
Element content is invalid according to the DTD/Schema. Expecting: name.
File: file:///C:/books/2007/IW3HTP4/examples/ch14_XML/Fig14_10/letter2.xml
Line: 9, Position: 17, ErrorCode: 0xC00CE014

      <address1>Box 12345</address1>
----------------^
```

Fig. 14.10 | XML Validator displaying an error message.

14.6 W3C XML Schema Documents

In this section, we introduce schemas for specifying XML document structure and validating XML documents. Many developers in the XML community believe that DTDs are not flexible enough to meet today's programming needs. For example, DTDs lack a way of indicating what specific type of data (e.g., numeric, text) an element can contain, and DTDs are not themselves XML documents, forcing developers to learn multiple grammars and developers to create multiple types of parsers. These and other limitations have led to the development of schemas.

Unlike DTDs, schemas do not use EBNF grammar. Instead, schemas use XML syntax and are actually XML documents that programs can manipulate. Like DTDs, schemas are used by validating parsers to validate documents.

In this section, we focus on the W3C's **XML Schema** vocabulary (note the capital "S" in "Schema"). We use the term XML Schema in the rest of the chapter whenever we refer to W3C's XML Schema vocabulary. For the latest information on XML Schema, visit `www.w3.org/XML/Schema`. For tutorials on XML Schema concepts beyond what we present here, visit `www.w3schools.com/schema/default.asp`.

Recall that a DTD describes an XML document's structure, not the content of its elements. For example,

```
<quantity>5</quantity>
```

contains character data. If the document that contains element `quantity` references a DTD, an XML parser can validate the document to confirm that this element indeed does contain `PCDATA` content. However, the parser cannot validate that the content is numeric; DTDs do not provide this capability. So, unfortunately, the parser also considers

```
<quantity>hello</quantity>
```

to be valid. An application that uses the XML document containing this markup should test that the data in element `quantity` is numeric and take appropriate action if it is not.

XML Schema enables schema authors to specify that element `quantity`'s data must be numeric or, even more specifically, an integer. A parser validating the XML document against this schema can determine that `5` conforms and `hello` does not. An XML document that conforms to a schema document is **schema valid**, and one that does not conform is **schema invalid**. Schemas are XML documents and therefore must themselves be valid.

Validating Against an XML Schema Document

Figure 14.11 shows a schema-valid XML document named `book.xml`, and Fig. 14.12 shows the pertinent XML Schema document (`book.xsd`) that defines the structure for `book.xml`. By convention, schemas use the **`.xsd`** extension. We used an online XSD schema validator provided at

```
www.xmlforasp.net/SchemaValidator.aspx
```

to ensure that the XML document in Fig. 14.11 conforms to the schema in Fig. 14.12. To validate the schema document itself (i.e., `book.xsd`) and produce the output shown in Fig. 14.12, we used an online XSV (XML Schema Validator) provided by the W3C at

```
www.w3.org/2001/03/webdata/xsv
```

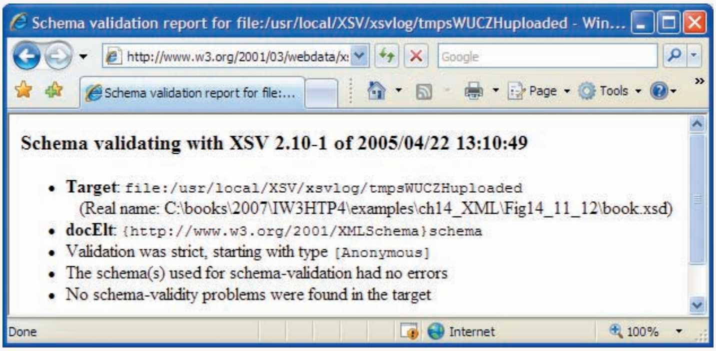
These tools are free and enforce the W3C's specifications regarding XML Schemas and schema validation.

```xml
<?xml version = "1.0"?>

<!-- Fig. 14.11: book.xml -->
<!-- Book list marked up as XML -->
<deitel:books xmlns:deitel = "http://www.deitel.com/booklist">
   <book>
      <title>Visual Basic 2005 How to Program, 3/e</title>
   </book>
   <book>
      <title>Visual C# 2005 How to Program, 2/e</title>
   </book>
   <book>
      <title>Java How to Program, 7/e</title>
   </book>
   <book>
      <title>C++ How to Program, 6/e</title>
   </book>
   <book>
      <title>Internet and World Wide Web How to Program, 4/e</title>
   </book>
</deitel:books>
```

Fig. 14.11 | Schema-valid XML document describing a list of books.

```xml
<?xml version = "1.0"?>

<!-- Fig. 14.12: book.xsd           -->
<!-- Simple W3C XML Schema document -->
<schema xmlns = "http://www.w3.org/2001/XMLSchema"
   xmlns:deitel = "http://www.deitel.com/booklist"
   targetNamespace = "http://www.deitel.com/booklist">

   <element name = "books" type = "deitel:BooksType"/>

   <complexType name = "BooksType">
      <sequence>
         <element name = "book" type = "deitel:SingleBookType"
            minOccurs = "1" maxOccurs = "unbounded"/>
      </sequence>
   </complexType>

   <complexType name = "SingleBookType">
      <sequence>
         <element name = "title" type = "string"/>
      </sequence>
   </complexType>
</schema>
```

Fig. 14.12 | XML Schema document for `book.xml`. (Part 1 of 2.)


Schema validation report for file:/usr/local/XSV/xsvlog/tmpsWUCZHuploaded - Win...
http://www.w3.org/2001/03/webdata/x:
Google
Schema validation report for file:...
Page
Tools
Done
Internet
100%

Schema validating with XSV 2.10-1 of 2005/04/22 13:10:49

- **Target**: `file:/usr/local/XSV/xsvlog/tmpsWUCZHuploaded`
  (Real name: C:\books\2007\IW3HTP4\examples\ch14_XML\Fig14_11_12\book.xsd)
- **docElt**: `{http://www.w3.org/2001/XMLSchema}schema`
- Validation was strict, starting with type `[Anonymous]`
- The schema(s) used for schema-validation had no errors
- No schema-validity problems were found in the target

Fig. 14.12 | XML Schema document for `book.xml`. (Part 2 of 2.)

Figure 14.11 contains markup describing several Deitel books. The `books` element (line 5) has the namespace prefix `deitel`, indicating that the `books` element is a part of the `http://www.deitel.com/booklist` namespace.

Creating an XML Schema Document

Figure 14.12 presents the XML Schema document that specifies the structure of `book.xml` (Fig. 14.11). This document defines an XML-based language (i.e., a vocabulary) for writing XML documents about collections of books. The schema defines the elements, attributes and parent/child relationships that such a document can (or must) include. The schema also specifies the type of data that these elements and attributes may contain.

Root element **`schema`** (Fig. 14.12, lines 5–23) contains elements that define the structure of an XML document such as `book.xml`. Line 5 specifies as the default namespace the standard W3C XML Schema namespace URI—**`http://www.w3.org/2001/XMLSchema`**. This namespace contains predefined elements (e.g., root-element `schema`) that comprise the XML Schema vocabulary—the language used to write an XML Schema document.

Portability Tip 14.3

W3C XML Schema authors specify URI `http://www.w3.org/2001/XMLSchema` when referring to the XML Schema namespace. This namespace contains predefined elements that comprise the XML Schema vocabulary. Specifying this URI ensures that validation tools correctly identify XML Schema elements and do not confuse them with those defined by document authors.

Line 6 binds the URI `http://www.deitel.com/booklist` to namespace prefix `deitel`. As we discuss momentarily, the schema uses this namespace to differentiate names created by us from names that are part of the XML Schema namespace. Line 7 also specifies `http://www.deitel.com/booklist` as the **`targetNamespace`** of the schema. This attribute identifies the namespace of the XML vocabulary that this schema defines. Note that the `targetNamespace` of `book.xsd` is the same as the namespace referenced in line 5 of `book.xml` (Fig. 14.11). This is what “connects” the XML document with the schema that defines its structure. When an XML schema validator examines `book.xml` and `book.xsd`, it will recognize that `book.xml` uses elements and attributes from the `http://www.deitel.com/booklist` namespace. The validator also will recognize that this namespace is the namespace defined in `book.xsd` (i.e., the schema’s `targetNamespace`).

Thus the validator knows where to look for the structural rules for the elements and attributes used in `book.xml`.

Defining an Element in XML Schema

In XML Schema, the **`element`** tag (line 9) defines an element to be included in an XML document that conforms to the schema. In other words, `element` specifies the actual *elements* that can be used to mark up data. Line 9 defines the `books` element, which we use as the root element in `book.xml` (Fig. 14.11). Attributes **`name`** and **`type`** specify the `element`'s name and type, respectively. An element's type indicates the data that the element may contain. Possible types include XML Schema-defined types (e.g., `string`, `double`) and user-defined types (e.g., `BooksType`, which is defined in lines 11–16). Figure 14.13 lists several of XML Schema's many built-in types. For a complete list of built-in types, see Section 3 of the specification found at `www.w3.org/TR/xmlschema-2`.

XML Schema type	Description	Ranges or structures	Examples
`string`	A character string		`"hello"`
`boolean`	True or false	`true`, `false`	`true`
`decimal`	A decimal numeral	`i` * (10^n), where `i` is an integer and `n` is an integer that is less than or equal to zero.	`5`, `-12`, `-45.78`
`float`	A floating-point number	`m` * (2^e), where `m` is an integer whose absolute value is less than 2^{24} and `e` is an integer in the range `-149` to `104`. Plus three additional numbers: positive infinity, negative infinity and not-a-number (`NaN`).	`0`, `12`, `-109.375`, `NaN`
`double`	A floating-point number	`m` * (2^e), where `m` is an integer whose absolute value is less than 2^{53} and `e` is an integer in the range `-1075` to `970`. Plus three additional numbers: positive infinity, negative infinity and not-a-number (`NaN`).	`0`, `12`, `-109.375`, `NaN`
`long`	A whole number	`-9223372036854775808` to `9223372036854775807`, inclusive.	`1234567890`, `-1234567890`
`int`	A whole number	`-2147483648` to `2147483647`, inclusive.	`1234567890`, `-1234567890`

Fig. 14.13 | Some XML Schema types. (Part 1 of 2.)

XML Schema type	Description	Ranges or structures	Examples
`short`	A whole number	`-32768` to `32767`, inclusive.	`12, -345`
`date`	A date consisting of a year, month and day	`yyyy-mm` with an optional `dd` and an optional time zone, where `yyyy` is four digits long and `mm` and `dd` are two digits long.	`2005-05-10`
`time`	A time consisting of hours, minutes and seconds	`hh:mm:ss` with an optional time zone, where `hh`, `mm` and `ss` are two digits long.	`16:30:25-05:00`

Fig. 14.13 | Some XML Schema types. (Part 2 of 2.)

In this example, `books` is defined as an element of type `deitel:BooksType` (line 9). `BooksType` is a user-defined type (lines 11–16) in the `http://www.deitel.com/booklist` namespace and therefore must have the namespace prefix `deitel`. It is not an existing XML Schema type.

Two categories of type exist in XML Schema—**simple types** and **complex types**. Simple and complex types differ only in that simple types cannot contain attributes or child elements and complex types can.

A user-defined type that contains attributes or child elements must be defined as a complex type. Lines 11–16 use element **`complexType`** to define `BooksType` as a complex type that has a child element named `book`. The `sequence` element (lines 12–15) allows you to specify the sequential order in which child elements must appear. The `element` (lines 13–14) nested within the `complexType` element indicates that a `BooksType` element (e.g., `books`) can contain child elements named `book` of type `deitel:SingleBookType` (defined in lines 18–22). Attribute **`minOccurs`** (line 14), with value `1`, specifies that elements of type `BooksType` must contain a minimum of one `book` element. Attribute **`maxOccurs`** (line 14), with value **`unbounded`**, specifies that elements of type `BooksType` may have any number of `book` child elements.

Lines 18–22 define the complex type `SingleBookType`. An element of this type contains a child element named `title`. Line 20 defines element `title` to be of simple type `string`. Recall that elements of a simple type cannot contain attributes or child elements. The `schema` end tag (`</schema>`, line 23) declares the end of the XML Schema document.

A Closer Look at Types in XML Schema

Every element in XML Schema has a type. Types include the built-in types provided by XML Schema (Fig. 14.13) or user-defined types (e.g., `SingleBookType` in Fig. 14.12).

Every simple type defines a **restriction** on an XML Schema-defined type or a restriction on a user-defined type. Restrictions limit the possible values that an element can hold.

Complex types are divided into two groups—those with **simple content** and those with **complex content**. Both can contain attributes, but only complex content can contain child elements. Complex types with simple content must extend or restrict some other existing type. Complex types with complex content do not have this limitation. We demonstrate complex types with each kind of content in the next example.

The schema document in Fig. 14.14 creates both simple types and complex types. The XML document in Fig. 14.15 (`laptop.xml`) follows the structure defined in Fig. 14.14 to describe parts of a laptop computer. A document such as `laptop.xml` that conforms to a schema is known as an **XML instance document**—the document is an instance (i.e., example) of the schema.

Line 5 declares the default namespace to be the standard XML Schema namespace—any elements without a prefix are assumed to be in the XML Schema namespace. Line 6 binds the namespace prefix `computer` to the namespace `http://www.deitel.com/computer`. Line 7 identifies this namespace as the `targetNamespace`—the namespace being defined by the current XML Schema document.

To design the XML elements for describing laptop computers, we first create a simple type in lines 9–13 using the **`simpleType`** element. We name this `simpleType` `gigahertz`

```
 1  <?xml version = "1.0"?>
 2  <!-- Fig. 14.14: computer.xsd -->
 3  <!-- W3C XML Schema document  -->
 4
 5  <schema xmlns = "http://www.w3.org/2001/XMLSchema"
 6     xmlns:computer = "http://www.deitel.com/computer"
 7     targetNamespace = "http://www.deitel.com/computer">
 8
 9     <simpleType name = "gigahertz">
10        <restriction base = "decimal">
11           <minInclusive value = "2.1"/>
12        </restriction>
13     </simpleType>
14
15     <complexType name = "CPU">
16        <simpleContent>
17           <extension base = "string">
18              <attribute name = "model" type = "string"/>
19           </extension>
20        </simpleContent>
21     </complexType>
22
23     <complexType name = "portable">
24        <all>
25           <element name = "processor" type = "computer:CPU"/>
26           <element name = "monitor" type = "int"/>
27           <element name = "CPUSpeed" type = "computer:gigahertz"/>
28           <element name = "RAM" type = "int"/>
29        </all>
30        <attribute name = "manufacturer" type = "string"/>
31     </complexType>
32
33     <element name = "laptop" type = "computer:portable"/>
34  </schema>
```

Fig. 14.14 | XML Schema document defining simple and complex types.

because it will be used to describe the clock speed of the processor in gigahertz. Simple types are restrictions of a type typically called a **base type**. For this `simpleType`, line 10 declares the base type as `decimal`, and we restrict the value to be at least `2.1` by using the **`minInclusive`** element in line 11.

Next, we declare a `complexType` named `CPU` that has **`simpleContent`** (lines 16–20). Remember that a complex type with simple content can have attributes but not child elements. Also recall that complex types with simple content must extend or restrict some XML Schema type or user-defined type. The **`extension`** element with attribute **`base`** (line 17) sets the base type to `string`. In this `complexType`, we extend the base type `string` with an attribute. The **`attribute`** element (line 18) gives the `complexType` an attribute of type `string` named `model`. Thus an element of type `CPU` must contain `string` text (because the base type is `string`) and may contain a `model` attribute that is also of type `string`.

Last, we define type `portable`, which is a `complexType` with complex content (lines 23–31). Such types are allowed to have child elements and attributes. The element **`all`** (lines 24–29) encloses elements that must each be included once in the corresponding XML instance document. These elements can be included in any order. This complex type holds four elements—`processor`, `monitor`, `CPUSpeed` and `RAM`. They are given types `CPU`, `int`, `gigahertz` and `int`, respectively. When using types `CPU` and `gigahertz`, we must include the namespace prefix `computer`, because these user-defined types are part of the `computer` namespace (`http://www.deitel.com/computer`)—the namespace defined in the current document (line 7). Also, `portable` contains an attribute defined in line 30. The `attribute` element indicates that elements of type `portable` contain an attribute of type `string` named `manufacturer`.

Line 33 declares the actual element that uses the three types defined in the schema. The element is called `laptop` and is of type `portable`. We must use the namespace prefix `computer` in front of `portable`.

We have now created an element named `laptop` that contains child elements `processor`, `monitor`, `CPUSpeed` and `RAM`, and an attribute `manufacturer`. Figure 14.15 uses the `laptop` element defined in the `computer.xsd` schema. Once again, we used an online XSD schema validator (`www.xmlforasp.net/SchemaValidator.aspx`) to ensure that this XML instance document adheres to the schema's structural rules.

Line 5 declares namespace prefix `computer`. The `laptop` element requires this prefix because it is part of the `http://www.deitel.com/computer` namespace. Line 6 sets the

```
1   <?xml version = "1.0"?>
2
3   <!-- Fig. 14.15: laptop.xml            -->
4   <!-- Laptop components marked up as XML -->
5   <computer:laptop xmlns:computer = "http://www.deitel.com/computer"
6      manufacturer = "IBM">
7
8      <processor model = "Centrino">Intel</processor>
9      <monitor>17</monitor>
10     <CPUSpeed>2.4</CPUSpeed>
11     <RAM>256</RAM>
12  </computer:laptop>
```

Fig. 14.15 | XML document using the `laptop` element defined in `computer.xsd`.

laptop's `manufacturer` attribute, and lines 8–11 use the elements defined in the schema to describe the laptop's characteristics.

This section introduced W3C XML Schema documents for defining the structure of XML documents, and we validated XML instance documents against schemas using an online XSD schema validator. Section 14.7 discusses several XML vocabularies and demonstrates the MathML vocabulary. Section 14.10 demonstrates the RSS vocabulary.

14.7 XML Vocabularies

XML allows authors to create their own tags to describe data precisely. People and organizations in various fields of study have created many different kinds of XML for structuring data. Some of these markup languages are: **MathML (Mathematical Markup Language)**, **Scalable Vector Graphics (SVG)**, **Wireless Markup Language (WML)**, **Extensible Business Reporting Language (XBRL)**, **Extensible User Interface Language (XUL)** and **Product Data Markup Language (PDML)**. Two other examples of XML vocabularies are W3C XML Schema and the Extensible Stylesheet Language (XSL), which we discuss in Section 14.8. The following subsections describe MathML and other custom markup languages.

14.7.1 MathML™

Until recently, computers typically required specialized software packages such as TeX and LaTeX for displaying complex mathematical expressions. This section introduces MathML, which the W3C developed for describing mathematical notations and expressions. One application that can parse, render and edit MathML is the W3C's **Amaya**™ browser/editor, which can be downloaded from

```
www.w3.org/Amaya/User/BinDist.html
```

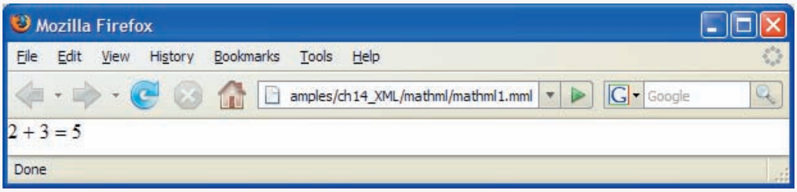
This page contains download links for several platforms. Amaya documentation and installation notes also are available at the W3C website. Firefox also can render MathML, but it requires additional fonts. Instructions for downloading and installing these fonts are available at `www.mozilla.org/projects/mathml/fonts/`. You can download a plug-in (`www.dessci.com/en/products/mathplayer/`) to render MathML in Internet Explorer .

MathML markup describes mathematical expressions for display. MathML is divided into two types of markup—**content** markup and **presentation** markup. Content markup provides tags that embody mathematical concepts. Content MathML allows programmers to write mathematical notation specific to different areas of mathematics. For instance, the multiplication symbol has one meaning in set theory and another meaning in linear algebra. Content MathML distinguishes between different uses of the same symbol. Programmers can take content MathML markup, discern mathematical context and evaluate the marked-up mathematical operations. Presentation MathML is directed toward formatting and displaying mathematical notation. We focus on Presentation MathML in the MathML examples.

Simple Equation in MathML

Figure 14.16 uses MathML to mark up a simple expression. For this example, we show the expression rendered in Firefox.

```
1  <?xml version="1.0" encoding="iso-8859-1"?>
2  <!DOCTYPE math PUBLIC "-//W3C//DTD MathML 2.0//EN"
3     "http://www.w3.org/TR/MathML2/dtd/mathml2.dtd">
4
5  <!-- Fig. 14.16: mathml1.mml -->
6  <!-- MathML equation. -->
7  <math xmlns="http://www.w3.org/1998/Math/MathML">
8     <mn>2</mn>
9     <mo>+</mo>
10    <mn>3</mn>
11    <mo>=</mo>
12    <mn>5</mn>
13 </math>
```


Mozilla Firefox
File Edit View History Bookmarks Tools Help
amples/ch14_XML/mathml/mathml1.mml
Google

2 + 3 = 5

Done

Fig. 14.16 | Expression marked up with MathML and displayed in the Firefox browser.

By convention, MathML files end with the `.mml` filename extension. A MathML document's root node is the `math` element, and its default namespace is `http://www.w3.org/1998/Math/MathML` (line 7). The **`mn` element** (line 8) marks up a number. The **`mo` element** (line 9) marks up an operator (e.g., +). Using this markup, we define the expression 2 + 3 = 5, which any MathML capable browser can display.

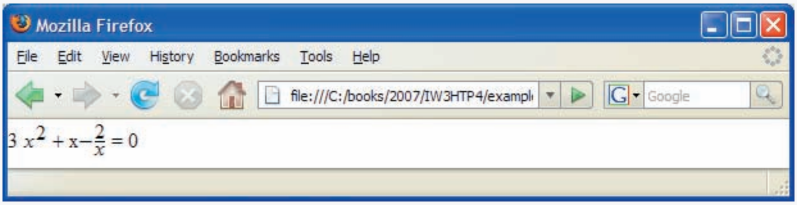
Algebraic Equation in MathML

Let's consider using MathML to mark up an algebraic equation containing exponents and arithmetic operators (Fig. 14.17). For this example, we again show the expression rendered in Firefox.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE math PUBLIC "-//W3C//DTD MathML 2.0//EN"
   "http://www.w3.org/TR/MathML2/dtd/mathml2.dtd">

<!-- Fig. 14.17: mathml2.html -->
<!-- MathML algebraic equation. -->
<math xmlns="http://www.w3.org/1998/Math/MathML">
   <mn>3</mn>
   <mo>&InvisibleTimes;</mo>
   <msup>
      <mi>x</mi>
      <mn>2</mn>
   </msup>
```

Fig. 14.17 | Algebraic equation marked up with MathML and displayed in the Firefox browser. (Part 1 of 2.)

```
14       <mo>+</mo>
15       <mn>x</mn>
16       <mo>&minus;</mo>
17       <mfrac>
18          <mn>2</mn>
19          <mi>x</mi>
20       </mfrac>
21       <mo>=</mo>
22       <mn>0</mn>
23    </math>
```


Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///C:/books/2007/IW3HTP4/exampl
Google

$3\,x^2 + x - \frac{2}{x} = 0$

Fig. 14.17 | Algebraic equation marked up with MathML and displayed in the Firefox browser. (Part 2 of 2.)

Line 9 uses entity reference `&InvisibleTimes;` to indicate a multiplication operation without explicit symbolic representation (i.e., the multiplication symbol does not appear between the 3 and x). For exponentiation, lines 10–13 use the `msup` element, which represents a superscript. This `msup` element has two children—the expression to be superscripted (i.e., the base) and the superscript (i.e., the exponent). Correspondingly, the `msub` element represents a subscript. To display variables such as x, line 11 uses identifier element `mi`.

To display a fraction, lines 17–20 uses the `mfrac` element. Lines 18–19 specify the numerator and the denominator for the fraction. If either the numerator or the denominator contains more than one element, it must appear in an `mrow` element.

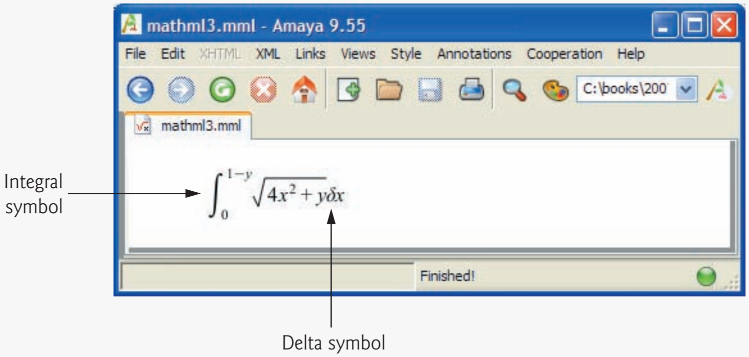
Calculus Expression in MathML

Figure 14.18 marks up a calculus expression that contains an integral symbol and a square-root symbol.

```
1   <?xml version="1.0" encoding="iso-8859-1"?>
2   <!DOCTYPE math PUBLIC "-//W3C//DTD MathML 2.0//EN"
3      "http://www.w3.org/TR/MathML2/dtd/mathml2.dtd">
4
5   <!-- Fig. 14.18 mathml3.html -->
6   <!-- Calculus example using MathML -->
7   <math xmlns="http://www.w3.org/1998/Math/MathML">
8      <mrow>
9         <msubsup>
10           <mo>&int;</mo>
```

Fig. 14.18 | Calculus expression marked up with MathML and displayed in the Amaya browser. [Courtesy of World Wide Web Consortium (W3C).] (Part 1 of 2.)

```
11            <mn>0</mn>
12            <mrow>
13               <mn>1</mn>
14               <mo>&minus;</mo>
15               <mi>y</mi>
16            </mrow>
17         </msubsup>
18         <msqrt>
19            <mn>4</mn>
20            <mo>&InvisibleTimes;</mo>
21            <msup>
22               <mi>x</mi>
23               <mn>2</mn>
24            </msup>
25            <mo>+</mo>
26            <mi>y</mi>
27         </msqrt>
28         <mo>&delta;</mo>
29         <mi>x</mi>
30      </mrow>
31   </math>
```



mathml3.mml - Amaya 9.55
File Edit XHTML XML Links Views Style Annotations Cooperation Help
C:\books\200
mathml3.mml
Finished!

$$\int_0^{1-y}\sqrt{4x^2+y\delta x}$$

Integral symbol

Delta symbol

Fig. 14.18 | Calculus expression marked up with MathML and displayed in the Amaya browser. [Courtesy of World Wide Web Consortium (W3C).] (Part 2 of 2.)

Lines 8–30 group the entire expression in an **`mrow` element**, which is used to group elements that are positioned horizontally in an expression. The entity reference **`&int;`** (line 10) represents the integral symbol, while the **`msubsup` element** (lines 9–17) specifies the subscript and superscript a base expression (e.g., the integral symbol). Element `mo` marks up the integral operator. The `msubsup` element requires three child elements—an operator (e.g., the integral entity, line 10), the subscript expression (line 11) and the superscript expression (lines 12–16). Element `mn` (line 11) marks up the number (i.e., `0`) that represents the subscript. Element `mrow` (lines 12–16) marks up the superscript expression (i.e., `1-`*y*).

Element **`msqrt`** (lines 18–27) represents a square-root expression. Line 28 introduces entity reference `&delta;` for representing a lowercase delta symbol. Delta is an operator, so line 28 places this entity in element `mo`. To see other operations and symbols in MathML, visit `www.w3.org/Math`.

14.7.2 Other Markup Languages

Literally hundreds of markup languages derive from XML. Every day developers find new uses for XML. Figure 14.20 summarizes a few of these markup languages. The website

```
www.service-architecture.com/xml/articles/index.html
```

provides a nice list of common XML vocabularies and descriptions.

Markup language	Description
Chemical Markup Language (CML)	Chemical Markup Language (CML) is an XML vocabulary for representing molecular and chemical information. Many previous methods for storing this type of information (e.g., special file types) inhibited document reuse. CML takes advantage of XML's portability to enable document authors to use and reuse molecular information without corrupting important data in the process.
VoiceXML™	The VoiceXML Forum founded by AT&T, IBM, Lucent and Motorola developed VoiceXML. It provides interactive voice communication between humans and computers through a telephone, PDA (personal digital assistant) or desktop computer. IBM's VoiceXML SDK can process VoiceXML documents. Visit `www.voicexml.org` for more information on VoiceXML.
Synchronous Multimedia Integration Language (SMIL™)	SMIL is an XML vocabulary for multimedia presentations. The W3C was the primary developer of SMIL, with contributions from some companies. Visit `www.w3.org/AudioVideo` for more on SMIL.
Research Information Exchange Markup Language (RIXML)	RIXML, developed by a consortium of brokerage firms, marks up investment data. Visit `www.rixml.org` for more information on RIXML.
Geography Markup Language (GML)	OpenGIS developed the Geography Markup Language to describe geographic information. Visit `www.opengis.org` for more information on GML.
Extensible User Interface Language (XUL)	The Mozilla Project created the Extensible User Interface Language for describing graphical user interfaces in a platform-independent way.

Fig. 14.19 | Various markup languages derived from XML.

14.8 Extensible Stylesheet Language and XSL Transformations

Extensible Stylesheet Language (XSL) documents specify how programs are to render XML document data. XSL is a group of three technologies—**XSL-FO (XSL Formatting Objects)**, **XPath (XML Path Language)** and **XSLT (XSL Transformations)**. XSL-FO is a vocabulary for specifying formatting, and XPath is a string-based language of expressions used by XML and many of its related technologies for effectively and efficiently locating structures and data (such as specific elements and attributes) in XML documents.

The third portion of XSL—XSL Transformations (XSLT)—is a technology for transforming XML documents into other documents—i.e., transforming the structure of the XML document data to another structure. XSLT provides elements that define rules for transforming one XML document to produce a different XML document. This is useful when you want to use data in multiple applications or on multiple platforms, each of which may be designed to work with documents written in a particular vocabulary. For example, XSLT allows you to convert a simple XML document to an XHTML document that presents the XML document's data (or a subset of the data) formatted for display in a web browser.

Transforming an XML document using XSLT involves two tree structures—the **source tree** (i.e., the XML document to be transformed) and the **result tree** (i.e., the XML document to be created). XPath is used to locate parts of the source-tree document that match **templates** defined in an **XSL style sheet**. When a match occurs (i.e., a node matches a template), the matching template executes and adds its result to the result tree. When there are no more matches, XSLT has transformed the source tree into the result tree. The XSLT does not analyze every node of the source tree; it selectively navigates the source tree using XPath's `select` and `match` attributes. For XSLT to function, the source tree must be properly structured. Schemas, DTDs and validating parsers can validate document structure before using XPath and XSLTs.

A Simple XSL Example

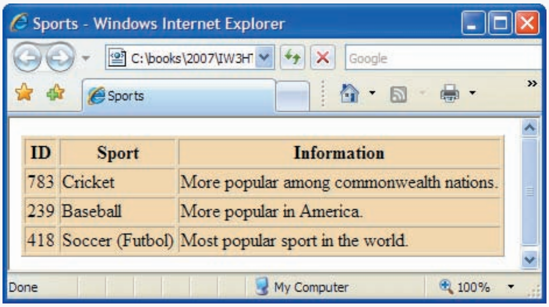
Figure 14.20 lists an XML document that describes various sports. The output shows the result of the transformation (specified in the XSLT template of Fig. 14.21) rendered by Internet Explorer.

To perform transformations, an XSLT processor is required. Popular XSLT processors include Microsoft's MSXML and the Apache Software Foundation's **Xalan 2** (`xml.apache.org`). The XML document in Fig. 14.20 is transformed into an XHTML document by MSXML when the document is loaded in Internet Explorer. MSXML is both an XML parser and an XSLT processor. Firefox also includes an XSLT processor.

```xml
<?xml version = "1.0"?>
<?xml-stylesheet type = "text/xsl" href = "sports.xsl"?>

<!-- Fig. 14.20: sports.xml -->
<!-- Sports Database -->

<sports>
   <game id = "783">
      <name>Cricket</name>

      <paragraph>
         More popular among commonwealth nations.
      </paragraph>
   </game>
```

Fig. 14.20 | XML document that describes various sports. (Part 1 of 2.)

```
15
16       <game id = "239">
17          <name>Baseball</name>
18
19          <paragraph>
20             More popular in America.
21          </paragraph>
22       </game>
23
24       <game id = "418">
25          <name>Soccer (Futbol)</name>
26
27          <paragraph>
28             Most popular sport in the world.
29          </paragraph>
30       </game>
31    </sports>
```



ID	Sport	Information
783	Cricket	More popular among commonwealth nations.
239	Baseball	More popular in America.
418	Soccer (Futbol)	Most popular sport in the world.

Fig. 14.20 | XML document that describes various sports. (Part 2 of 2.)

Line 2 (Fig. 14.20) is a **processing instruction** (**PI**) that references the XSL style sheet `sports.xsl` (Fig. 14.21). A processing instruction is embedded in an XML document and provides application-specific information to whichever XML processor the application uses. In this particular case, the processing instruction specifies the location of an XSLT document with which to transform the XML document. The **`<?`** and **`?>`** (line 2, Fig. 14.20) delimit a processing instruction, which consists of a **PI target** (e.g., `xml-stylesheet`) and a **PI value** (e.g., `type = "text/xsl" href = "sports.xsl"`). The PI value's `type` attribute specifies that `sports.xsl` is a `text/xsl` file (i.e., a text file containing XSL content). The `href` attribute specifies the name and location of the style sheet to apply—in this case, `sports.xsl` in the current directory.

Software Engineering Observation 14.5

XSL enables document authors to separate data presentation (specified in XSL documents) from data description (specified in XML documents).

Common Programming Error 14.10

You will sometimes see the XML processing instruction `<?xml-stylesheet?>` written as `<?xml:stylesheet?>` with a colon rather than a dash. The version with a colon results in an XML parsing error in Firefox.

Figure 14.21 shows the XSL document for transforming the structured data of the XML document of Fig. 14.20 into an XHTML document for presentation. By convention, XSL documents have the filename extension `.xsl`.

Lines 6–7 begin the XSL style sheet with the **`stylesheet`** start tag. Attribute **`version`** specifies the XSLT version to which this document conforms. Line 7 binds namespace prefix **`xsl`** to the W3C's XSLT URI (i.e., `http://www.w3.org/1999/XSL/Transform`).

```
 1  <?xml version = "1.0"?>
 2  <!-- Fig. 14.21: sports.xsl -->
 3  <!-- A simple XSLT transformation -->
 4
 5  <!-- reference XSL style sheet URI -->
 6  <xsl-stylesheet version = "1.0"
 7     xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
 8
 9     <xsl:output method = "html" omit-xml-declaration = "no"
10        doctype-system =
11           "http://www.w3c.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
12        doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>
13
14     <xsl:template match = "/"> <!-- match root element -->
15
16     <html xmlns = "http://www.w3.org/1999/xhtml">
17        <head>
18           <title>Sports</title>
19        </head>
20
21        <body>
22           <table border = "1" bgcolor = "wheat">
23              <thead>
24                 <tr>
25                    <th>ID</th>
26                    <th>Sport</th>
27                    <th>Information</th>
28                 </tr>
29              </thead>
30
31              <!-- insert each name and paragraph element value -->
32              <!-- into a table row. -->
33              <xsl:for-each select = "/sports/game">
34                 <tr>
35                    <td><xsl:value-of select = "@id"/></td>
36                    <td><xsl:value-of select = "name"/></td>
37                    <td><xsl:value-of select = "paragraph"/></td>
38                 </tr>
39              </xsl:for-each>
40           </table>
41        </body>
42     </html>
43
44     </xsl:template>
45  </xsl:stylesheet>
```

Fig. 14.21 | XSLT that creates elements and attributes in an XHTML document.

Lines 9–12 use element **xsl:output** to write an XHTML document type declaration (DOCTYPE) to the result tree (i.e., the XML document to be created). The DOCTYPE identifies XHTML as the type of the resulting document. Attribute method is assigned "html", which indicates that HTML is being output to the result tree. Attribute **omit-xml-declaration** specifies whether the transformation should write the XML declaration to the result tree. In this case, we do not want to omit the XML declaration, so we assign to this attribute the value "no". Attributes doctype-system and doctype-public write the DOCTYPE DTD information to the result tree.

XSLT uses templates (i.e., **xsl:template** elements) to describe how to transform particular nodes from the source tree to the result tree. A template is applied to nodes that are specified in the required match attribute. Line 14 uses the **match** attribute to select the document root (i.e., the conceptual part of the document that contains the root element and everything below it) of the XML source document (i.e., sports.xml). The XPath character **/** (a forward slash) always selects the document root. Recall that XPath is a string-based language used to locate parts of an XML document easily. In XPath, a leading forward slash specifies that we are using absolute addressing (i.e., we are starting from the root and defining paths down the source tree). In the XML document of Fig. 14.20, the child nodes of the document root are the two processing instruction nodes (lines 1–2), the two comment nodes (lines 4–5) and the sports element node (lines 7–31). The template in Fig. 14.21, line 14, matches a node (i.e., the root node), so the contents of the template are now added to the result tree.

The MSXML processor writes the XHTML in lines 16–29 (Fig. 14.21) to the result tree exactly as it appears in the XSL document. Now the result tree consists of the DOCTYPE definition and the XHTML code from lines 16–29. Lines 33–39 use element **xsl:for-each** to iterate through the source XML document, searching for game elements. Attribute **select** is an XPath expression that specifies the nodes (called the node set) on which the xsl:for-each operates. Again, the first forward slash means that we are using absolute addressing. The forward slash between sports and game indicates that game is a child node of sports. Thus, the xsl:for-each finds game nodes that are children of the sports node. The XML document sports.xml contains only one sports node, which is also the document root node. After finding the elements that match the selection criteria, the xsl:for-each processes each element with the code in lines 34–38 (these lines produce one row in a table each time they execute) and places the result of lines 34–38 in the result tree.

Line 35 uses element **value-of** to retrieve attribute id's value and place it in a td element in the result tree. The XPath symbol **@** specifies that id is an attribute node of the context node game. Lines 36–37 place the name and paragraph element values in td elements and insert them in the result tree. When an XPath expression has no beginning forward slash, the expression uses relative addressing. Omitting the beginning forward slash tells the xsl:value-of select statements to search for name and paragraph elements that are children of the context node, not the root node. Due to the last XPath expression selection, the current context node is game, which indeed has an id attribute, a name child element and a paragraph child element.

Using XSLT to Sort and Format Data

Figure 14.22 presents an XML document (sorting.xml) that marks up information about a book. Note that several elements of the markup describing the book appear out of

```
 1  <?xml version = "1.0"?>
 2  <?xml-stylesheet type = "text/xsl" href = "sorting.xsl"?>
 3
 4  <!-- Fig. 14.22: sorting.xml -->
 5  <!-- XML document containing book information -->
 6  <book isbn = "999-99999-9-X">
 7     <title>Deitel&apos;s XML Primer</title>
 8
 9     <author>
10        <firstName>Jane</firstName>
11        <lastName>Blue</lastName>
12     </author>
13
14     <chapters>
15        <frontMatter>
16           <preface pages = "2" />
17           <contents pages = "5" />
18           <illustrations pages = "4" />
19        </frontMatter>
20
21        <chapter number = "3" pages = "44">Advanced XML</chapter>
22        <chapter number = "2" pages = "35">Intermediate XML</chapter>
23        <appendix number = "B" pages = "26">Parsers and Tools</appendix>
24        <appendix number = "A" pages = "7">Entities</appendix>
25        <chapter number = "1" pages = "28">XML Fundamentals</chapter>
26     </chapters>
27
28     <media type = "CD" />
29  </book>
```

Fig. 14.22 | XML document containing book information.

order (e.g., the element describing Chapter 3 appears before the element describing Chapter 2). We arranged them this way purposely to demonstrate that the XSL style sheet referenced in line 2 (`sorting.xsl`) can sort the XML file's data for presentation purposes.

Figure 14.23 presents an XSL document (`sorting.xsl`) for transforming `sorting.xml` (Fig. 14.22) to XHTML. Recall that an XSL document navigates a source tree and builds a result tree. In this example, the source tree is XML, and the output tree is XHTML. Line 14 of Fig. 14.23 matches the root element of the document in Fig. 14.22. Line 15 outputs an `html` start tag to the result tree. In line 16, the `<xsl:apply-templates/>` element specifies that the XSLT processor is to apply the `xsl:templates` defined in this XSL document to the current node's (i.e., the document root's) children. The content from the applied templates is output in the `html` element that ends at line 17.

Lines 21–84 specify a template that matches element `book`. The template indicates how to format the information contained in `book` elements of `sorting.xml` (Fig. 14.22) as XHTML.

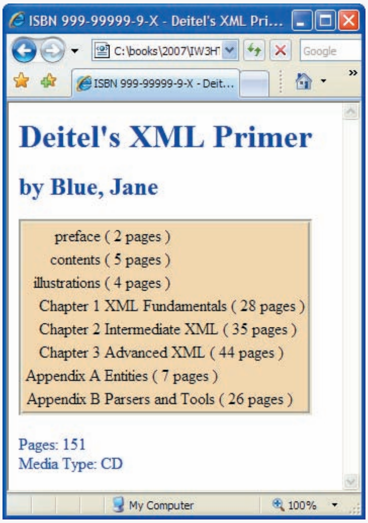
Lines 23–24 create the title for the XHTML document. We use the `book`'s ISBN (from attribute `isbn`) and the contents of element `title` to create the string that appears in the browser window's title bar (**ISBN 999-99999-9-X - Deitel's XML Primer**).

Line 28 creates a header element that contains the book's title. Lines 29–31 create a header element that contains the book's author. Because the context node (i.e., the current

```
<?xml version = "1.0"?>

<!-- Fig. 14.23: sorting.xsl -->
<!-- Transformation of book information into XHTML -->
<xsl:stylesheet version = "1.0"
   xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">

   <!-- write XML declaration and DOCTYPE DTD information -->
   <xsl:output method = "html" omit-xml-declaration = "no"
      doctype-system = "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd"
      doctype-public = "-//W3C//DTD XHTML 1.1//EN"/>

   <!-- match document root -->
   <xsl:template match = "/">
      <html xmlns = "http://www.w3.org/1999/xhtml">
         <xsl:apply-templates/>
      </html>
   </xsl:template>

   <!-- match book -->
   <xsl:template match = "book">
      <head>
         <title>ISBN <xsl:value-of select = "@isbn"/> -
            <xsl:value-of select = "title"/></title>
      </head>

      <body>
         <h1 style = "color: blue"><xsl:value-of select = "title"/></h1>
         <h2 style = "color: blue">by
            <xsl:value-of select = "author/lastName"/>,
            <xsl:value-of select = "author/firstName"/></h2>

         <table style = "border-style: groove; background-color: wheat">

            <xsl:for-each select = "chapters/frontMatter/*">
               <tr>
                  <td style = "text-align: right">
                     <xsl:value-of select = "name()"/>
                  </td>

                  <td>
                     ( <xsl:value-of select = "@pages"/> pages )
                  </td>
               </tr>
            </xsl:for-each>

            <xsl:for-each select = "chapters/chapter">
               <xsl:sort select = "@number" data-type = "number"
                    order = "ascending"/>
               <tr>
                  <td style = "text-align: right">
                     Chapter <xsl:value-of select = "@number"/>
                  </td>
```

Fig. 14.23 | XSL document that transforms `sorting.xml` into XHTML. (Part 1 of 2.)

```

                  <td>
                     <xsl:value-of select = "text()"/>
                     ( <xsl:value-of select = "@pages"/> pages )
                  </td>
               </tr>
            </xsl:for-each>

            <xsl:for-each select = "chapters/appendix">
               <xsl:sort select = "@number" data-type = "text"
                  order = "ascending"/>
               <tr>
                  <td style = "text-align: right">
                     Appendix <xsl:value-of select = "@number"/>
                  </td>

                  <td>
                     <xsl:value-of select = "text()"/>
                     ( <xsl:value-of select = "@pages"/> pages )
                  </td>
               </tr>
            </xsl:for-each>
         </table>

         <br /><p style = "color: blue">Pages:
            <xsl:variable name = "pagecount"
               select = "sum(chapters//*/@pages)"/>
            <xsl:value-of select = "$pagecount"/>
         <br />Media Type: <xsl:value-of select = "media/@type"/></p>
      </body>
   </xsl:template>
</xsl:stylesheet>
```


ISBN 999-99999-9-X - Deitel's XML Pri...
C:\books\2007\IW3HT
Google
ISBN 999-99999-9-X - Deit...
My Computer
100%

Deitel's XML Primer

by Blue, Jane

preface (2 pages)
contents (5 pages)
illustrations (4 pages)
Chapter 1 XML Fundamentals (28 pages)
Chapter 2 Intermediate XML (35 pages)
Chapter 3 Advanced XML (44 pages)
Appendix A Entities (7 pages)
Appendix B Parsers and Tools (26 pages)

Pages: 151
Media Type: CD

Fig. 14.23 | XSL document that transforms `sorting.xml` into XHTML. (Part 2 of 2.)

node being processed) is book, the XPath expression author/lastName selects the author's last name, and the expression author/firstName selects the author's first name.

Line 35 selects each element (indicated by an asterisk) that is a child of element frontMatter. Line 38 calls **node-set function name** to retrieve the current node's element name (e.g., preface). The current node is the context node specified in the xsl:for-each (line 35). Line 42 retrieves the value of the pages attribute of the current node.

Line 47 selects each chapter element. Lines 48–49 use element **xsl:sort** to sort chapters by number in ascending order. Attribute **select** selects the value of attribute number in context node chapter. Attribute **data-type**, with value "number", specifies a numeric sort, and attribute **order**, with value "ascending", specifies ascending order. Attribute data-type also accepts the value "text" (line 63), and attribute order also accepts the value "descending". Line 56 uses **node-set function text** to obtain the text between the chapter start and end tags (i.e., the name of the chapter). Line 57 retrieves the value of the pages attribute of the current node. Lines 62–75 perform similar tasks for each appendix.

Lines 79–80 use an **XSL variable** to store the value of the book's total page count and output the page count to the result tree. Attribute **name** specifies the variable's name (i.e., pagecount), and attribute select assigns a value to the variable. Function **sum** (line 80) totals the values for all page attribute values. The two slashes between chapters and * indicate a **recursive descent**—the MSXML processor will search for elements that contain an attribute named pages in all descendant nodes of chapters. The XPath expression

```
//*
```

selects all the nodes in an XML document. Line 81 retrieves the value of the newly created XSL variable pagecount by placing a dollar sign in front of its name.

Summary of XSL Style-Sheet Elements

This section's examples used several predefined XSL elements to perform various operations. Figure 14.24 lists these elements and several other commonly used XSL elements. For more information on these elements and XSL in general, see www.w3.org/Style/XSL.

Element	Description
`<xsl:apply-templates>`	Applies the templates of the XSL document to the children of the current node.
`<xsl:apply-templates match = "`*expression*`">`	Applies the templates of the XSL document to the children of *expression*. The value of the attribute match (i.e., *expression*) must be an XPath expression that specifies elements.
`<xsl:template>`	Contains rules to apply when a specified node is matched.
`<xsl:value-of select = "`*expression*`">`	Selects the value of an XML element and adds it to the output tree of the transformation. The required select attribute contains an XPath expression.

Fig. 14.24 | XSL style-sheet elements. (Part 1 of 2.)

Element	Description
`<xsl:for-each select = "`*expression*`">`	Applies a template to every node selected by the XPath specified by the `select` attribute.
`<xsl:sort select = "`*expression*`">`	Used as a child element of an `<xsl:apply-templates>` or `<xsl:for-each>` element. Sorts the nodes selected by the `<xsl:apply-template>` or `<xsl:for-each>` element so that the nodes are processed in sorted order.
`<xsl:output>`	Has various attributes to define the format (e.g., XML, XHTML), version (e.g., 1.0, 2.0), document type and media type of the output document. This tag is a top-level element—it can be used only as a child element of an `xml:stylesheet`.
`<xsl:copy>`	Adds the current node to the output tree.

Fig. 14.24 | XSL style-sheet elements. (Part 2 of 2.)

This section introduced Extensible Stylesheet Language (XSL) and showed how to create XSL transformations to convert XML documents from one format to another. We showed how to transform XML documents to XHTML documents for display in a web browser. Recall that these transformations are performed by MSXML, Internet Explorer's built-in XML parser and XSLT processor. In most business applications, XML documents are transferred between business partners and are transformed to other XML vocabularies programmatically. Section 14.9 discusses the XML Document Object Model (DOM) and demonstrates how to manupulate the DOM of an XML document using JavaScript.

14.9 Document Object Model (DOM)

Although an XML document is a text file, retrieving data from the document using traditional sequential file processing techniques is neither practical nor efficient, especially for adding and removing elements dynamically.

Upon successfully parsing a document, some XML parsers store document data as tree structures in memory. Figure 14.25 illustrates the tree structure for the root element of the document `article.xml` (Fig. 14.2). This hierarchical tree structure is called a **Document Object Model** (**DOM**) **tree**, and an XML parser that creates this type of structure is known as a **DOM parser**. Each element name (e.g., `article`, `date`, `firstName`) is represented by a node. A node that contains other nodes (called **child nodes** or children) is called a **parent node** (e.g., `author`). A parent node can have many children, but a child node can have only one parent node. Nodes that are peers (e.g., `firstName` and `lastName`) are called **sibling nodes**. A node's **descendant nodes** include its children, its children's children and so on. A node's **ancestor nodes** include its parent, its parent's parent and so on. Many of the XML DOM capabilities you'll see in this section are similar or identical to those of the XHTML DOM you learned in Chapter 12.

The DOM tree has a single **root node**, which contains all the other nodes in the document. For example, the root node of the DOM tree that represents `article.xml` contains a node for the XML declaration (line 1), two nodes for the comments (lines 3–4) and a node for the XML document's root element `article` (line 5).

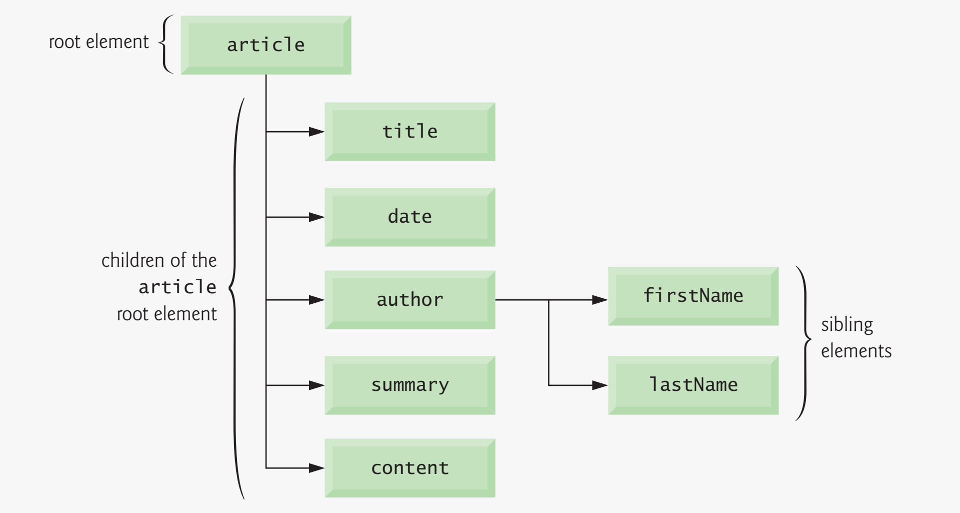


Fig. 14.25 | Tree structure for the document `article.xml` of Fig. 14.2.

To introduce document manipulation with the XML Document Object Model, we provide a scripting example (Fig. 14.26) that uses JavaScript and XML. This example loads the XML document `article.xml` (Fig. 14.2) and uses the XML DOM API to display the document's element names and values. The example also provides buttons that enable you to navigate the DOM structure. As you click each button, an appropriate part of the document is highlighted. All of this is done in a manner that enables the example to execute in both Internet Explorer 7 and Firefox 2. Figure 14.26 lists the JavaScript code that manipulates this XML document and displays its content in an XHTML page.

Overview of the **body** *Element*

Lines 203–217 create the XHTML document's `body`. When the body loads, its `onload` event calls our JavaScript function `loadXMLDocument` to load and display the contents of `article.xml` in the `div` at line 216 (`outputDiv`). Lines 204–215 define a form consisting of five buttons. When each button is pressed, it invokes one of our JavaScript functions to navigate `article.xml`'s DOM structure.

Global Script Variables

Lines 16–21 in the `script` element (lines 14–201) declare several variables used throughout the script. Variable `doc` references a DOM object representation of `article.xml`. Variable `outputHTML` stores the markup that will be placed in `outputDiv`. Variable `idCounter` is used to track the unique `id` attributes that we assign to each element in the `outputHTML` markup. These `id`s will be used to dynamically highlight parts of the document when the user clicks the buttons in the form. Variable `depth` determines the indentation level for the content in `article.xml`. We use this to structure the output using the nesting of the elements in `article.xml`. Variables `current` and `previous` track the current and previous nodes in `article.xml`'s DOM structure as the user navigates it.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 14.26: XMLDOMTraversal.html -->
<!-- Traversing an XML document using the XML DOM. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
   <title>Traversing an XML document using the XML DOM</title>
   <style type = "text/css">
      .highlighted { background-color: yellow }
      #outputDiv { font: 10pt "Lucida Console", monospace; }
   </style>
   <script type="text/javascript">
   <!--
   var doc; // variable to reference the XML document
   var outputHTML = ""; // stores text to output in outputDiv
   var idCounter = 1; // used to create div IDs
   var depth = -1; // tree depth is -1 to start
   var current = null; // represents the current node for traversals
   var previous = null; // represent prior node in traversals

   // load XML document based on whether the browser is IE7 or Firefox 2
   function loadXMLDocument( url )
   {
      if ( window.ActiveXObject ) // IE7
      {
         // create IE7-specific XML document object
         doc = new ActiveXObject( "Msxml2.DOMDocument.6.0" );
         doc.async = false; // specifies synchronous loading of XML doc
         doc.load( url ); // load the XML document specified by url
         buildHTML( doc.childNodes ); // display the nodes
         displayDoc();
      } // end if
      else if ( document.implementation &&
         document.implementation.createDocument ) // other browsers
      {
         // create XML document object
         doc = document.implementation.createDocument( "", "", null );
         doc.load( url ); // load the XML document specified by url
         doc.onload = function() // function to execute when doc loads
         {
            buildHTML( doc.childNodes ); // called by XML doc onload event
            displayDoc(); // display the HTML
         } // end XML document's onload event handler
      } // end else
      else // not supported
         alert( 'This script is not supported by your browser' );
   } // end function loadXMLDocument

   // traverse xmlDocument and build XHTML representation of its content
   function buildHTML( childList )
   {
```

Fig. 14.26 | Traversing an XML document using the XML DOM. (Part 1 of 8.)

```
      ++depth; // increase tab depth

      // display each node's content
      for ( var i = 0; i < childList.length; i++ )
      {
         switch ( childList[ i ].nodeType )
         {
            case 1: // Node.ELEMENT_NODE; value used for portability
               outputHTML += "<div id=\"id" + idCounter + "\">";
               spaceOutput( depth ); // insert spaces
               outputHTML += childList[ i ].nodeName; // show node's name
               ++idCounter; // increment the id counter

               // if current node has children, call buildHTML recursively
               if ( childList[ i ].childNodes.length != 0 )
                  buildHTML( childList[ i ].childNodes );

               outputHTML += "</div>";
               break;
            case 3: // Node.TEXT_NODE; value used for portability
            case 8: // Node.COMMENT_NODE; value used for portability
               // if nodeValue is not 3 or 6 spaces (Firefox issue),
               // include nodeValue in HTML
               if ( childList[ i ].nodeValue.indexOf( "   " ) == -1 &&
                  childList[ i ].nodeValue.indexOf( "      " ) == -1 )
               {
                  outputHTML += "<div id=\"id" + idCounter + "\">";
                  spaceOutput( depth ); // insert spaces
                  outputHTML += childList[ i ].nodeValue + "</div>";
                  ++idCounter; // increment the id counter
               } // end if
         } // end switch
      } // end for

      --depth; // decrease tab depth
   } // end function buildHTML

   // display the XML document and highlight the first child
   function displayDoc()
   {
      document.getElementById( "outputDiv" ).innerHTML = outputHTML;
      current = document.getElementById( 'id1' );
      setCurrentNodeStyle( current.id, true );
   } // end function displayDoc

   // insert non-breaking spaces for indentation
   function spaceOutput( number )
   {
      for ( var i = 0; i < number; i++ )
      {
         outputHTML += "&nbsp;&nbsp;&nbsp;";
      } // end for
   } // end function spaceOutput
```

Fig. 14.26 | Traversing an XML document using the XML DOM. (Part 2 of 8.)

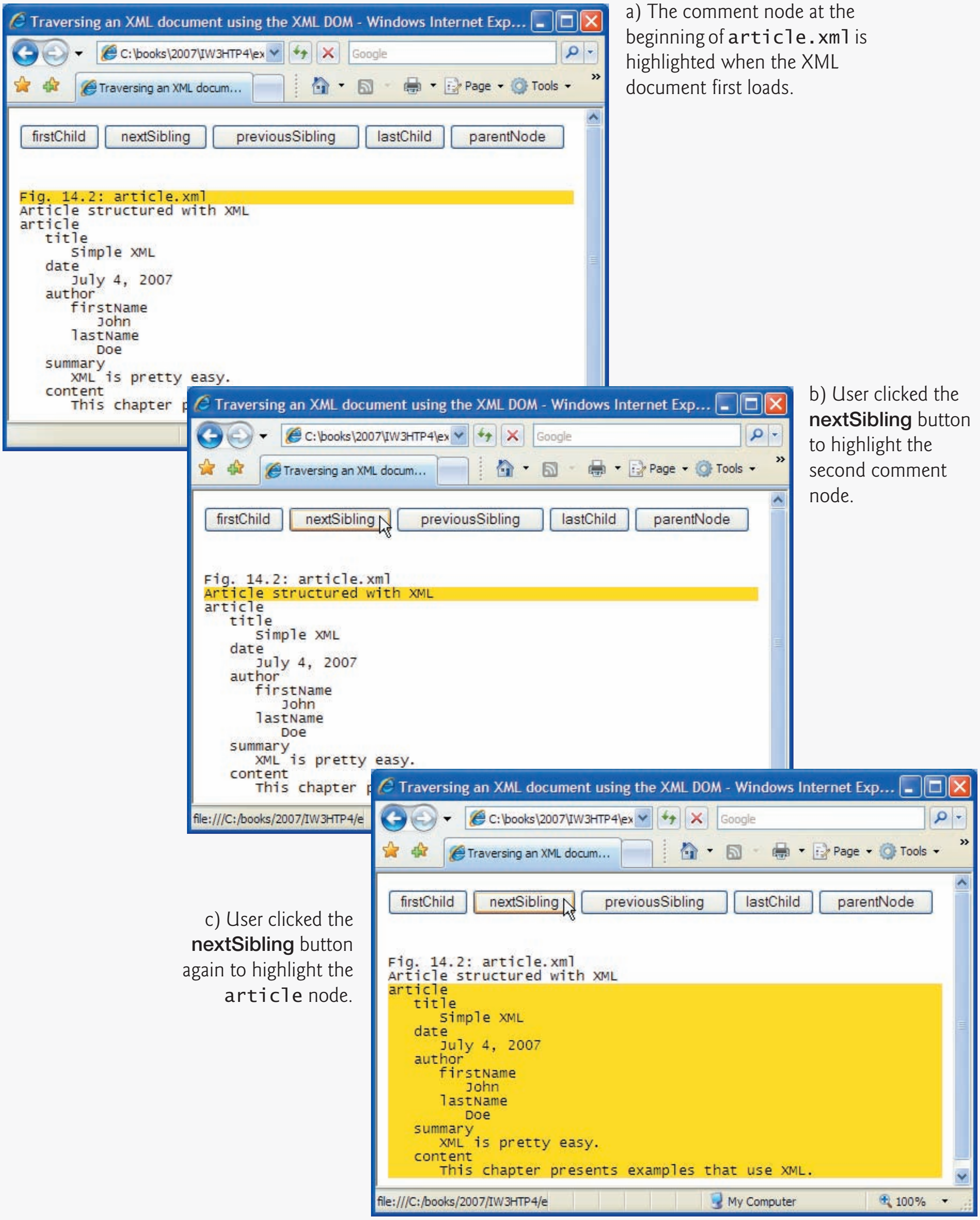
```

   // highlight first child of current node
   function processFirstChild()
   {
      if ( current.childNodes.length == 1 && // only one child
         current.firstChild.nodeType == 3 )  // and it's a text node
      {
         alert( "There is no child node" );
      } // end if
      else if ( current.childNodes.length > 1 )
      {
         previous = current; // save currently highlighted node

         if ( current.firstChild.nodeType != 3 ) // if not text node
            current = current.firstChild; // get new current node
         else // if text node, use firstChild's nextSibling instead
            current = current.firstChild.nextSibling; // get first sibling

         setCurrentNodeStyle( previous.id, false ); // remove highlight
         setCurrentNodeStyle( current.id, true ); // add highlight
      } // end if
      else
         alert( "There is no child node" );
   } // end function processFirstChild

   // highlight next sibling of current node
   function processNextSibling()
   {
      if ( current.id != "outputDiv" && current.nextSibling )
      {
         previous = current; // save currently highlighted node
         current = current.nextSibling; // get new current node
         setCurrentNodeStyle( previous.id, false ); // remove highlight
         setCurrentNodeStyle( current.id, true ); // add highlight
      } // end if
      else
         alert( "There is no next sibling" );
   } // end function processNextSibling

   // highlight previous sibling of current node if it is not a text node
   function processPreviousSibling()
   {
      if ( current.id != "outputDiv" && current.previousSibling &&
         current.previousSibling.nodeType != 3 )
      {
         previous = current; // save currently highlighted node
         current = current.previousSibling; // get new current node
         setCurrentNodeStyle( previous.id, false ); // remove highlight
         setCurrentNodeStyle( current.id, true ); // add highlight
      } // end if
      else
         alert( "There is no previous sibling" );
   } // end function processPreviousSibling
```

Fig. 14.26 | Traversing an XML document using the XML DOM. (Part 3 of 8.)

```

   // highlight last child of current node
   function processLastChild()
   {
      if ( current.childNodes.length == 1 &&
         current.lastChild.nodeType == 3 )
      {
         alert( "There is no child node" );
      } // end if
      else if ( current.childNodes.length != 0 )
      {
         previous = current; // save currently highlighted node
         current = current.lastChild; // get new current node
         setCurrentNodeStyle( previous.id, false ); // remove highlight
         setCurrentNodeStyle( current.id, true ); // add highlight
      } // end if
      else
         alert( "There is no child node" );
   } // end function processLastChild

   // highlight parent of current node
   function processParentNode()
   {
      if ( current.parentNode.id != "body" )
      {
         previous = current; // save currently highlighted node
         current = current.parentNode; // get new current node
         setCurrentNodeStyle( previous.id, false ); // remove highlight
         setCurrentNodeStyle( current.id, true ); // add highlight
      } // end if
      else
         alert( "There is no parent node" );
   } // end function processParentNode

   // set style of node with specified id
   function setCurrentNodeStyle( id, highlight )
   {
      document.getElementById( id ).className =
         ( highlight ? "highlighted" : "" );
   } // end function setCurrentNodeStyle
   // -->
   </script>
</head>
<body id = "body" onload = "loadXMLDocument( 'article.xml' );">
   <form action = "" onsubmit = "return false;">
      <input type = "submit" value = "firstChild"
         onclick = "processFirstChild()"/>
      <input type = "submit" value = "nextSibling"
         onclick = "processNextSibling()"/>
      <input type = "submit" value = "previousSibling"
         onclick = "processPreviousSibling()"/>
      <input type = "submit" value = "lastChild"
         onclick = "processLastChild()"/>
```

Fig. 14.26 | Traversing an XML document using the XML DOM. (Part 4 of 8.)

```
      <input type = "submit" value = "parentNode"
         onclick = "processParentNode()"/>
   </form><br/>
   <div id = "outputDiv"></div>
</body>
</html>
```


Traversing an XML document using the XML DOM - Windows Internet Exp...
C:\books\2007\IW3HTP4\ex
Google
Traversing an XML docum...
Page
Tools
firstChild
nextSibling
previousSibling
lastChild
parentNode
file:///C:/books/2007/IW3HTP4/e
My Computer
100%

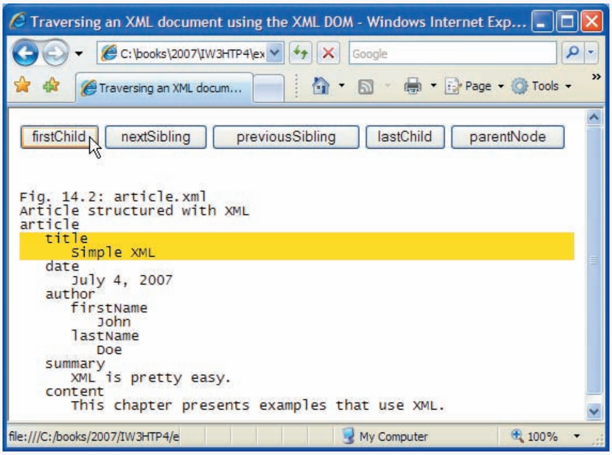
```
Fig. 14.2: article.xml
Article structured with XML
article
   title
      Simple XML
   date
      July 4, 2007
   author
      firstName
         John
      lastName
         Doe
   summary
      XML is pretty easy.
   content
      This chapter presents examples that use XML.
```

a) The comment node at the beginning of `article.xml` is highlighted when the XML document first loads.

b) User clicked the **nextSibling** button to highlight the second comment node.

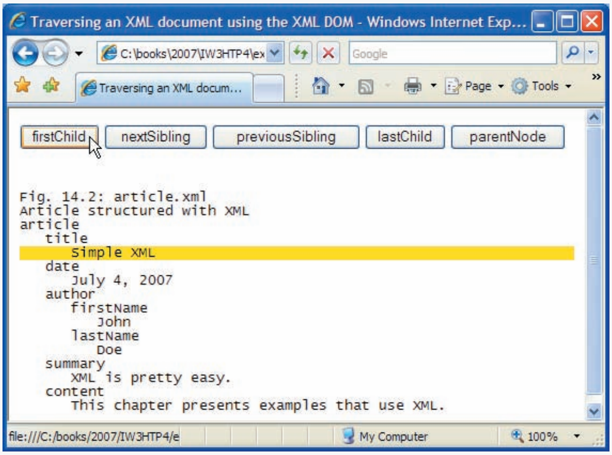
c) User clicked the **nextSibling** button again to highlight the `article` node.

Fig. 14.26 | Traversing an XML document using the XML DOM. (Part 5 of 8.)


Traversing an XML document using the XML DOM - Windows Internet Exp...
C:\books\2007\IW3HTP4\ex
Google
Traversing an XML docum...
Page
Tools
firstChild
nextSibling
previousSibling
lastChild
parentNode
file:///C:/books/2007/IW3HTP4/e
My Computer
100%

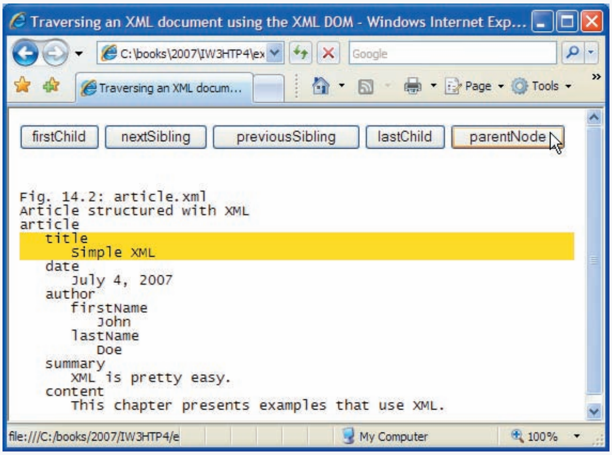
```
Fig. 14.2: article.xml
Article structured with XML
article
   title
      Simple XML
   date
      July 4, 2007
   author
      firstName
         John
      lastName
         Doe
   summary
      XML is pretty easy.
   content
      This chapter presents examples that use XML.
```

d) User clicked the **firstChild** button to highlight the `article` node's `title` child node.


Traversing an XML document using the XML DOM - Windows Internet Exp...
C:\books\2007\IW3HTP4\ex
Google
Traversing an XML docum...
Page
Tools
firstChild
nextSibling
previousSibling
lastChild
parentNode
file:///C:/books/2007/IW3HTP4/e
My Computer
100%

```
Fig. 14.2: article.xml
Article structured with XML
article
   title
      Simple XML
   date
      July 4, 2007
   author
      firstName
         John
      lastName
         Doe
   summary
      XML is pretty easy.
   content
      This chapter presents examples that use XML.
```

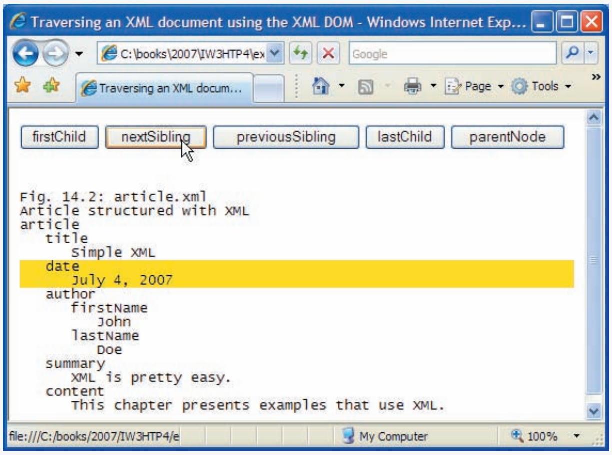
e) User clicked the **firstChild** button again to highlight the `title` node's text child node.


Traversing an XML document using the XML DOM - Windows Internet Exp...
C:\books\2007\IW3HTP4\ex
Google
Traversing an XML docum...
Page
Tools
firstChild
nextSibling
previousSibling
lastChild
parentNode
file:///C:/books/2007/IW3HTP4/e
My Computer
100%

```
Fig. 14.2: article.xml
Article structured with XML
article
   title
      Simple XML
   date
      July 4, 2007
   author
      firstName
         John
      lastName
         Doe
   summary
      XML is pretty easy.
   content
      This chapter presents examples that use XML.
```

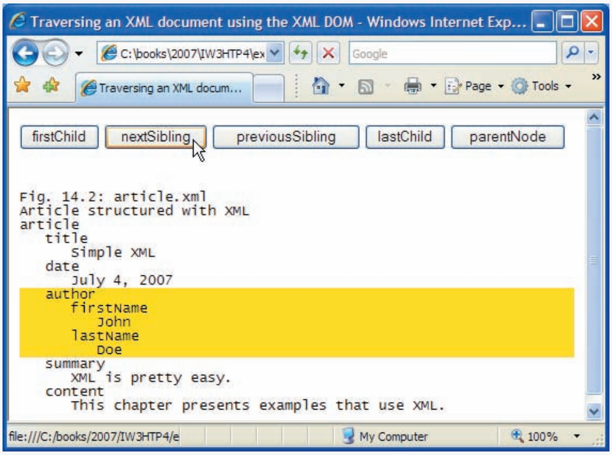
f) User clicked the **parentNode** button to highlight the text node's parent `title` node.

Fig. 14.26 | Traversing an XML document using the XML DOM. (Part 6 of 8.)


Traversing an XML document using the XML DOM - Windows Internet Exp...
C:\books\2007\IW3HTP4\ex
Google
Traversing an XML docum...
Page
Tools
firstChild
nextSibling
previousSibling
lastChild
parentNode
file:///C:/books/2007/IW3HTP4/e
My Computer
100%

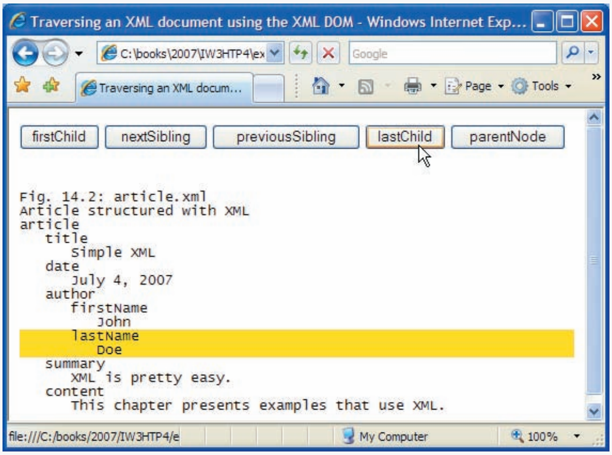
```
Fig. 14.2: article.xml
Article structured with XML
article
   title
      Simple XML
   date
      July 4, 2007
   author
      firstName
         John
      lastName
         Doe
   summary
      XML is pretty easy.
   content
      This chapter presents examples that use XML.
```

g) User clicked the **nextSibling** button to highlight the `title` node's `date` sibling node.


Traversing an XML document using the XML DOM - Windows Internet Exp...
C:\books\2007\IW3HTP4\ex
Google
Traversing an XML docum...
Page
Tools
firstChild
nextSibling
previousSibling
lastChild
parentNode
file:///C:/books/2007/IW3HTP4/e
My Computer
100%

```
Fig. 14.2: article.xml
Article structured with XML
article
   title
      Simple XML
   date
      July 4, 2007
   author
      firstName
         John
      lastName
         Doe
   summary
      XML is pretty easy.
   content
      This chapter presents examples that use XML.
```

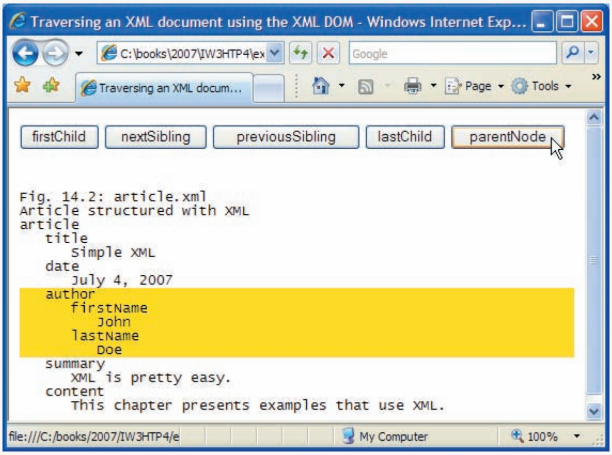
h) User clicked the **nextSibling** button to highlight the `date` node's `author` sibling node.


Traversing an XML document using the XML DOM - Windows Internet Exp...
C:\books\2007\IW3HTP4\ex
Google
Traversing an XML docum...
Page
Tools
firstChild
nextSibling
previousSibling
lastChild
parentNode
file:///C:/books/2007/IW3HTP4/e
My Computer
100%

```
Fig. 14.2: article.xml
Article structured with XML
article
   title
      Simple XML
   date
      July 4, 2007
   author
      firstName
         John
      lastName
         Doe
   summary
      XML is pretty easy.
   content
      This chapter presents examples that use XML.
```

i) User clicked the **lastChild** button to highlight the `author` node's last child node (`lastName`).

Fig. 14.26 | Traversing an XML document using the XML DOM. (Part 7 of 8.)


Traversing an XML document using the XML DOM - Windows Internet Exp...
C:\books\2007\IW3HTP4\ex
Google
Traversing an XML docum...
Page
Tools
firstChild
nextSibling
previousSibling
lastChild
parentNode
file:///C:/books/2007/IW3HTP4/e
My Computer
100%

```
Fig. 14.2: article.xml
Article structured with XML
article
   title
      Simple XML
   date
      July 4, 2007
   author
      firstName
         John
      lastName
         Doe
   summary
      XML is pretty easy.
   content
      This chapter presents examples that use XML.
```

j) User clicked the **parentNode** button to highlight the lastName node's author parent node.

Fig. 14.26 | Traversing an XML document using the XML DOM. (Part 8 of 8.)

Function loadXMLDocument

Function loadXMLDocument (lines 24–49) receives the URL of an XML document to load, then loads the document based on whether the browser is Internet Explorer 7 (26–34) or Firefox 2 (lines 35–46)—the code for Firefox 2 works in several other browsers as well. Line 26 determines whether window.**ActiveXObject** exists. If so, this indicates that the browser is Internet Explorer. Line 29 creates a Microsoft ActiveXObject that loads Microsoft's **MSXML parser**, which provides capabilities for manipulating XML documents. Line 30 indicates that we'd like the XML document to be loaded synchronously, then line 31 uses the ActiveXObject's **load method** to load article.xml. When this completes, we call our buildHTML method (defined in lines 52–89) to construct an XHTML representation of the XML document. The expression doc.childNodes is a list of the XML document's top-level nodes. Line 33 calls our displayDoc function (lines 92–97) to display the contents of article.xml in outputDiv.

If the browser is Firefox 2, then the document object's **implementation** property and the implementation property's **createDocument** method will exist (lines 35–36). In this case, line 39 uses the createDocument method to create an empty XML document object. If necessary, you can specify the XML document's namespace as the first argument and its root element as the second argument. We used empty strings for both in this example. According to the site www.w3schools.com/xml/xml_parser.asp, the third argument is not implemented yet, so it should always be null. Line 40 calls its load method to load article.xml. Firefox loads the XML document asynchronously, so you must use the XML document's onload property to specify a function to call (an anonymous function in this example) when the document finishes loading. When this event occurs, lines 43–44 call buildHTML and displayDoc just as we did in lines 32–33.

Common Programming Error 14.11

Attempting to process the contents of a dynamically loaded XML document in Firefox before the document's onload event fires is a logic error. The document's contents are not available until the onload event fires.

Function buildHTML

Function buildHTML (lines 52–89) is a recursive function that receives a list of nodes as an argument. Line 54 increments the depth for indentation purposes. Lines 57–86 iterate through the nodes in the list. The switch statement (lines 59–85) uses the current node's **nodeType property** to determine whether the current node is an element (line 61), a text node (i.e., the text content of an element; line 73) or a comment node (line 74). If it is an element, then we begin a new div element in our XHTML (line 62) and give it a unique id. Then function spaceOutput (defined in lines 100–106) appends **nonbreaking spaces (&nbsp;)**—i.e., spaces that the browser is not allowed to collapse or that can be used to keep words together—to indent the current element to the correct level. Line 64 appends the name of the current element using the node's **nodeName property**. If the current element has children, the length of the current node's childNodes list is nonzero and line 69 recursively calls buildHTML to append the current element's child nodes to the markup. When that recursive call completes, line 71 completes the div element that we started at line 62.

If the current element is a text node, lines 77–78 obtain the node's value with the **nodeValue property** and use the string method indexOf to determine whether the node's value starts with three or six spaces. Unfortunately, unlike MSMXL, Firefox's XML parser does not ignore the white space used for indentation in XML documents. Instead it creates text nodes containing just the space characters. The condition in lines 77–78 enables us to ignore these nodes in Firefox. If the node contains text, lines 80–82 append a new div to the markup and use the node's nodeValue property to insert that text in the div. Line 88 in buildHTML decrements the depth counter.

Portability Tip 14.4

Firefox's XML parser does not ignore white space used for indentation in XML documents. Instead, it creates text nodes containing the white-space characters.

Function displayDoc

In function displayDoc (lines 92–97), line 94 uses the DOM's getElementById method to obtain the outputDiv element and set its innerHTML property to the new markup generated by buildHTML. Then, line 95 sets variable current to refer to the div with id 'id1' in the new markup, and line 96 uses our setCurrentNodeStyle method (defined at lines 195–199) to highlight that div.

Functions processFirstChild *and* processLastChild

Function processFirstChild (lines 109–130) is invoked by the onclick event of the button at lines 205–206. If the current node has only one child and it's a text node (lines 111–112), line 114 displays an alert dialog indicating that there is no child node—we navigate only to nested XML elements in this example. If there are two or more children, line 118 stores the value of current in previous, and lines 120–123 set current to refer to its **firstChild** (if this child is not a text node) or its firstChild's **nextSibling** (if the firstChild is a text node)—again, this is to ensure that we navigate only to nodes that represent XML elements. Then lines 125–126 unhighlight the previous node and highlight the new current node. Function processLastChild (lines 162–178) works similarly, using the current node's **lastChild** property.

Functions **processNextSibling** *and* **processPreviousSibling**

Function processNextSibling (lines 133–144) first ensures that the current node is not the outputDiv and that nextSibling exists. If so, lines 137–140 adjust the previous and current nodes accordingly and update their highlighting. Function processPreviousSibling (lines 147–159) works similarly, ensuring first that the current node is not the outputDiv, that previousSibling exists and that previousSibling is not a text node.

Function **processParentNode**

Function processParentNode (lines 181–192) first checks whether the current node's **parentNode** is the XHTML page's body. If not, lines 185–188 adjust the previous and current nodes accordingly and update their highlighting.

Common DOM Properties

The tables in Figs. 14.27–14.32 describe many common DOM properties and methods. Some of the key DOM objects are **Node** (a node in the tree), **NodeList** (an ordered set of Nodes), **Document** (the document), **Element** (an element node), **Attr** (an attribute node) and **Text** (a text node). There are many more objects, properties and methods than we can possibly list here. Our XML Resource Center (www.deitel.com/XML/) includes links to various DOM reference websites.

Property/Method	Description
nodeType	An integer representing the node type.
nodeName	The name of the node.
nodeValue	A string or null depending on the node type.
parentNode	The parent node.
childNodes	A NodeList (Fig. 14.28) with all the children of the node.
firstChild	The first child in the Node's NodeList.
lastChild	The last child in the Node's NodeList.
previousSibling	The node preceding this node; null if there is no such node.
nextSibling	The node following this node; null if there is no such node.
attributes	A collection of Attr objects (Fig. 14.31) containing the attributes for this node.
insertBefore	Inserts the node (passed as the first argument) before the existing node (passed as the second argument). If the new node is already in the tree, it is removed before insertion. The same behavior is true for other methods that add nodes.
replaceChild	Replaces the second argument node with the first argument node.
removeChild	Removes the child node passed to it.
appendChild	Appends the node it receives to the list of child nodes.

Fig. 14.27 | Common Node properties and methods.

Property/Method	Description
`item`	Method that receives an index number and returns the element node at that index. Indices range from 0 to *length* – 1. You can also access the nodes in a `NodeList` via array indexing.
`length`	The total number of nodes in the list.

Fig. 14.28 | `NodeList` property and method.

Property/Method	Description
`documentElement`	The root node of the document.
`createElement`	Creates and returns an element node with the specified tag name.
`createAttribute`	Creates and returns an `Attr` node (Fig. 14.31) with the specified name and value.
`createTextNode`	Creates and returns a text node that contains the specified text.
`getElementsByTagName`	Returns a `NodeList` of all the nodes in the subtree with the name specified as the first argument, ordered as they would be encountered in a preorder traversal. An optional second argument specifies either the direct child nodes (0) or any descendant (1).

Fig. 14.29 | `Document` properties and methods.

Property/Method	Description
`tagName`	The name of the element.
`getAttribute`	Returns the value of the specified attribute.
`setAttribute`	Changes the value of the attribute passed as the first argument to the value passed as the second argument.
`removeAttribute`	Removes the specified attribute.
`getAttributeNode`	Returns the specified attribute node.
`setAttributeNode`	Adds a new attribute node with the specified name.

Fig. 14.30 | `Element` property and methods.

Property	Description
`value`	The specified attribute’s value.
`name`	The name of the attribute.

Fig. 14.31 | `Attr` properties.

Property	Description
`data`	The text contained in the node.
`length`	The number of characters contained in the node.

Fig. 14.32 | `Text` methods.

Locating Data in XML Documents with XPath

Although you can use XML DOM capabilities to navigate through and manipulate nodes, this is not the most efficient means of locating data in an XML document's DOM tree. A simpler way to locate nodes is to search for lists of nodes matching search criteria that are written as XPath expressions. Recall that XPath (XML Path Language) provides a syntax for locating specific nodes in XML documents effectively and efficiently. XPath is a string-based language of expressions used by XML and many of its related technologies (such as XSLT, discussed in Section 14.8).

Figure 14.33 enables the user to enter XPath expressions in an XHTML form. When the user clicks the **Get Matches** button, the script applies the XPath expression to the XML DOM and displays the matching nodes. Figure 14.34 shows the XML document `sports.xml` that we use in this example. [*Note:* The versions of `sports.xml` presented in Fig. 14.34 and Fig. 14.20 are nearly identical. In the current example, we do not want to apply an XSLT, so we omit the processing instruction found in line 2 of Fig. 14.20. We also removed extra blank lines to save space.]

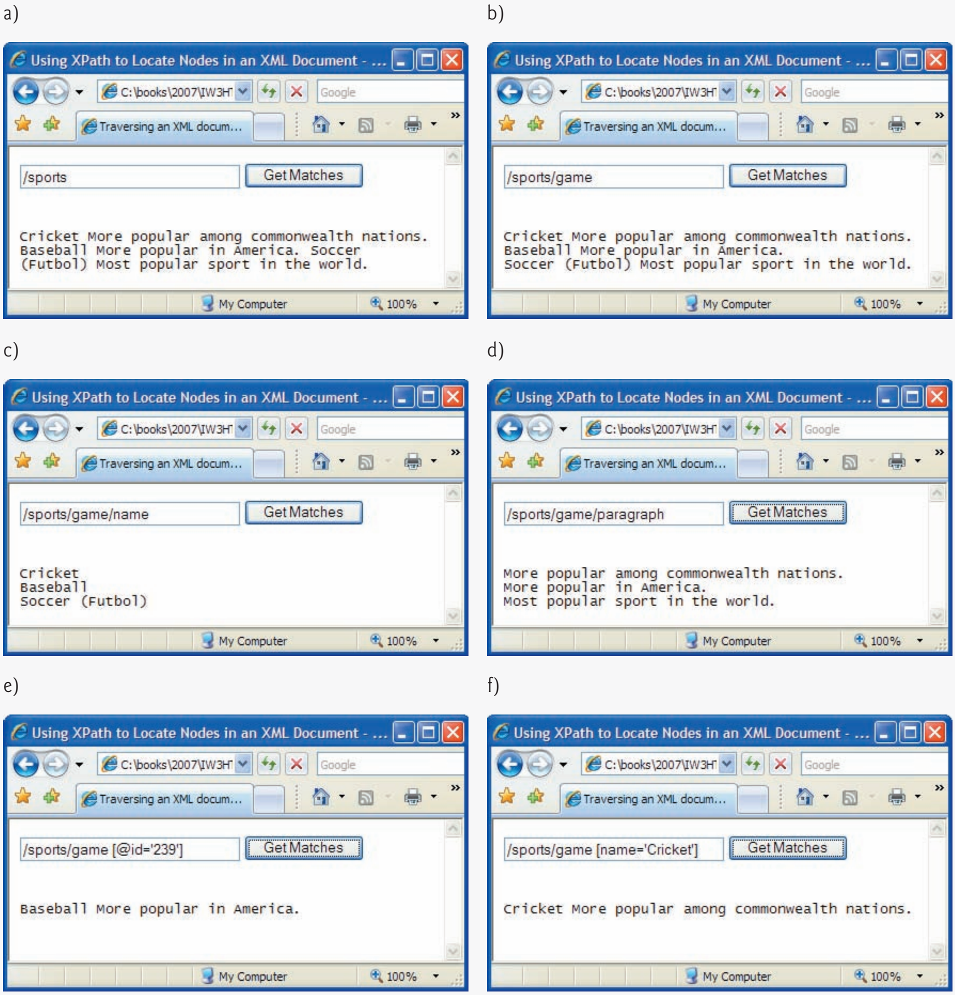
```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 14.33: xpath.html -->
<!-- Using XPath to locate nodes in an XML document. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
   <title>Using XPath to Locate Nodes in an XML Document</title>
   <style type = "text/css">
      #outputDiv { font: 10pt "Lucida Console", monospace; }
   </style>
   <script type = "text/javascript">
   <!--
   var doc; // variable to reference the XML document
   var outputHTML = ""; // stores text to output in outputDiv
   var browser = ""; // used to determine which browser is being used

   // load XML document based on whether the browser is IE7 or Firefox 2
   function loadXMLDocument( url )
   {
      if ( window.ActiveXObject ) // IE7
      {
```

Fig. 14.33 | Using XPath to locate nodes in an XML document. (Part 1 of 3.)

```
            // create IE7-specific XML document object
            doc = new ActiveXObject( "Msxml2.DOMDocument.6.0" );
            doc.async = false; // specifies synchronous loading of XML doc
            doc.load( url ); // load the XML document specified by url
            browser = "IE7"; // set browser
         } // end if
         else if ( document.implementation &&
            document.implementation.createDocument ) // other browsers
         {
            // create XML document object
            doc = document.implementation.createDocument( "", "", null );
            doc.load( url ); // load the XML document specified by url
            browser = "FF2"; // set browser
         } // end else
         else // not supported
            alert( 'This script is not supported by your browser' );
      } // end function loadXMLDocument

      // display the XML document
      function displayDoc()
      {
         document.getElementById( "outputDiv" ).innerHTML = outputHTML;
      } // end function displayDoc

      // obtain and apply XPath expression
      function processXPathExpression()
      {
         var xpathExpression = document.getElementById( "inputField" ).value;
         outputHTML = "";

         if ( browser == "IE7" )
         {
            var result = doc.selectNodes( xpathExpression );

            for ( var i = 0; i < result.length; i++ )
               outputHTML += "<div style='clear: both'>" +
                  result.item( i ).text + "</div>";
         } // end if
         else // browser == "FF2"
         {
            var result = document.evaluate( xpathExpression, doc, null,
               XPathResult.ANY_TYPE, null );
            var current = result.iterateNext();

            while ( current )
            {
               outputHTML += "<div style='clear: both'>" +
                  current.textContent + "</div>";
               current = result.iterateNext();
            } // end while
         } // end else

```

Fig. 14.33 | Using XPath to locate nodes in an XML document. (Part 2 of 3.)

```
      displayDoc();
   } // end function processXPathExpression
   // -->
   </script>
</head>
<body id = "body" onload = "loadXMLDocument( 'sports.xml' );">
   <form action = "" onsubmit = "return false;">
      <input id = "inputField" type = "text" style = "width: 200px"/>
      <input type = "submit" value = "Get Matches"
         onclick = "processXPathExpression()"/>
   </form><br/>
   <div id = "outputDiv"></div>
</body>
</html>
```


Using XPath to Locate Nodes in an XML Document - ...
C:\books\2007\IW3HT
Google
Traversing an XML docum...
Get Matches
My Computer
100%

a)

/sports

```
Cricket More popular among commonwealth nations.
Baseball More popular in America. Soccer
(Futbol) Most popular sport in the world.
```

b)

/sports/game

```
Cricket More popular among commonwealth nations.
Baseball More popular in America.
Soccer (Futbol) Most popular sport in the world.
```

c)

/sports/game/name

```
Cricket
Baseball
Soccer (Futbol)
```

d)

/sports/game/paragraph

```
More popular among commonwealth nations.
More popular in America.
Most popular sport in the world.
```

e)

/sports/game [@id='239']

```
Baseball More popular in America.
```

f)

/sports/game [name='Cricket']

```
Cricket More popular among commonwealth nations.
```

Fig. 14.33 | Using XPath to locate nodes in an XML document. (Part 3 of 3.)

```
<?xml version = "1.0"?>

<!-- Fig. 14.34: sports.xml -->
<!-- Sports Database       -->
<sports>
  <game id = "783">
      <name>Cricket</name>
      <paragraph>
         More popular among commonwealth nations.
      </paragraph>
   </game>
   <game id = "239">
      <name>Baseball</name>
      <paragraph>
         More popular in America.
      </paragraph>
   </game>
   <game id = "418">
      <name>Soccer (Futbol)</name>
      <paragraph>
         Most popular sport in the world.
      </paragraph>
   </game>
</sports>
```

Fig. 14.34 | XML document that describes various sports.

The program of Fig. 14.33 loads the XML document `sports.xml` (Fig. 14.34) using the same techniques we presented in Fig. 14.26, so we focus on only the new features in this example. Internet Explorer 7 (MSXML) and Firefox 2 handle XPath processing differently, so this example declares the variable `browser` (line 17) to store the browser that loaded the page. In function `loadDocument` (lines 20–40), lines 28 and 36 assign a string to variable `browser` indicating the appropriate browser.

When the `body` of this XHTML document loads, its `onload` event calls `loadDocument` (line 81) to load the `sports.xml` file. The user specifies the XPath expression in the `input` element at line 83. When the user clicks the **Get Matches** button (lines 84–85), its `onclick` event handler invokes our `processXPathExpression` function to locate any matches and display the results in `outputDiv` (line 87).

Function `processXPathExpression` (lines 49–77) first obtains the XPath expression (line 51). The `document` object's `getElementById` method returns the element with the `id` `"inputField"`; then we use its value property to get the XPath expression. Lines 54–61 apply the XPath expression in Internet Explorer 7, and lines 62–74 apply the XPath expression in Firefox 2. In IE7, the XML document object's **`selectNodes` method** receives an XPath expression as an argument and returns a collection of elements that match the expression. Lines 58–60 iterate through the results and mark up each one in a separate `div` element. After this loop completes, line 76 displays the generated markup in `outputDiv`.

For Firefox 2, lines 64–65 invoke the XML document object's **`evaluate` method**, which receives five arguments—the XPath expression, the document to apply the expression to, a namespace resolver, a result type and an `XPathResult` object into which to place the results. If the last argument is `null`, the function simply returns a new **`XPathResult`** object containing the matches. The namespace resolver argument can be null if you are not using XML namespace prefixes in the XPath processing. Lines 66–73 iterate through the XPathResult and mark up the results. Line 66 invokes the XPathResult's iterateNext method to position to the first result. If there is a result, the condition in line 68 will be true, and lines 70–71 create a div for that result. Line 72 then positions to the next result. After this loop completes, line 76 displays the generated markup in outputDiv.

Figure 14.35 summarizes the XPath expressions that we demonstrate in Fig. 14.33's sample outputs. For more information on using XPath in Firefox, visit the site developer.mozilla.org/en/docs/XPath. For more information on using XPath in Internet Explorer, visit msdn.microsoft.com/msdnmag/issues/0900/xml/.

Expression	Description
`/sports`	Matches all sports nodes that are child nodes of the document root node.
`/sports/game`	Matches all game nodes that are child nodes of sports, which is a child of the document root.
`/sports/game/name`	Matches all name nodes that are child nodes of game. The game is a child of sports, which is a child of the document root.
`/sports/game/paragraph`	Matches all paragraph nodes that are child nodes of game. The game is a child of sports, which is a child of the document root.
`/sports/game [@id='239']`	Matches the game node with the id number 239. The game is a child of sports, which is a child of the document root.
`/sports/game [name='Cricket']`	Matches all game nodes that contain a child element whose name is Cricket. The game is a child of sports, which is a child of the document root.

Fig. 14.35 | XPath expressions and descriptions.

14.10 RSS

RSS stands for RDF (Resource Description Framework) Site Summary and is also known as Rich Site Summary and Really Simple Syndication. RSS is an XML format used to syndicate website content, such as news articles, blog entries, product reviews, podcasts, vodcasts and more for inclusion on other websites. An RSS feed contains an **rss** root element with a version attribute and a **channel** child element with **item** subelements. Depending on the RSS version, the channel and item elements have certain required and optional child elements. The item elements provide the feed subscriber with a link to a web page or file, a title and description of the page or file. The most commonly used RSS feed versions are 0.91, 1.0, and 2.0, with RSS 2.0 being the most popular version. We discuss only RSS version 2.0 in this section.

RSS version 2.0, introduced in 2002, builds upon the RSS 0.9x versions. Version 2.0 does not contain length limitations or item element limitations of earlier versions, makes some formerly required elements optional, and adds new channel and item subelements. Removing length limitations on item descriptions allows RSS feeds to contain entire articles, blog entries and other web content. You can also have partial feeds that provide only a summary of the syndicated content. Partial feeds require the RSS subscriber to visit a website to view the complete content. RSS 2.0 allows item elements to contain an enclosure element providing the location of a media file that is related to the item. Such enclosures enable syndication of audio and video (such as podcasts and vodcasts) via RSS feeds.

By providing up-to-date, linkable content for anyone to use, RSS enables website developers to draw more traffic. It also allows users to get news and information from many sources easily and reduces content development time. RSS simplifies importing information from portals, weblogs and news sites. Any piece of information can be syndicated via RSS, not just news. After putting information in RSS format, an RSS program, such as a feed reader or aggregator, can check the feed for changes and react to them. For more details on RSS and for links to many RSS sites, visit our RSS Resource Center at www.deitel.com/RSS.

*RSS 2.0 **channel** and **item** Elements*

In RSS 2.0, the required child elements of channel are description, link and title, and the required child element of an item is either title or description. Figures 14.36–14.37 overview the child elements of channels and items, respectively.

Element	Description
title	The name of the channel or feed.
link	The URL to the website of the channel or feed the RSS is coming from.
description	A description of the channel or feed.
language	The language the channel is in, using W3C language values.
copyright	The copyright material of the channel or feed.
managingEditor	The e-mail address of the editor of the channel or feed.
webMaster	The e-mail address for the webmaster of the channel or feed.
pubDate	The date of the channel or feed release, using the RFC 822 Date and Time Specification—e.g., Sun, 14 Jan 2007 8:00:00 EST.
lastBuildDate	The last date the channel or feed was changed, using the RFC 822 Date and Time Specification.
category	The category (or several categories) of the channel or feed. This element has an optional attribute tag.
generator	Indicates the program that was used to generate the channel or feed.

Fig. 14.36 | channel elements and descriptions. (Part 1 of 2.)

Element	Description
`docs`	The URL of the documentation for the format used in the RSS file.
`cloud`	Specifies a SOAP web service that supports the `rssCloud` interface (`cyber.law.harvard.edu/rss/soapMeetsRss.html#rsscloudInterface`).
`ttl`	(Time To Live) A number of minutes for how long the `channel` or feed can be cached before refreshing from the source.
`image`	The GIF, JPEG or PNG image that can be displayed with the `channel` or feed. This element contains the required children `title`, `link` and `url`, and the optional children `description`, `height` and `width`.
`rating`	The PICS (Platform for Internet Content Selection) rating for the `channel` or feed.
`textInput`	Specifies a text input box to display with the `channel` or feed. This element contains the required children `title`, `name`, `link` and `description`.
`skipHours`	Tells aggregators which hours they can skip checking for new content.
`skipDays`	Tells aggregators which days they can skip checking for new content.

Fig. 14.36 | `channel` elements and descriptions. (Part 2 of 2.)

Element	Description
`title`	The title of the `item`.
`link`	The URL of the `item`.
`description`	The description of the `item`.
`author`	The e-mail address of the author of the `item`.
`category`	The category (or several categories) of the `item`. This element has an optional `attribute` tag.
`comments`	The URL of a page for comments related to the `item`.
`enclosure`	The location of a media object attached to the `item`. This element has the required attributes `type`, `url` and `length`.
`guid`	(Globally Unique Identifier) A string that uniquely identifies the `item`.
`pubDate`	The date the item was published, using the RFC 822 Date and Time Specification—e.g., Sun, 14 Jan 2007 8:00:00 EST.
`source`	The RSS channel the `item` came from. This element has a required attribute `url`.

Fig. 14.37 | `item` elements and descriptions.

Browsers and RSS Feeds

Many of the latest web browsers can now view RSS feeds, determine whether a website offers feeds, allow you to subscribe to feeds and create feed lists. An **RSS aggregator** keeps tracks of many RSS feeds and brings together information from the separate feeds. There are many RSS aggregators available, including Bloglines, BottomFeeder, FeedDemon, Microsoft Internet Explorer 7, Mozilla Firefox 2.0, My Yahoo, NewsGator and Opera 9.

To allow browsers and search engines to determine whether a web page contains an RSS feed, a `link` element can be added to the `head` of a page as follows:

```
<link rel = "alternate" type = "application/rss+xml" title = "RSS"
href = "file">
```

Many sites provide RSS feed validators. Some examples of RSS feed validators are `validator.w3.org/feed`, `feedvalidator.org`, and `www.validome.org/rss-atom/`.

Creating a Feed Aggregator

The DOM and XSL can be used to create RSS aggregators. A simple RSS aggregator uses an XSL stylesheet to format RSS feeds as XHTML. Figure 14.38 loads two XML documents—an RSS feed (a small portion of which is shown in Fig. 14.39) and an XSL style sheet—then uses JavaScript to apply an XSL transformation to the RSS content and render it on the page. You'll notice as we discuss this program that there is little commonality between Internet Explorer 7 and Firefox with regard to programmatically applying XSL transformations. This is one of the reasons that JavaScript libraries have become popular in web development—they tend to hide such browser-specific issues from you. We discuss the Dojo toolkit—one of many popular JavaScript libraries—in Section 15.8. For more information on JavaScript libraries, see our JavaScript and Ajax Resource Centers (`www.deitel.com/JavaScript/` and `www.deitel.com/Ajax/`, respectively).

Determining the Browser Type and Loading the Documents

When this page first loads, lines 19–23 (Fig. 14.38) determine whether the browser is Internet Explorer 7 or Firefox 2 and store the result in variable `browser` for use throughout the script. After the body of this XHTML document loads, its `onload` event calls function `start` (lines 26–48) to load RSS and XSL files as XML documents, and to transform the RSS. Since Internet Explorer 7 can download the files synchronously, lines 30–33 perform the loading, transformation and display steps sequentially. As mentioned previously, Firefox 2 loads the files asynchronously. For this reason, line 37 starts loading the `rss.xsl` document (included with this example's code), and lines 38–46 register an `onload` event handler for that document. When the document finishes loading, line 40 begins loading the `deitel-20.xml` RSS document. Lines 41–45 register an `onload` event handler for this second document. When it finishes loading, lines 43–44 perform the transformation and display the results.

Transforming the RSS to XHTML

Function `applyTransform` (Fig. 14.38, lines 75–96) performs the browser-specific XSL transformations using the RSS document and XSL document it receives as arguments. Line 81 uses the MSXML object's built-in XSLT capabilities to apply the transformations. Method **`transformNode`** is invoked on the `rssDocument` object and receives the `xslDocument` object as an argument.

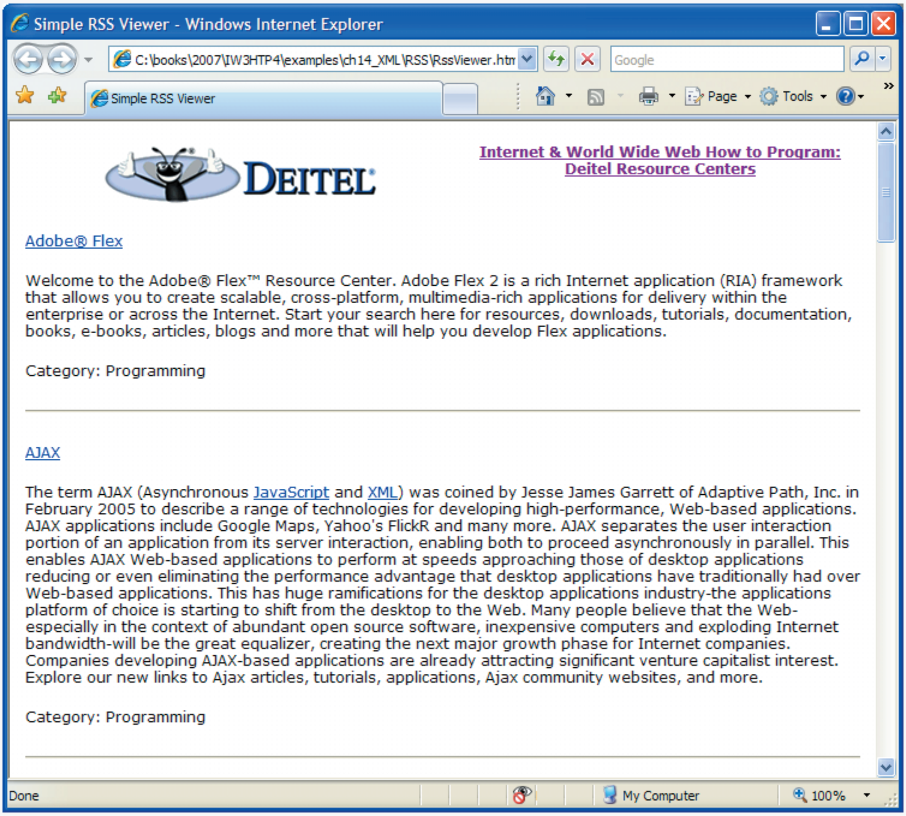
```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 14.38: RssViewer.html -->
<!-- Simple RSS viewer. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
   <title>Simple RSS Viewer</title>
   <style type = "text/css">
      #outputDiv { font: 12px Verdana, Geneva, Arial,
                   Helvetica, sans-serif; }
   </style>
   <script type = "text/javascript">
   <!--
   var browser = ""; // used to determine which browser is being used

   // is the browser Internet Explorer 7 or Firefox 2?
   if ( window.ActiveXObject ) // IE7
      browser = "IE7";
   else if ( document.implementation &&
      document.implementation.createDocument ) // FF2 and other browsers
      browser = "FF2";

   // load both the RSS feed and the XSL file to process it
   function start()
   {
      if ( browser == "IE7" )
      {
         var xsl = loadXMLDocument( 'rss.xsl' ); // load XSL file
         var rss = loadXMLDocument( 'deitel-20.xml' ); // load RSS feed
         var result = applyTransform( rss, xsl ); // apply transform
         displayTransformedRss( result ); // display feed info
      } // end if
      else if ( browser == "FF2" )
      {
         var xsl = loadXMLDocument( 'rss.xsl' ); // load XSL file
         xsl.onload = function() // function to execute when xsl loads
         {
            var rss = loadXMLDocument( 'deitel-20.xml' ); // load RSS feed
            rss.onload = function() // function to execute when rss loads
            {
               var result = applyTransform( rss, xsl ); // apply transform
               displayTransformedRss( result ); // display feed info
            } // end onload event handler for rss
         } // end onload event handler for xsl
      } // end else
   } // end function start

   // load XML document based on whether the browser is IE7 or Firefox 2
   function loadXMLDocument( url )
   {
      var doc = ""; // variable to manage loading file
```

Fig. 14.38 | Rendering an RSS feed in a web page using XSLT and JavaScript. (Part 1 of 3.)

```
         if ( browser == "IE7" ) // IE7
         {
            // create IE7-specific XML document object
            doc = new ActiveXObject( "Msxml2.DOMDocument.6.0" );
            doc.async = false; // specifies synchronous loading of XML doc
            doc.load( url ); // load the XML document specified by url
         } // end if
         else if ( browser == "FF2" ) // other browsers
         {
            // create XML document object
            doc = document.implementation.createDocument( "", "", null );
            doc.load( url ); // load the XML document specified by url
         } // end else
         else // not supported
            alert( 'This script is not supported by your browser' );

         return doc; // return the loaded document
      } // end function loadXMLDocument

      // apply XSL transformation and show results
      function applyTransform( rssDocument, xslDocument )
      {
         var result; // stores transformed RSS

         // transform the RSS feed to XHTML
         if ( browser == "IE7" )
            result = rssDocument.transformNode( xslDocument );
         else // browser == "FF2"
         {
            // create Firefox object to perform transformation
            var xsltProcessor = new XSLTProcessor();

            // specify XSL stylesheet to use in transformation
            xsltProcessor.importStylesheet( xslDocument );

            // apply the transformation
            result =
               xsltProcessor.transformToFragment( rssDocument, document );
         } // end else

         return result; // return the transformed RSS
      } // end function applyTransform

      // display the XML document and highlight the first child
      function displayTransformedRss( resultXHTML )
      {
         if ( browser == "IE7" )
            document.getElementById( "outputDiv" ).innerHTML = resultXHTML;
         else // browser == "FF2"
            document.getElementById( "outputDiv" ).appendChild(
               resultXHTML );
      } // end function displayTransformedRss
```

Fig. 14.38 | Rendering an RSS feed in a web page using XSLT and JavaScript. (Part 2 of 3.)

```
   // -->
   </script>
</head>
<body id = "body" onload = "start();">
   <div id = "outputDiv"></div>
</body>
</html>
```


Simple RSS Viewer - Windows Internet Explorer
C:\books\2007\IW3HTP4\examples\ch14_XML\RSS\RssViewer.htm
Google
Simple RSS Viewer
Page
Tools
DEITEL
Done
My Computer
100%

Internet & World Wide Web How to Program: Deitel Resource Centers

Adobe® Flex

Welcome to the Adobe® Flex™ Resource Center. Adobe Flex 2 is a rich Internet application (RIA) framework that allows you to create scalable, cross-platform, multimedia-rich applications for delivery within the enterprise or across the Internet. Start your search here for resources, downloads, tutorials, documentation, books, e-books, articles, blogs and more that will help you develop Flex applications.

Category: Programming

AJAX

The term AJAX (Asynchronous JavaScript and XML) was coined by Jesse James Garrett of Adaptive Path, Inc. in February 2005 to describe a range of technologies for developing high-performance, Web-based applications. AJAX applications include Google Maps, Yahoo's FlickR and many more. AJAX separates the user interaction portion of an application from its server interaction, enabling both to proceed asynchronously in parallel. This enables AJAX Web-based applications to perform at speeds approaching those of desktop applications reducing or even eliminating the performance advantage that desktop applications have traditionally had over Web-based applications. This has huge ramifications for the desktop applications industry-the applications platform of choice is starting to shift from the desktop to the Web. Many people believe that the Web-especially in the context of abundant open source software, inexpensive computers and exploding Internet bandwidth-will be the great equalizer, creating the next major growth phase for Internet companies. Companies developing AJAX-based applications are already attracting significant venture capitalist interest. Explore our new links to Ajax articles, tutorials, applications, Ajax community websites, and more.

Category: Programming

Fig. 14.38 | Rendering an RSS feed in a web page using XSLT and JavaScript. (Part 3 of 3.)

```
<?xml version="1.0" encoding="utf-8"?>

<!-- Fig. 14.39: deitel-20.xml -->
<!-- RSS 2.0 feed of Deitel Resource Centers -->
<rss version="2.0">
   <channel>
      <title>
         Internet &#38; World Wide Web How to Program:
         Deitel Resource Centers
      </title>
```

Fig. 14.39 | RSS 2.0 sample feed. (Part 1 of 2.)

```
      <link>http://www.deitel.com/ResourceCenters.html</link>
      <description>
         Check out our growing network of Resource Centers that focus on
         many of today's hottest programming, Web 2.0 and technology
         topics. Start your search here for downloads, tutorials,
         documentation, books, e-books, blogs, RSS feeds, journals,
         articles, training, webcasts, podcasts, videos and more.
      </description>
      <language>en-us</language>
      <image>
         <url>
            http://www.deitel.com/Portals/0/deitel_transparent_smaller.png
         </url>
         <title>Deitel.com</title>
         <link>http://www.deitel.com/</link>
      </image>

      <item>
         <title>Adobe® Flex</title>
         <link>http://www.deitel.com/Flex/</link>
         <description>
            <p>
            Welcome to the Adobe® Flex™ Resource Center. Adobe Flex 2 is a
            rich Internet application (RIA) framework that allows you to
            create scalable, cross-platform, multimedia-rich applications
            for delivery within the enterprise or across the Internet.
            Start your search here for resources, downloads, tutorials,
            documentation, books, e-books, articles, blogs and more that
            will help you develop Flex applications.
            </p>
         </description>
         <category>Programming</category>
      </item>
   </channel>
</rss>
```

Fig. 14.39 | RSS 2.0 sample feed. (Part 2 of 2.)

Firefox provides built-in XSLT processing in the form of the **`XSLTProcessor`** object (created at line 85). After creating this object, you use its **`importStylesheet`** method to specify the XSL stylesheet you'd like to apply (line 88). Finally, lines 91–92 apply the transformation by invoking the `XSLTProcessor` object's **`transformToFragment`** method, which returns a document fragment—i.e., a piece of a document. In our case, the `rss.xsl` document transforms the RSS into an XHTML `table` element that we'll append to the `outputDiv` element in our XHTML page. The arguments to `transformToFragment` are the document to transform and the document object to which the transformed fragment will belong. To learn more about `XSLTProcessor`, visit `developer.mozilla.org/en/docs/The_XSLT/JavaScript_Interface_in_Gecko`.

In each browser's case, after the transformation, the resulting XHTML markup is assigned to variable `result` and returned from function `applyTransform`. Then function `displayTransformedRss` is called.

Displaying the XHTML Markup

Function `displayTransformedRss` (lines 99–106) displays the transformed RSS in the `outputDiv` element (line 111 in the `body`). In both Internet Explorer 7 and Firefox 2, we use the DOM method `getElementById` to obtain the `outputDiv` element. In Internet Explorer 7, the node's `innerHTML` property is used to add the table as a child of the `outputDiv` element (line 102). In Firefox, the node's `appendChild` method must be used to append the table (a document fragment) to the `outputDiv` element.

14.11 Wrap-Up

In this chapter, we studied Extensible Markup Language and several of its related technologies. We began by discussing some basic XML terminology, introducing the concepts of markup, XML vocabularies and XML parsers (validating and nonvalidating). We then demonstrated how to describe and structure data in XML, illustrating these points with examples marking up an article and a business letter.

The chapter discussed the concept of an XML namespace. You learned that each namespace has a unique name that provides a means for document authors to unambiguously refer to elements with the same name (i.e., prevent naming collisions). We presented examples of defining two namespaces in the same document, as well as setting the default namespace for a document.

We also discussed how to create DTDs and schemas for specifying and validating the structure of an XML document. We showed how to use various tools to confirm whether XML documents are valid (i.e., conform to a DTD or schema).

The chapter demonstrated how to create and use XSL documents to specify rules for converting XML documents between formats. Specifically, you learned how to format and sort XML data as XHTML for display in a web browser.

The final sections of the chapter presented more advanced uses of XML. We demonstrated how to retrieve and display data from an XML document using JavaScript. We illustrated how a Document Object Model (DOM) tree represents each element of an XML document as a node in the tree. You also learned how to traverse the DOM tree, interact with individual nodes in the DOM tree from JavaScript code, search for nodes using XPath and apply XSL transformations.

Chapter 15 begins our discussion of Rich Internet Applications (RIAs)—web applications that approximate the look, feel and usability of desktop applications. RIAs have two key attributes—performance and rich GUI. You'll learn about Ajax (Asynchronous JavaScript and XML), which uses all the concepts you've learned so far to build rich web applications. You'll see that Ajax techniques are key to the responsiveness of RIAs.

14.12 Web Resources

`www.deitel.com/XML/`

The Deitel XML Resource Center focuses on the vast amount of free XML content available online, plus some for-sale items. Start your search here for tools, downloads, tutorials, podcasts, wikis, documentation, conferences, FAQs, books, e-books, sample chapters, articles, newsgroups, forums, downloads from CNET's download.com, jobs and contract opportunities, and more that will help you develop XML applications.

Summary

Section 14.1 Introduction

- XML is a portable, widely supported, open (i.e., nonproprietary) technology for data storage and exchange.

Section 14.2 XML Basics

- XML documents are readable by both humans and machines.
- XML permits document authors to create custom markup for any type of information. This enables document authors to create entirely new markup languages that describe specific types of data, including mathematical formulas, chemical molecular structures, music and recipes.
- An XML parser is responsible for identifying components of XML documents (typically files with the `.xml` extension) and then storing those components in a data structure for manipulation.
- An XML document can optionally reference a Document Type Definition (DTD) or schema that defines the XML document's structure.
- An XML document that conforms to a DTD/schema (i.e., has the appropriate structure) is valid.
- If an XML parser (validating or nonvalidating) can process an XML document successfully, that XML document is well-formed.

Section 14.3 Structuring Data

- An XML document begins with an optional XML declaration, which identifies the document as an XML document. The `version` attribute specifies the version of XML syntax used in the document.
- XML comments begin with `<!--` and end with `-->`.
- An XML document contains text that represents its content (i.e., data) and elements that specify its structure. XML documents delimit an element with start and end tags.
- The root element of an XML document encompasses all its other elements.
- XML element names can be of any length and can contain letters, digits, underscores, hyphens and periods. However, they must begin with either a letter or an underscore, and they should not begin with "`xml`" in any combination of uppercase and lowercase letters, as this is reserved for use in the XML standards.
- When a user loads an XML document in a browser, a parser parses the document, and the browser uses a style sheet to format the data for display.
- IE and Firefox each display minus (**–**) or plus (**+**) signs next to all container elements. A minus sign indicates that all child elements are being displayed. When clicked, a minus sign becomes a plus sign (which collapses the container element and hides all the children), and vice versa.
- Data can be placed between tags or in attributes (name/value pairs that appear within the angle brackets of start tags). Elements can have any number of attributes.

Section 14.4 XML Namespaces

- XML allows document authors to create their own markup, and as a result, naming collisions (i.e., two different elements that have the same name) can occur. XML namespaces provide a means for document authors to prevent collisions.
- Each namespace prefix is bound to a uniform resource identifier (URI) that uniquely identifies the namespace. A URI is a series of characters that differentiate names. Document authors create their own namespace prefixes. Any name can be used as a namespace prefix, but the namespace prefix `xml` is reserved for use in XML standards.

- To eliminate the need to place a namespace prefix in each element, authors can specify a default namespace for an element and its children. We declare a default namespace using keyword `xmlns` with a URI (Uniform Resource Identifier) as its value.
- Document authors commonly use URLs (Uniform Resource Locators) for URIs, because domain names (e.g., `deitel.com`) in URLs must be unique.

Section 14.5 Document Type Definitions (DTDs)

- DTDs and schemas specify documents' element types and attributes, and their relationships to one another.
- DTDs and schemas enable an XML parser to verify whether an XML document is valid (i.e., its elements contain the proper attributes and appear in the proper sequence).
- A DTD expresses the set of rules for document structure using an EBNF (Extended Backus-Naur Form) grammar.
- In a DTD, an `ELEMENT` element type declaration defines the rules for an element. An `ATTLIST` attribute-list declaration defines attributes for a particular element.

Section 14.6 W3C XML Schema Documents

- Unlike DTDs, schemas do not use EBNF grammar. Instead, they use XML syntax and are themselves XML documents that programs can manipulate.
- Unlike DTDs, XML Schema documents can specify what type of data (e.g., numeric, text) an element can contain.
- An XML document that conforms to a schema document is schema valid.
- Two categories of types exist in XML Schema: simple types and complex types. Simple types cannot contain attributes or child elements; complex types can.
- Every simple type defines a restriction on an XML Schema-defined schema type or on a user-defined type.
- Complex types can have either simple content or complex content. Both simple and complex content can contain attributes, but only complex content can contain child elements.
- Whereas complex types with simple content must extend or restrict some other existing type, complex types with complex content do not have this limitation.

Section 14.7 XML Vocabularies

- XML allows authors to create their own tags to describe data precisely.
- Some of these XML vocabularies includ MathML (Mathematical Markup Language), Scalable Vector Graphics (SVG), Wireless Markup Language (WML), Extensible Business Reporting Language (XBRL), Extensible User Interface Language (XUL), Product Data Markup Language (PDML), W3C XML Schema and Extensible Stylesheet Language (XSL).
- MathML markup describes mathematical expressions for display. MathML is divided into two types of markup—content markup and presentation markup.
- Content markup provides tags that embody mathematical concepts. Content MathML allows programmers to write mathematical notation specific to different areas of mathematics.
- Presentation MathML is directed toward formatting and displaying mathematical notation. We focus on Presentation MathML in the MathML examples.
- By convention, MathML files end with the `.mml` filename extension.
- A MathML document's root node is the `math` element and its default namespace is `http://www.w3.org/1998/Math/MathML`.

- The mn element marks up a number. The mo element marks up an operator.
- Entity reference &InvisibleTimes; indicates a multiplication operation without explicit symbolic representation.
- The msup element represents a superscript. It has two children—the expression to be superscripted (i.e., the base) and the superscript (i.e., the exponent). Correspondingly, the msub element represents a subscript.
- To display variables, use identifier element mi.
- The mfrac element displays a fraction. If either the numerator or the denominator contains more than one element, it must appear in an mrow element.
- An mrow element is used to group elements that are positioned horizontally in an expression.
- The entity reference &int; represents the integral symbol.
- The msubsup element specifies the subscript and superscript of a symbol. It requires three child elements—an operator, the subscript expression and the superscript expression.
- Element msqrt represents a square-root expression.
- Entity reference &delta; represents a lowercase delta symbol.

Section 14.8 Extensible Stylesheet Language and XSL Transformations

- XSL can convert XML into any text-based document. XSL documents have the extension .xsl.
- XPath is a string-based language of expressions used by XML and many of its related technologies for effectively and efficiently locating structures and data (such as specific elements and attributes) in XML documents.
- XPath is used to locate parts of the source-tree document that match templates defined in an XSL style sheet. When a match occurs (i.e., a node matches a template), the matching template executes and adds its result to the result tree. When there are no more matches, XSLT has transformed the source tree into the result tree.
- The XSLT does not analyze every node of the source tree; it selectively navigates the source tree using XPath's select and match attributes.
- For XSLT to function, the source tree must be properly structured. Schemas, DTDs and validating parsers can validate document structure before using XPath and XSLTs.
- XSL style sheets can be connected directly to an XML document by adding an xml:stylesheet processing instruction to the XML document.
- Two tree structures are involved in transforming an XML document using XSLT—the source tree (the document being transformed) and the result tree (the result of the transformation).
- The XPath character / (a forward slash) always selects the document root. In XPath, a leading forward slash specifies that we are using absolute addressing.
- An XPath expression with no beginning forward slash uses relative addressing.
- XSL element value-of retrieves an attribute's value. The @ symbol specifies an attribute node.
- XSL node-set function name retrieves the current node's element name.
- XSL node-set function text retrieves the text between an element's start and end tags.
- The XPath expression //* selects all the nodes in an XML document.

Section 14.9 Document Object Model (DOM)

- Although an XML document is a text file, retrieving data from the document using traditional sequential file processing techniques is neither practical nor efficient, especially for adding and removing elements dynamically.

- Upon successfully parsing a document, some XML parsers store document data as tree structures in memory. This hierarchical tree structure is called a Document Object Model (DOM) tree, and an XML parser that creates this type of structure is known as a DOM parser.
- Each element name is represented by a node. A node that contains other nodes is called a parent node. A parent node can have many children, but a child node can have only one parent node.
- Nodes that are peers are called sibling nodes.
- A node's descendant nodes include its children, its children's children and so on. A node's ancestor nodes include its parent, its parent's parent and so on.
- Many of the XML DOM capabilities are similar or identical to those of the XHTML DOM.
- The DOM tree has a single root node, which contains all the other nodes in the document.
- If `window.ActiveXObject` exists, the browser is Internet Explorer. An `ActiveXObject` that loads Microsoft's MSXML parser is used to manipulate XML documents in Internet Explorer.
- MSXML's `load` method loads an XML document.
- A document's `childNodes` property contains a list of the XML document's top-level nodes.
- If the browser is Firefox 2, then the `document` object's `implementation` property and the `implementation` property's `createDocument` method will exist.
- Firefox loads each XML document asynchronously, so you must use the XML document's `onload` property to specify a function to call when the document finishes loading to ensure that you can access the document's contents.
- A node's `nodeType` property contains the type of the node.
- Nonbreaking spaces (`&nbsp;`) are spaces that the browser is not allowed to collapse or that can be used to keep words together.
- The name of an element can be obtained by the node's `nodeName` property.
- If the currrent node has children, the length of the node's `childNodes` list is nonzero.
- The `nodeValue` property returns the value of an element.
- Node property `firstChild` refers to the first child of a given node. Similarly, `lastChild` refers to the last child of a given node.
- Node property `nextSibling` refers to the next sibling in a list of children of a particular node. Similarly, `previousSibling` refers to the current node's previous sibling.
- Property `parentNode` refers to the current node's parent node.
- A simpler way to locate nodes is to search for lists of node-matching search criteria that are written as XPath expressions.
- In IE7, the XML document object's `selectNodes` method receives an XPath expression as an argument and returns a collection of elements that match the expression.
- Firefox 2 searches for XPath matches using the XML document object's `evaluate` method, which receives five arguments—the XPath expression, the document to apply the expression to, a namespace resolver, a result type and an `XPathResult` object into which to place the results. If the last argument is `null`, the function simply returns a new `XPathResult` object containing the matches. The namespace resolver argument can be `null` if you are not using XML namespace prefixes in the XPath processing.

Section 14.10 RSS

- RSS stands for RDF (Resource Description Framework) Site Summary and is also known as Rich Site Summary and Really Simple Syndication.

- RSS is an XML format used to syndicate simple website content, such as news articles, blog entries, product reviews, podcasts, vodcasts and more.
- An RSS feed contains an `rss` root element with a `version` attribute and a `channel` child element with `item` subelements. Depending on the RSS version, the `channel` and `item` elements have certain required and optional child elements.
- The `item` elements provide the feed subscriber with a link to a web page or file, a title and description of the page or file.
- By providing up-to-date, free and linkable content for anyone to use, RSS enables website developers to draw more traffic.
- In RSS 2.0, the required child elements of `channel` are `description`, `link` and `title`, and the required child element of an `item` is either `title` or `description`.
- An RSS aggregator keeps tracks of many RSS feeds and brings together information from the separate feeds.
- Many sites provide RSS feed validators. Some examples of RSS feed validators are `validator.w3.org/feed`, `feedvalidator.org`, and `www.validome.org/rss-atom/`.
- The DOM and XSL can be used to create RSS aggregators. A simple RSS aggregator uses an XSL style sheet to format RSS feeds as XHTML.
- MSXML's built-in XSLT capabilities include method `transformNode` to apply an XSLT transformation. It is invoked on an RSS document object and receives the XSL document object as an argument.
- Firefox provides built-in XSLT processing in the form of the `XSLTProcessor` object. After creating this object, you use its `importStylesheet` method to specify the XSL style sheet you'd like to apply. Finally, you apply the transformation by invoking the `XSLTProcessor` object's `transformToFragment` method, which returns a document fragment.

Terminology

`&delta;` entity reference (MathML)
`&InvisibleTimes;` entity reference (MathML)
`&int;` entity reference (MathML)
`&nbsp;`
`.mml` filename extension for MathML documents
`/`, forward slash in end tags
`/`, XPath root selector
`<!--…-->`, XML comment tags
`<?` and `?>` XML processing instruction delimiters
`@`, XPath attribute symbol
absolute addressing (XPath)
`ActiveXObject` from Internet Explorer
`all` XML Schema element
Amaya (W3C browser)
ancestor node
`appendChild` method of a `Node`
asterisk (*) occurrence indicator
`ATTLIST` attribute-list declaration (DTD)
`Attr` object
`attribute` element
attribute in XML
attribute-list declaration
attribute value in XML
`base` attribute of element `extension`
base type (XML Schema)
`CDATA` keyword (DTD)
`channel` child element of an `rss` element
character data in XML
child element
child node (DOM tree)
`childNodes` property of a `Node`
complex content in XML Schema
`complexType` XML Schema element
container element
content
context node (XPath)
`createDocument` method of the `document` object's `implementation` property
`data-type` attribute (XPath)
default namespace

descendant node
DOCTYPE parts
document object
Document Object Model (DOM) tree
document root
Document Type Definition (DTD)
DOM parser
.dtd filename extension
element (XML)
ELEMENT element type declaration (DTD)
Element object
element type declaration
element XML Schema element
EMPTY keyword (DTD)
enclosure element (RSS)
end tag
evaluate method of a Firefox 2 XML document object
Expat XML Parser
Extensible Stylesheet Language (XSL)
Extensible User Interface Language (XUL)
base attribute
extension XML Schema element
external DTD
firstChild property of a DOM node
#FIXED keyword (DTD)
forward slash character (/) in end tags
getElementById method of the document object
getElementsByTagName method
identifier element (MathML)
implementation property
implementation property of the document object
#IMPLIED keyword (DTD)
importStylesheet method of the XSLTProcessor object (Firefox)
item subelement of channel child element of an rss element
lastChild property of a DOM node
load method of the ActiveXObject object
markup in XML
match attribute
Mathematical Markup Language (MathML)
maxOccurs XML Schema attribute
mfrac MathML element
mi MathML element
Microsoft XML Core Services (MSXML)
minInclusive XML Schema element
minOccurs XML Schema attribute
mn MathML element
mo MathML element
mrow MathML element
msqrt MathML element
msubsup MathML element
msup MathML element
MSXML (Microsoft XML Core Services)
MSXML parser
name attribute (XPath)
name node-set function
name XML Schema attribute
namespace prefix
naming collision
nested element
nextSibling property of a DOM node
Node object
node-set function
for-each element
NodeList object
nodeName property of a DOM node
nodeType property of a DOM node
nodeValue property of a DOM node
nonbreaking space (&nbsp;)
nonvalidating XML parser
occurrence indicator
omit-xml-declaration attribute
open technology
order attribute
parent element
parent node
parentNode property of a DOM node
parsed character data
parser
partial RSS feed
#PCDATA keyword (DTD)
plus sign (+) occurrence indicator
presentation
previousSibling property of a Node
processing instruction (PI)
processing instruction target
processing instruction value
Product Data Markup Language (PDML)
prolog (XML)
question mark (?) occurrence indicator
RDF (Resource Description Framework)
RDF Site Summary (RSS)
Really Simple Syndication (RSS)
recursive descent
relative addressing (XPath)
replaceChild method of a Node
#REQUIRED keyword (DTD)

Research Information Exchange Markup Language (RIXML) 863
Resource Description Framework (RDF)
restriction on built-in XML Schema data type
result tree (XSLT)
Rich Site Summary (RSS)
root element (XML)
root node
RSS (RDF Site Summary)
RSS 2.0 sample feed
RSS aggregator
rss root element
Scalable Vector Graphics (SVG)
schema invalid document
schema repository
schema valid XML document
schema XML Schema element
select attribute (XPath)
select attribute of xsl:for-each element
selectNodes method of MSXML
sibling node
siblings
simple content in XML Schema
simpleContent XML Schema element
simpleType XML Schema element
source tree (XSLT)
square-root symbol (MathML)
start tag
string XML Schema data type
style sheet
stylesheet start tag
sum function (XSL)
symbolic representation (MathML)
SYSTEM keyword in XML
targetNamespace XML Schema attribute
text node-set function
Text object
transformNode method of an MSXML document object
transformToFragment method of the XSLTProcessor object (Firefox)
type attribute in a processing instruction
type XML Schema attribute
validating XML parser
version attribute (XSL)
version in xml declaration
well-formed XML document
Wireless Markup Language (WML)
Xalan XSLT processor
XBRL (Extensible Business Reporting Language)
Xerces parser from Apache
XML (Extensible Markup Language)
.xml file extension
XML instance document
XML Path Language (XPath)
XML processor
XML Schema
XML vocabulary
xmlns attribute in XML
XPath (XML Path Language)
XPath expression
XPathResult object
.xsd filename extension
XSL (Extensible Stylesheet Language)
.xsl filename extension
XSL-FO (XSL Formatting Objects)
XSL Formatting Objects (XSL-FO)
XSL style sheet
XSL template
XSL variable
xsl:for-each element
xsl:output element
xsl:template element
xsl:value-of element
XSLTProcessor object (Firefox)
XUL (Extensible User Interface Language)

Self-Review Exercises

14.1 Which of the following are valid XML element names? (Select all that apply.)

a) yearBorn
b) year.Born
c) year Born
d) year-Born1
e) 2_year_born
f) _year_born_

14.2 State which of the following statements are *true* and which are *false*. If *false*, explain why.

a) XML is a technology for creating markup languages.
b) XML markup is delimited by forward and backward slashes (/ and \).
c) All XML start tags must have corresponding end tags.
d) Parsers check an XML document's syntax.
e) XML does not support namespaces.
f) When creating XML elements, document authors must use the set of XML tags provided by the W3C.
g) The pound character (#), dollar sign ($), ampersand (&) and angle brackets (< and >) are examples of XML reserved characters.
h) XML is not case sensitive.
i) XML Schemas are better than DTDs, because DTDs lack a way of indicating what specific type of data (e.g., numeric, text) an element can contain and DTDs are not themselves XML documents.
j) DTDs are written using an XML vocabulary.
k) Schema is a technology for locating information in an XML document.

14.3 Fill in the blanks for each of the following:

a) __________ help prevent naming collisions.
b) __________ embed application-specific information into an XML document.
c) __________ is Microsoft's XML parser.
d) XSL element __________ writes a `DOCTYPE` to the result tree.
e) XML Schema documents have root element __________.
f) XSL element __________ is the root element in an XSL document.
g) XSL element __________ selects specific XML elements using repetition.
h) Nodes that contain other nodes are called __________ nodes.
i) Nodes that are peers are called __________ nodes.

14.4 In Fig. 14.2, we subdivided the `author` element into more detailed pieces. How might you subdivide the `date` element? Use the date May 5, 2005, as an example.

14.5 Write a processing instruction that includes style sheet `wap.xsl`.

14.6 Write an XPath expression that locates `contact` nodes in `letter.xml` (Fig. 14.4).

Answers to Self-Review Exercises

14.1 a, b, d, f. [Choice c is incorrect because it contains a space. Choice e is incorrect because the first character is a number.]

14.2 a) True. b) False. In an XML document, markup text is delimited by tags enclosed in angle brackets (< and >) with a forward slash just after the < in the end tag. c) True. d) True. e) False. XML does support namespaces. f) False. When creating tags, document authors can use any valid name but should avoid ones that begin with the reserved word `xml` (also `XML`, `Xml`, etc.). g) False. XML reserved characters include the ampersand (&), the left angle bracket (<) and the right angle bracket (>), but not # and $. h)False. XML is case sensitive. i) True. j) False. DTDs use EBNF grammar, which is not XML syntax. k) False. XPath is a technology for locating information in an XML document. XML Schema provides a means for type checking XML documents and verifying their validity.

14.3 a) Namespaces. b) Processing instructions. c) MSXML. d) `xsl:output`. e) `schema`. f) `xsl:stylesheet`. g) `xsl:for-each`. h) parent. i) sibling.

14.4

```
<date>
   <month>May</month>
   <day>5</day>
   <year>2005</year>
</date>.
```

14.5 `<?xsl:stylesheet type = "text/xsl" href = "wap.xsl"?>`

14.6 `/letter/contact`.

Exercises

14.7 (*Nutrition Information XML Document*) Create an XML document that marks up the nutrition facts for a package of Grandma White's cookies. A package of cookies has a serving size of 1 package and the following nutritional value per serving: 260 calories, 100 fat calories, 11 grams of fat, 2 grams of saturated fat, 5 milligrams of cholesterol, 210 milligrams of sodium, 36 grams of total carbohydrates, 2 grams of fiber, 15 grams of sugars and 5 grams of protein. Name this document `nutrition.xml`. Load the XML document into Internet Explorer. [*Hint:* Your markup should contain elements describing the product name, serving size/amount, calories, sodium, cholesterol, proteins, etc. Mark up each nutrition fact/ingredient listed above.]

14.8 (*Nutrition Information XML Schema*) Write an XML Schema document (`nutrition.xsd`) specifying the structure of the XML document created in Exercise 14.7.

14.9 (*Nutrition Information XSL Style Sheet*) Write an XSL style sheet for your solution to Exercise 14.7 that displays the nutritional facts in an XHTML table. Modify Fig. 14.38 to output the results.

14.10 (*Sorting XSLT Modification*) Modify Fig. 14.23 (`sorting.xsl`) to sort by the number of pages rather than by chapter number. Save the modified document as `sorting_byPage.xsl`.

14.11 Modify Fig. 14.38 to use `sorting.xml` (Fig. 14.22), `sorting.xsl` (Fig. 14.23) and `sorting_byPage.xsl` (from Exercise 14.10). Display the result of transforming `sorting.xml` using each style sheet. [*Hint:* Remove the `xml:stylesheet` processing instruction from line 2 of `sorting.xml` before attempting to transform the file programmatically.]

15

Ajax-Enabled Rich Internet Applications

… the challenges are for the designers of these applications: to forget what we think we know about the limitations of the Web, and begin to imagine a wider, richer range of possibilities. It's going to be fun.

—Jesse James Garrett

Dojo is the standard library JavaScript never had.

—Alex Russell

To know how to suggest is the great art of teaching. To attain it we must be able to guess what will interest …

—Henri-Fredreic Amiel

It is characteristic of the epistemological tradition to present us with partial scenarios and then to demand whole or categorical answers as it were.

—Avrum Stroll

O! call back yesterday, bid time return.

—William Shakespeare

OBJECTIVES

In this chapter you will learn:

- What Ajax is and why it is important for building Rich Internet Applications.
- What asynchronous requests are and how they help give web applications the feel of desktop applications.
- What the `XMLHttpRequest` object is and how it's used to create and manage asynchronous requests to servers and to receive asynchronous responses from servers.
- Methods and properties of the `XMLHttpRequest` object.
- How to use XHTML, JavaScript, CSS, XML, JSON and the DOM in Ajax applications.
- How to use Ajax frameworks and toolkits, specifically Dojo, to conveniently create robust Ajax-enabled Rich Internet Applications.
- About resources for studying Ajax-related issues such as security, performance, debugging, the "back-button problem" and more.

Outline

15.1 Introduction

Despite the tremendous technological growth of the Internet over the past decade, the usability of web applications has lagged behind compared to that of desktop applications. Every significant interaction in a web application results in a waiting period while the application communicates over the Internet with a server. **Rich Internet Applications (RIAs)** are web applications that approximate the look, feel and usability of desktop applications. RIAs have two key attributes—performance and a rich GUI.

RIA performance comes from **Ajax (Asynchronous JavaScript and XML)**, which uses client-side scripting to make web applications more responsive. Ajax applications separate client-side user interaction and server communication, and run them in parallel, reducing the delays of server-side processing normally experienced by the user.

There are many ways to implement Ajax functionality. **"Raw" Ajax** uses JavaScript to send asynchronous requests to the server, then updates the page using the DOM (see Section 15.5). "Raw" Ajax is best suited for creating small Ajax components that asynchronously update a section of the page. However, when writing "raw" Ajax you need to deal directly with cross-browser portability issues, making it impractical for developing large-scale applications. These portability issues are hidden by **Ajax toolkits**, such as **Dojo** (Section 15.8), **Prototype**, **Script.aculo.us** and ASP.NET Ajax, which provide powerful ready-to-use controls and functions that enrich web applications, and simplify JavaScript coding by making it cross-browser compatible.

Traditional web applications use XHTML forms (Chapter 4) to build simple and thin GUIs compared to the rich GUIs of Windows, Macintosh and desktop systems in general. We achieve rich GUI in RIAs with Ajax toolkits and with RIA environments such as Adobe's Flex (Chapter 18), Microsoft's Silverlight (Chapter 19) and JavaServer Faces (Chapters 26–27). Such toolkits and environments provide powerful ready-to-use controls and functions that enrich web applications.

Previous chapters discussed XHTML, CSS, JavaScript, dynamic HTML, the DOM and XML. This chapter uses these technologies to build Ajax-enabled web applications. The client-side of Ajax applications is written in XHTML and CSS, and uses JavaScript to add functionality to the user interface. XML is used to structure the data passed between the server and the client. We'll also use JSON (JavaScript Object Notation) for this purpose. The Ajax component that manages interaction with the server is usually implemented with JavaScript's **`XMLHttpRequest` object**—commonly abbreviated as XHR. The server processing can be implemented using any server-side technology, such as PHP, ASP.NET, JavaServer Faces and Ruby on Rails—each of which we cover in later chapters.

This chapter begins with several examples that build basic Ajax applications using JavaScript and the `XMLHttpRequest` object. We then build an Ajax application with a rich calendar GUI using the Dojo Ajax toolkit. In subsequent chapters, we use tools such as Adobe Flex, Microsoft Silverlight and JavaServer Faces to build RIAs using Ajax. In Chapter 24, we'll demonstrate features of the Prototype and Script.aculo.us Ajax libraries, which come with the Ruby on Rails framework (and can be downloaded separately). Prototype provides capabilities similar to Dojo. Script.aculo.us provides many "eye candy" effects that enable you to beautify your Ajax applications and create rich interfaces. In Chapter 27, we present Ajax-enabled JavaServer Faces (JSF) components. JSF uses Dojo to implement many of its client-side Ajax capabilities.

15.2 Traditional Web Applications vs. Ajax Applications

In this section, we consider the key differences between traditional web applications and Ajax-based web applications.

Traditional Web Applications

Figure 15.1 presents the typical interactions between the client and the server in a traditional web application, such as one that uses a user registration form. First, the user fills in the form's fields, then submits the form (Fig. 15.1, *Step 1*). The browser generates a request to the server, which receives the request and processes it (*Step 2*). The server generates and sends a response containing the exact page that the browser will render (*Step 3*), which causes the browser to load the new page (*Step 4*) and temporarily makes the browser window blank. Note that the client *waits* for the server to respond and *reloads the entire page* with the data from the response (*Step 4*). While such a **synchronous request** is being pro-

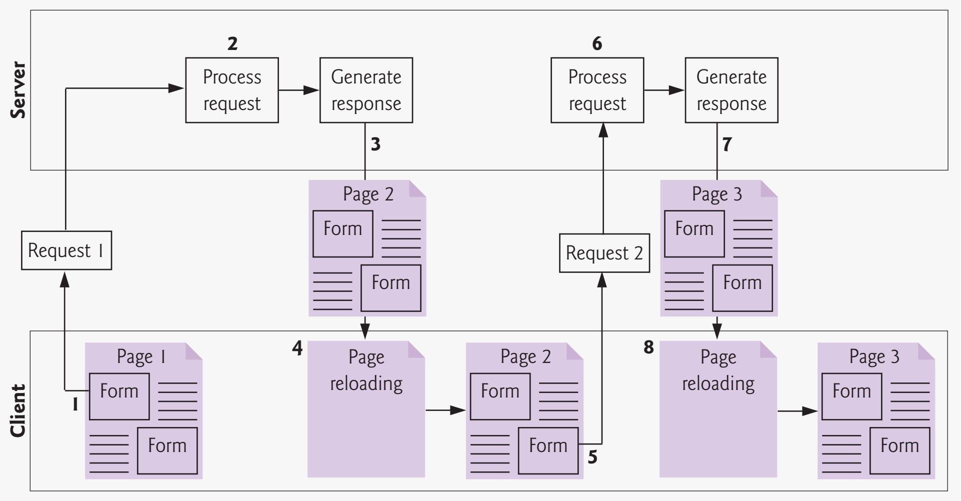


Fig. 15.1 | Classic web application reloading the page for every user interaction.

cessed on the server, the user cannot interact with the client web page. Frequent long periods of waiting, due perhaps to Internet congestion, have led some users to refer to the World Wide Web as the "World Wide Wait." If the user interacts with and submits another form, the process begins again (*Steps 5–8*).

This model was originally designed for a web of hypertext documents—what some people call the "brochure web." As the web evolved into a full-scale applications platform, the model shown in Fig. 15.1 yielded "choppy" application performance. Every full-page refresh required users to re-establish their understanding of the full-page contents. Users began to demand a model that would yield the responsive feel of desktop applications.

Ajax Web Applications

Ajax applications add a layer between the client and the server to manage communication between the two (Fig. 15.2). When the user interacts with the page, the client creates an `XMLHttpRequest` object to manage a request (*Step 1*). The `XMLHttpRequest` object sends the request to the server (*Step 2*) and awaits the response. The requests are **asynchronous**, so the user can continue interacting with the application on the client-side while the server processes the earlier request concurrently. Other user interactions could result in additional requests to the server (*Steps 3* and *4*). Once the server responds to the original request (*Step 5*), the `XMLHttpRequest` object that issued the request calls a client-side function to process the data returned by the server. This function—known as a **callback function**—uses **partial page updates** (*Step 6*) to display the data in the existing web page *without reloading the entire page*. At the same time, the server may be responding to the second request (*Step 7*) and the client-side may be starting to do another partial page update (*Step 8*). The callback function updates only a designated part of the page. Such partial page updates help make web applications more responsive, making them feel more like desktop applications. The web application does not load a new page while the user interacts with it.

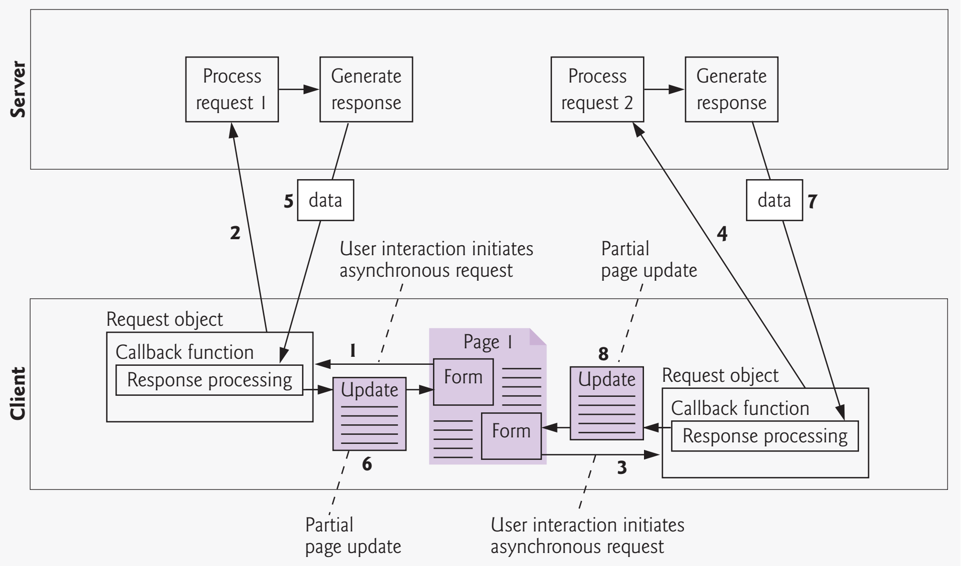


Fig. 15.2 | Ajax-enabled web application interacting with the server asynchronously.

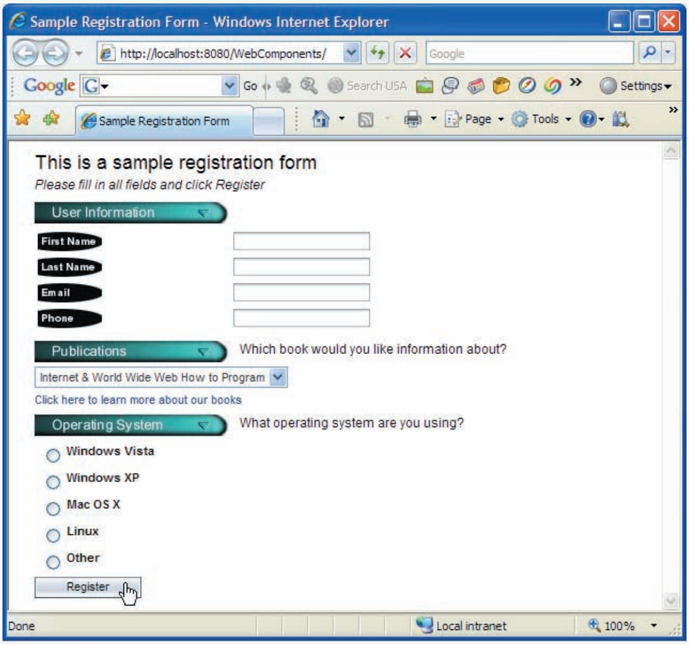
15.3 Rich Internet Applications (RIAs) with Ajax

Ajax improves the user experience by making interactive web applications more responsive. Consider a registration form with a number of fields (e.g., first name, last name e-mail address, telephone number, etc.) and a **Register** (or **Submit**) button that sends the entered data to the server. Usually each field has rules that the user's entries have to follow (e.g., valid e-mail address, valid telephone number, etc.).

When the user clicks **Register**, a classic XHTML form sends the server all of the data to be validated (Fig. 15.3). While the server is validating the data, the user cannot interact with the page. The server finds invalid data, generates a new page identifying the errors in the form and sends it back to the client—which renders the page in the browser. Once the user fixes the errors and clicks the **Register** button, the cycle repeats until no errors are found, then the data is stored on the server. The entire page reloads every time the user submits invalid data.

Ajax-enabled forms are more interactive. Rather than sending the entire form to be validated, entries are validated dynamically as the user enters data into the fields. For example, consider a website registration form that requires a unique e-mail address. When the user enters an e-mail address into the appropriate field, then moves to the next form field to continue entering data, an asynchronous request is sent to the server to validate the e-mail address. If the e-mail address is not unique, the server sends an error message that is displayed on the page informing the user of the problem (Fig. 15.4). By sending each entry asynchronously, the user can address each invalid entry quickly, versus making edits and resubmitting the entire form repeatedly until all entries are valid. Asynchronous

a) A sample registration form in which the user has not filled in the required fields, but attempts to submit the form anyway by clicking **Register**.



Sample Registration Form - Windows Internet Explorer

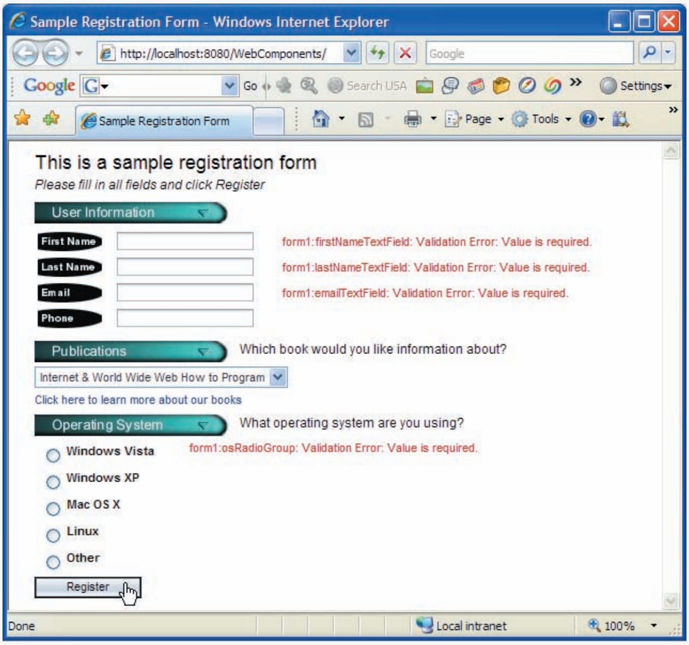
http://localhost:8080/WebComponents/

This is a sample registration form

Please fill in all fields and click Register

Fig. 15.3 | Classic XHTML form: User submits entire form to server, which validates the data entered (if any). Server responds indicating fields with invalid or missing data. (Part 1 of 2.)

b) The server responds by indicating all the form fields with missing or invalid data. The user must correct the problems and resubmit the entire form repeatedly until all errors are corrected.


Sample Registration Form - Windows Internet Explorer
http://localhost:8080/WebComponents/
Google
Google
Go
Search USA
Settings
Sample Registration Form
Page
Tools
Done
Local intranet
100%

This is a sample registration form

Please fill in all fields and click Register

User Information

First Name		form1:firstNameTextField: Validation Error: Value is required.
Last Name		form1:lastNameTextField: Validation Error: Value is required.
Email		form1:emailTextField: Validation Error: Value is required.
Phone		

Publications — Which book would you like information about?

Internet & World Wide Web How to Program

Click here to learn more about our books

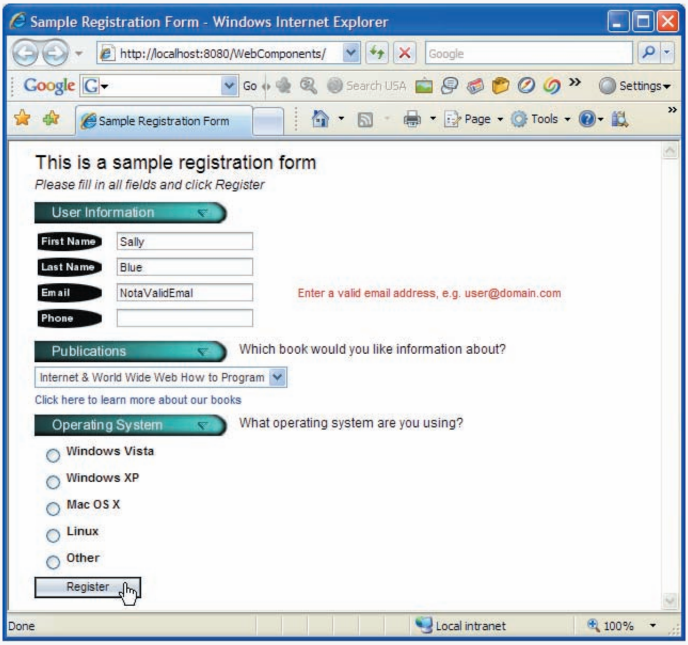
Operating System — What operating system are you using?

form1:osRadioGroup: Validation Error: Value is required.

- ○ Windows Vista
- ○ Windows XP
- ○ Mac OS X
- ○ Linux
- ○ Other

Register

Fig. 15.3 | Classic XHTML form: User submits entire form to server, which validates the data entered (if any). Server responds indicating fields with invalid or missing data. (Part 2 of 2.)


Sample Registration Form - Windows Internet Explorer
http://localhost:8080/WebComponents/
Google
Google
Go
Search USA
Settings
Sample Registration Form
Page
Tools
Done
Local intranet
100%

This is a sample registration form

Please fill in all fields and click Register

User Information

First Name	Sally	
Last Name	Blue	
Email	NotaValidEmal	Enter a valid email address, e.g. user@domain.com
Phone		

Publications — Which book would you like information about?

Internet & World Wide Web How to Program

Click here to learn more about our books

Operating System — What operating system are you using?

- ○ Windows Vista
- ○ Windows XP
- ○ Mac OS X
- ○ Linux
- ○ Other

Register

Fig. 15.4 | Ajax-enabled form shows errors asynchronously when user moves to another field.

requests could also be used to fill some fields based on previous fields (e.g., automatically filling in the “city” and “state” fields based on the zip code entered by the user).

15.4 History of Ajax

The term Ajax was coined by Jesse James Garrett of Adaptive Path in February 2005, when he was presenting the previously unnamed technology to a client. The technologies of Ajax (XHTML, JavaScript, CSS, the DOM and XML) have all existed for many years.

Asynchronous page updates can be traced back to earlier browsers. In the 1990s, Netscape’s LiveScript made it possible to include scripts in web pages (e.g., web forms) that could run on the client. LiveScript evolved into JavaScript. In 1998, Microsoft introduced the XMLHttpRequest object to create and manage asynchronous requests and responses. Popular applications like Flickr and Google’s Gmail use the XMLHttpRequest object to update pages dynamically. For example, Flickr uses the technology for its text editing, tagging and organizational features; Gmail continuously checks the server for new e-mail; and Google Maps allows you to drag a map in any direction, downloading the new areas on the map without reloading the entire page.

The name Ajax immediately caught on and brought attention to its component technologies. Ajax has become one of the hottest web-development technologies, enabling webtop applications to challenge the dominance of established desktop applications.

15.5 “Raw” Ajax Example Using the XMLHttpRequest Object

In this section, we use the XMLHttpRequest object to create and manage asynchronous requests. The XMLHttpRequest object (which resides on the client) is the layer between the client and the server that manages asynchronous requests in Ajax applications. This object is supported on most browsers, though they may implement it differently—a common issue in JavaScript programming. To initiate an asynchronous request (shown in Fig. 15.5), you create an instance of the XMLHttpRequest object, then use its open method to set up the request and its send method to initiate the request. We summarize the XMLHttpRequest properties and methods in Figs. 15.6–15.7.

Figure 15.5 presents an Ajax application in which the user interacts with the page by moving the mouse over book-cover images. We use the onmouseover and onmouseout events (discussed in Chapter 13) to trigger events when the user moves the mouse over and out of an image, respectively. The onmouseover event calls function getContent with the URL of the document containing the book’s description. The function makes this request asynchronously using an XMLHttpRequest object. When the XMLHttpRequest object receives the response, the book description is displayed below the book images. When the user moves the mouse out of the image, the onmouseout event calls function clearContent to clear the display box. These tasks are accomplished without reloading the page on the client. You can test-drive this example at test.deitel.com/examples/iw3htp4/ajax/fig15_05/SwitchContent.html.

Performance Tip 15.1

When an Ajax application requests a file from a server, such as an XHTML document or an image, the browser typically caches that file. Subsequent requests for the same file can load it from the browser’s cache rather than making the round trip to the server again.

Software Engineering Observation 15.1

For security purposes, the `XMLHttpRequest` *object doesn’t allow a web application to request resources from domain names other than the one that served the application. For this reason, the web application and its resources must reside on the same web server (this could be a web server on your local computer).This is commonly known as the* ***same origin policy (SOP)****. SOP aims to close a vulnerability called* ***cross-site scripting****, also known as* ***XSS****, which allows an attacker to compromise a website’s security by injecting a malicious script onto the page from another domain. To learn more about XSS visit* `en.wikipedia.org/wiki/XSS`*. To get content from another domain securely, you can implement a server-side proxy—an application on the web application’s web server—that can make requests to other servers on the web application’s behalf.*

Asynchronous Requests

The function `getContent` (lines 19–35) sends the asynchronous request. Line 24 creates the `XMLHttpRequest` object, which manages the asynchronous request. We store the object in the global variable `asyncRequest` (declared at line 16) so that it can be accessed anywhere in the script.

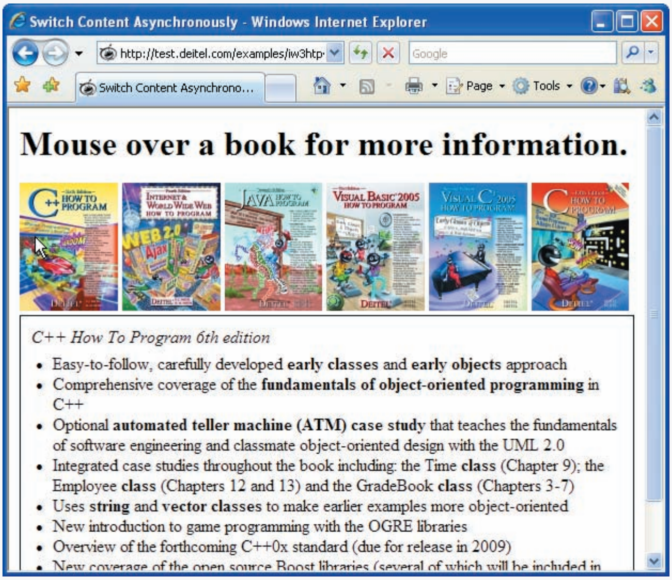
```
 1  <?xml version = "1.0" encoding = "utf-8"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. 15.5: SwitchContent.html -->
 6  <!-- Asynchronously display content without reloading the page. -->
 7  <html xmlns = "http://www.w3.org/1999/xhtml">
 8  <head>
 9     <style type="text/css">
10        .box { border: 1px solid black;
11               padding: 10px }
12     </style>
13     <title>Switch Content Asynchronously</title>
14     <script type = "text/javascript" language = "JavaScript">
15        <!--
16        var asyncRequest; // variable to hold XMLHttpRequest object
17
18        // set up and send the asynchronous request
19        function getContent( url )
20        {
21           // attempt to create the XMLHttpRequest and make the request
22           try
23           {
24              asyncRequest = new XMLHttpRequest(); // create request object
25
26              // register event handler
27              asyncRequest.onreadystatechange = stateChange;
28              asyncRequest.open( 'GET', url, true ); // prepare the request
29              asyncRequest.send( null ); // send the request
30           } // end try
```

Fig. 15.5 | Asynchronously display content without reloading the page. (Part 1 of 3.)

```
         catch ( exception )
         {
            alert( 'Request failed.' );
         } // end catch
      } // end function getContent

      // displays the response data on the page
      function stateChange()
      {
         if ( asyncRequest.readyState == 4 && asyncRequest.status == 200 )
         {
            document.getElementById( 'contentArea' ).innerHTML =
               asyncRequest.responseText; // places text in contentArea
         } // end if
      } // end function stateChange

      // clear the content of the box
      function clearContent()
      {
         document.getElementById( 'contentArea' ).innerHTML = '';
      } // end function clearContent
      // -->
   </script>
</head>
<body>
   <h1>Mouse over a book for more information.</h1>
   <img src =
      "http://test.deitel.com/examples/iw3htp4/ajax/thumbs/cpphtp6.jpg"
      onmouseover = 'getContent( "cpphtp6.html" )'
      onmouseout = 'clearContent()'/>
   <img src =
      "http://test.deitel.com/examples/iw3htp4/ajax/thumbs/iw3htp4.jpg"
      onmouseover = 'getContent( "iw3htp4.html" )'
      onmouseout = 'clearContent()'/>
   <img src =
      "http://test.deitel.com/examples/iw3htp4/ajax/thumbs/jhtp7.jpg"
      onmouseover = 'getContent( "jhtp7.html" )'
      onmouseout = 'clearContent()'/>
   <img src =
      "http://test.deitel.com/examples/iw3htp4/ajax/thumbs/vbhtp3.jpg"
      onmouseover = 'getContent( "vbhtp3.html" )'
      onmouseout = 'clearContent()'/>
   <img src =
      "http://test.deitel.com/examples/iw3htp4/ajax/thumbs/vcsharphtp2.jpg"
      onmouseover = 'getContent( "vcsharphtp2.html" )'
      onmouseout = 'clearContent()'/>
   <img src =
      "http://test.deitel.com/examples/iw3htp4/ajax/thumbs/chtp5.jpg"
      onmouseover = 'getContent( "chtp5.html" )'
      onmouseout = 'clearContent()'/>
   <div class = "box" id = "contentArea">&nbsp;</div>
</body>
</html>
```

Fig. 15.5 | Asynchronously display content without reloading the page. (Part 2 of 3.)

a) User hovers over *C++ How to Program* book cover image, causing an asynchronous request to the server to obtain the book's description. When the response is received, the application performs a partial page update to display the description.

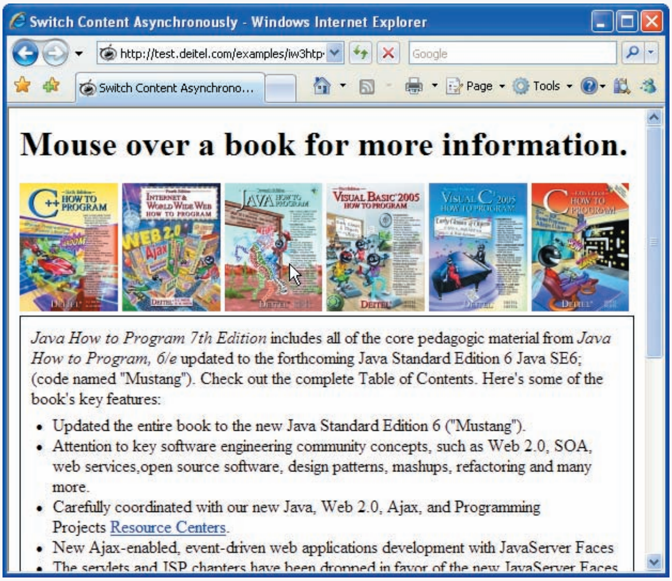

Switch Content Asynchronously - Windows Internet Explorer
http://test.deitel.com/examples/iw3htp
Google
Switch Content Asynchrono...
Page
Tools

Mouse over a book for more information.

C++ How To Program 6th edition

- Easy-to-follow, carefully developed **early classes** and **early objects** approach
- Comprehensive coverage of the **fundamentals of object-oriented programming** in C++
- Optional **automated teller machine (ATM) case study** that teaches the fundamentals of software engineering and classmate object-oriented design with the UML 2.0
- Integrated case studies throughout the book including: the Time **class** (Chapter 9); the Employee **class** (Chapters 12 and 13) and the GradeBook **class** (Chapters 3-7)
- Uses **string** and **vector classes** to make earlier examples more object-oriented
- New introduction to game programming with the OGRE libraries
- Overview of the forthcoming C++0x standard (due for release in 2009)
- New coverage of the open source Boost libraries (several of which will be included in

b) User hovers over *Java How to Program* book cover image, causing the process to repeat.


Switch Content Asynchronously - Windows Internet Explorer
http://test.deitel.com/examples/iw3htp
Google
Switch Content Asynchrono...
Page
Tools

Mouse over a book for more information.

Java How to Program 7th Edition includes all of the core pedagogic material from *Java How to Program, 6/e* updated to the forthcoming Java Standard Edition 6 Java SE6; (code named "Mustang"). Check out the complete Table of Contents. Here's some of the book's key features:

- Updated the entire book to the new Java Standard Edition 6 ("Mustang").
- Attention to key software engineering community concepts, such as Web 2.0, SOA, web services,open source software, design patterns, mashups, refactoring and many more.
- Carefully coordinated with our new Java, Web 2.0, Ajax, and Programming Projects Resource Centers.
- New Ajax-enabled, event-driven web applications development with JavaServer Faces
- The servlets and JSP chapters have been dropped in favor of the new JavaServer Faces

Fig. 15.5 | Asynchronously display content without reloading the page. (Part 3 of 3.)

Line 28 calls the `XMLHttpRequest` `open` method to prepare an asynchronous `GET` request. In this example, the `url` parameter specifies the address of an HTML document containing the description of a particular book. When the third argument is `true`, the request is asynchronous. The URL is passed to function getContent in response to the onmouseover event for each image. Line 29 sends the asynchronous request to the server by calling XMLHttpRequest send method. The argument null indicates that this request is not submitting data in the body of the request.

Exception Handling

Lines 22–34 introduce **exception handling**. An **exception** is an indication of a problem that occurs during a program's execution. The name "exception" implies that the problem occurs infrequently—if the "rule" is that a statement normally executes correctly, then the "exception to the rule" is that a problem occurs. Exception handling enables you to create applications that can resolve (or handle) exceptions—in some cases allowing a program to continue executing as if no problem had been encountered.

Lines 22–30 contain a **try block**, which encloses the code that might cause an exception and the code that should not execute if an exception occurs (i.e., if an exception occurs in a statement of the try block, the remaining code in the try block is skipped). A try block consists of the keyword try followed by a block of code enclosed in curly braces ({}). If there is a problem sending the request—e.g., if a user tries to access the page using an older browser that does not support XMLHttpRequest—the try block terminates immediately and a **catch block** (also called a **catch clause** or **exception handler**) catches (i.e., receives) and handles an exception. The catch block (lines 31–34) begins with the keyword catch and is followed by a parameter in parentheses (called the exception parameter) and a block of code enclosed in curly braces. The exception parameter's name (exception in this example) enables the catch block to interact with a caught exception object (for example, to obtain the name of the exception or an exception-specific error message via the exception object's name and message properties). In this case, we simply display our own error message 'Request Failed' and terminate the getContent function. The request can fail because a user accesses the web page with an older browser or the content that is being requested is located on a different domain.

Callback Functions

The stateChange function (lines 38–45) is the callback function that is called when the client receives the response data. Line 27 registers function stateChange as the event handler for the XMLHttpRequest object's onreadystatechange event. Whenever the request makes progress, the XMLHttpRequest calls the onreadystatechange event handler. This progress is monitored by the readyState property, which has a value from 0 to 4. The value 0 indicates that the request is not initialized and the value 4 indicates that the request is complete—all the values for this property are summarized in Fig. 15.6. If the request completes successfully (line 40), lines 42–43 use the XMLHttpRequest object's responseText property to obtain the response data and place it in the div element named contentArea (defined at line 81). We use the DOM's getElementById method to get this div element, and use the element's innerHTML property to place the content in the div.

XMLHttpRequest Object Properties and Methods

Figures 15.6 and 15.7 summarize some of the XMLHttpRequest object's properties and methods, respectively. The properties are crucial to interacting with asynchronous requests. The methods initialize, configure and send asynchronous requests.

Property	Description
onreadystatechange	Stores the callback function—the event handler that gets called when the server responds.
readyState	Keeps track of the request's progress. It is usually used in the callback function to determine when the code that processes the response should be launched. The readyState value 0 signifies that the request is uninitialized; 1 signifies that the request is loading; 2 signifies that the request has been loaded; 3 signifies that data is actively being sent from the server; and 4 signifies that the request has been completed.
responseText	Text that is returned to the client by the server.
responseXML	If the server's response is in XML format, this property contains the XML document; otherwise, it is empty. It can be used like a document object in JavaScript, which makes it useful for receiving complex data (e.g. populating a table).
status	HTTP status code of the request. A status of 200 means that request was successful. A status of 404 means that the requested resource was not found. A status of 500 denotes that there was an error while the server was proccessing the request.
statusText	Additional information on the request's status. It is often used to display the error to the user when the request fails.

Fig. 15.6 | XMLHttpRequest object properties.

Method	Description
open	Initializes the request and has two mandatory parameters—method and URL. The method parameter specifies the purpose of the request—typically GET if the request is to take data from the server or POST if the request will contain a body in addition to the headers. The URL parameter specifies the address of the file on the server that will generate the response. A third optional boolean parameter specifies whether the request is asynchronous—it's set to true by default.
send	Sends the request to the sever. It has one optional parameter, data, which specifies the data to be POSTed to the server—it's set to null by default.
setRequestHeader	Alters the header of the request. The two parameters specify the header and its new value. It is often used to set the content-type field.

Fig. 15.7 | XMLHttpRequest object methods. (Part 1 of 2.)

Method	Description
`getResponseHeader`	Returns the header data that precedes the response body. It takes one parameter, the name of the header to retrieve. This call is often used to determine the response's type, to parse the response correctly.
`getAllResponseHeaders`	Returns an array that contains all the headers that precede the response body.
`abort`	Cancels the current request.

Fig. 15.7 | `XMLHttpRequest` object methods. (Part 2 of 2.)

15.6 Using XML and the DOM

When passing structured data between the server and the client, Ajax applications often use XML because it is easy to generate and parse. When the `XMLHttpRequest` object receives XML data, it parses and stores the data as an XML DOM object in the `responseXML` property. The example in Fig. 15.8 asynchronously requests from a server XML documents containing URLs of book-cover images, then displays the images in an HTML table. The code that configures the asynchronous request is the same as in Fig. 15.5. You can test-drive this application at `test.deitel.com/examples/iw3htp4/ajax/fig15_08/PullImagesOntoPage.html` (the book-cover images will be easier to see on the screen).

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 15.8: PullImagesOntoPage.html -->
<!-- Image catalog that uses Ajax to request XML data asynchronously. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
<title> Pulling Images onto the Page </title>
<style type = "text/css">
   td { padding: 4px }
   img { border: 1px solid black }
</style>
<script type = "text/javascript" language = "Javascript">
   var asyncRequest; // variable to hold XMLHttpRequest object

   // set up and send the asynchronous request to the XML file
   function getImages( url )
   {
      // attempt to create the XMLHttpRequest and make the request
      try
      {
         asyncRequest = new XMLHttpRequest(); // create request object

```

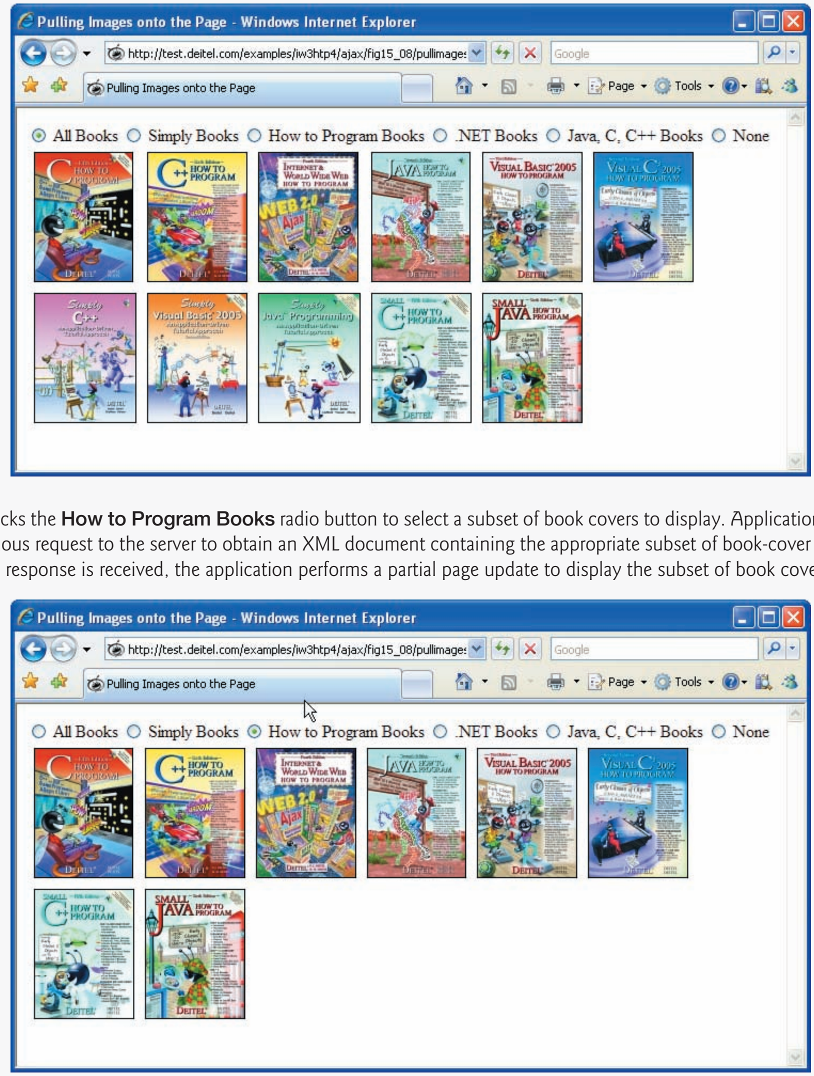
Fig. 15.8 | Image catalog that uses Ajax to request XML data asynchronously. (Part 1 of 4.)

```
            // register event handler
            asyncRequest.onreadystatechange = processResponse;
            asyncRequest.open( 'GET', url, true ); // prepare the request
            asyncRequest.send( null ); // send the request
         } // end try
         catch ( exception )
         {
            alert( 'Request Failed' );
         } // end catch
      } // end function getImages

      // parses the XML response; dynamically creates a table using DOM and
      // populates it with the response data; displays the table on the page
      function processResponse()
      {
         // if request completed successfully and responseXML is non-null
         if ( asyncRequest.readyState == 4 && asyncRequest.status == 200 &&
            asyncRequest.responseXML )
         {
            clearTable(); // prepare to display a new set of images

            // get the covers from the responseXML
            var covers = asyncRequest.responseXML.getElementsByTagName(
               "cover" )

            // get base URL for the images
            var baseUrl = asyncRequest.responseXML.getElementsByTagName(
               "baseurl" ).item( 0 ).firstChild.nodeValue;

            // get the placeholder div element named covers
            var output = document.getElementById( "covers" );

            // create a table to display the images
            var imageTable = document.createElement( 'table' );

            // create the table's body
            var tableBody = document.createElement( 'tbody' );

            var rowCount = 0; // tracks number of images in current row
            var imageRow = document.createElement( "tr" ); // create row

            // place images in row
            for ( var i = 0; i < covers.length; i++ )
            {
               var cover = covers.item( i ); // get a cover from covers array

               // get the image filename
               var image = cover.getElementsByTagName( "image" ).
                  item( 0 ).firstChild.nodeValue;

               // create table cell and img element to display the image
               var imageCell = document.createElement( "td" );
```

Fig. 15.8 | Image catalog that uses Ajax to request XML data asynchronously. (Part 2 of 4.)

```
            var imageTag = document.createElement( "img" );

            // set img element's src attribute
            imageTag.setAttribute( "src", baseUrl + escape( image ) );
            imageCell.appendChild( imageTag ); // place img in cell
            imageRow.appendChild( imageCell ); // place cell in row
            rowCount++; // increment number of images in row

            // if there are 6 images in the row, append the row to
            // table and start a new row
            if ( rowCount == 6 && i + 1 < covers.length )
            {
               tableBody.appendChild( imageRow );
               imageRow = document.createElement( "tr" );
               rowCount = 0;
            } // end if statement
         } // end for statement

         tableBody.appendChild( imageRow ); // append row to table body
         imageTable.appendChild( tableBody ); // append body to table
         output.appendChild( imageTable ); // append table to covers div
      } // end if
   } // end function processResponse

   // deletes the data in the table.
   function clearTable()
   {
      document.getElementById( "covers" ).innerHTML = '';
   }// end function clearTable
   </script>
</head>
<body>
   <input type = "radio" checked = "unchecked" name ="Books" value = "all"
      onclick = 'getImages( "all.xml" )'/> All Books
   <input type = "radio" checked = "unchecked"
      name = "Books" value = "simply"
      onclick = 'getImages( "simply.xml" )'/>  Simply Books
   <input type = "radio" checked = "unchecked"
      name = "Books" value = "howto"
      onclick = 'getImages( "howto.xml" )'/> How to Program Books
   <input type = "radio" checked = "unchecked"
      name = "Books" value = "dotnet"
      onclick = 'getImages( "dotnet.xml" )'/> .NET Books
   <input type = "radio" checked = "unchecked"
      name = "Books" value = "javaccpp"
      onclick = 'getImages( "javaccpp.xml" )'/> Java, C, C++ Books
   <input type = "radio" checked = "checked" name = "Books" value = "none"
      onclick = 'clearTable()'/> None
   <br/>
   <div id = "covers"></div>
</body>
</html>
```

Fig. 15.8 | Image catalog that uses Ajax to request XML data asynchronously. (Part 3 of 4.)



a) User clicks the **All Books** radio button to display all the book covers. The application sends an asynchronous request to the server to obtain an XML document containing the list of book-cover filenames. When the response is received, the application performs a partial page update to display the set of book covers.

Pulling Images onto the Page - Windows Internet Explorer

http://test.deitel.com/examples/iw3htp4/ajax/fig15_08/pullimage:

● All Books ○ Simply Books ○ How to Program Books ○ .NET Books ○ Java, C, C++ Books ○ None

b) User clicks the **How to Program Books** radio button to select a subset of book covers to display. Application sends an asynchronous request to the server to obtain an XML document containing the appropriate subset of book-cover filenames. When the response is received, the application performs a partial page update to display the subset of book covers.

Pulling Images onto the Page - Windows Internet Explorer

http://test.deitel.com/examples/iw3htp4/ajax/fig15_08/pullimage:

○ All Books ○ Simply Books ● How to Program Books ○ .NET Books ○ Java, C, C++ Books ○ None

Fig. 15.8 | Image catalog that uses Ajax to request XML data asynchronously. (Part 4 of 4.)

When the `XMLHttpRequest` object receives the response, it invokes the callback function `processResponse` (lines 38–99). We use `XMLHttpRequest` object's `responseXML` property to access the XML returned by the server. Lines 41–42 check that the request was successful, and that the `responseXML` property is not empty. The XML file that we requested includes a `baseURL` node that contains the address of the image directory and a collection of `cover` nodes that contain image filenames. `responseXML` is a document object, so we can extract data from it using the XML DOM functions. Lines 47–52 use the DOM's method **getElementsByTagName** to extract all the image filenames from cover nodes and the URL of the directory from the baseURL node. Since the baseURL has no child nodes, we use item(0).firstChild.nodeValue to obtain the directory's address and store it in variable baseURL. The image filenames are stored in the covers array.

As in Fig. 15.5 we have a placeholder div element (line 126) to specify where the image table will be displayed on the page. Line 55 stores the div in variable output, so we can fill it with content later in the program.

Lines 58–93 generate an XHTML table dynamically, using the createElement, setAttribute and appendChild DOM methods. Method createElement creates an XHTML element of the specified type. Method **setAttribute** adds or changes an attribute of an XHTML element. Method appendChild inserts one XHTML element into another. Lines 58 and 61 create the table and tbody elements, respectively. We restrict each row to no more than six images, which we track with variable rowCount variable. Each iteration of the for statement (lines 67–93) obtains the filename of the image to be inserted (lines 69–73), creates a table cell element where the image will be inserted (line 76) and creates an <img> element (line 77). Line 80 sets the image's src attribute to the image's URL, which we build by concatenating the filename to the base URL of the XHTML document. Lines 81–82 insert the <img> element into the cell and the cell into the table row. When the row has six cells, it is inserted into the table and a new row is created (lines 87–92). Once all the rows have been inserted into the table, the table is inserted into the placeholder element covers that is referenced by variable output (line 97). This element is located on the bottom of the web page.

Function clearTable (lines 102–105) is called to clear images when the user switches radio buttons. The text is cleared by setting the innerHTML property of the placeholder element to the empty string.

15.7 Creating a Full-Scale Ajax-Enabled Application

Our next example demonstrates additional Ajax capabilities. The web application interacts with a web service to obtain data and to modify data in a server-side database. The web application and server communicate with a data format called JSON (JavaScript Object Notation). In addition, the application demonstrates server-side validation that occurs in parallel with the user interacting with the web application. You can test the application at test.deitel.com/examples/iw3htp4/ajax/fig15_09_10/AddressBook.html.

Using JSON

JSON (JavaScript Object Notation)—a simple way to represent JavaScript objects as strings—is an alternative way (to XML) for passing data between the client and the server. Each object in JSON is represented as a list of property names and values contained in curly braces, in the following format:

```
{ "propertyName1" : value1, "propertyName2": value2 }
```

Arrays are represented in JSON with square brackets in the following format:

```
[ value1, value2, value3 ]
```

Each value can be a string, a number, a JSON representation of an object, true, false or null. You can convert JSON strings into JavaScript objects with JavaScript's eval function. To evaluate a JSON string properly, a left parenthesis should be placed at the beginning of the string and a right parenthesis at the end of the string before the string is passed to the eval function.

The eval function creates a potential security risk—it executes any embedded JavaScript code in its string argument, possibly allowing a harmful script to be injected into JSON. A more secure way to process JSON is to use a JSON parser. In our examples, we use the open source parser from www.json.org/js.html. When you download its JavaScript file, place it in the same folder as your application. Then, link the json.js file into your XHTML file with the following statement in the head section:

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

You can now call function parseJSON on a JSON string to convert it to a JavaScript object.

JSON strings are easier to create and parse than XML, and require fewer bytes. For these reasons, JSON is commonly used to communicate in client/server interaction. For more information on JSON, visit our JSON Resource Center at www.deitel.com/json.

Rich Functionality

The previous examples in this chapter requested data from static files on the server. The example in Fig. 15.9 is an address-book application that communicates with a server-side application. The application uses server-side processing to give the page the functionality and usability of a desktop application. We use JSON to encode server-side responses and to create objects on the fly.

Initially the address book loads a list of entries, each containing a first and last name (Fig. 15.9(a)). Each time the user clicks a name, the address book uses Ajax functionality to load the person's address from the server and expand the entry *without reloading the page* (Fig. 15.9(b))—and it does this *in parallel* with allowing the user to click other names. The application allows the user to search the address book by typing a last name. As the user enters each keystroke, the application asynchronously displays the list of names in which the last name starts with the characters the user has entered so far (Fig. 15.9(c), Fig. 15.9 (d) and Fig. 15.9(e))—a popular feature called **type ahead**.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 15.9 addressbook.html -->
<!-- Ajax enabled address book application. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
   <title>Address Book</title>
   <link rel = "stylesheet" type = "text/css" href = "address.css" />
   <script type = "text/javascript" src = "json.js"></script>
   <script type = "text/javascript">
      <!--
      // URL of the web service
      var webServiceUrl = '/AddressBookWebService/AddressService.asmx';
```

Fig. 15.9 | Ajax-enabled address-book application. (Part 1 of 10.)

```

      var phoneValid = false; // indicates if the telephone is valid
      var zipValid = false; //indicates if the zip code is valid

      // get a list of names from the server and display them
      function showAddressBook()
      {
         // hide the "addEntry" form and show the address book
         document.getElementById( 'addEntry' ).style.display = 'none';
         document.getElementById( 'addressBook' ).style.display = 'block';

         var params = "[]"; // create an empty object
         callWebService( 'getAllNames', params, parseData );
      } // end function showAddressBook

      // send the asynchronous request to the web service
      function callWebService( method, paramString, callBack )
      {
         // build request URL string
         var requestUrl = webServiceUrl + "/" + method;
         var params = paramString.parseJSON();

         // build the parameter string to add to the url
         for ( var i = 0; i < params.length; i++ )
         {
            // checks whether it is the first parameter and builds
            // the parameter string accordingly
            if ( i == 0 )
               requestUrl = requestUrl + "?" + params[ i ].param +
                  "=" + params[ i ].value; // add first parameter to url
            else
               requestUrl = requestUrl + "&" + params[ i ].param +
                  "=" + params[ i ].value; // add other parameters to url
         } // end for

         // attempt to send the asynchronous request
         try
         {
            var asyncRequest = new XMLHttpRequest(); // create request

            // set up callback function and store it
            asyncRequest.onreadystatechange = function()
            {
               callBack( asyncRequest );
            }; // end anonymous function

            // send the asynchronous request
            asyncRequest.open( 'GET', requestUrl, true );
            asyncRequest.setRequestHeader("Accept",
               "application/json; charset=utf-8" );
            asyncRequest.send(); // send request
         } // end try
```

Fig. 15.9 | Ajax-enabled address-book application. (Part 2 of 10.)

```
         catch ( exception )
         {
            alert ( 'Request Failed' );
         } // end catch
      } // end function callWebService

      // parse JSON data and display it on the page
      function parseData( asyncRequest )
      {
         // if request has completed successfully process the response
         if ( asyncRequest.readyState == 4 && asyncRequest.status == 200 )
         {
            // convert the JSON string to an Object
            var data = asyncRequest.responseText.parseJSON();
            displayNames( data ); // display data on the page
         } // end if
      } // end function parseData

      // use the DOM to display the retrieved address book entries
      function displayNames( data )
      {
         // get the placeholder element from the page
         var listBox = document.getElementById( 'Names' );
         listBox.innerHTML = ''; // clear the names on the page

         // iterate over retrieved entries and display them on the page
         for ( var i = 0; i < data.length; i++ )
         {
            // dynamically create a div element for each entry
            // and a fieldset element to place it in
            var entry = document.createElement( 'div' );
            var field = document.createElement( 'fieldset' );
            entry.onclick = handleOnClick; // set onclick event handler
            entry.id = i; // set the id
            entry.innerHTML = data[ i ].First + ' ' + data[ i ].Last;
            field.appendChild( entry ); // insert entry into the field
            listBox.appendChild( field ); // display the field
          } // end for
      } // end function displayAll

      // event handler for entry's onclick event
      function handleOnClick()
      {
         // call getAddress with the element's content as a parameter
         getAddress( eval( 'this' ), eval( 'this.innerHTML' ) );
      } // end function handleOnClick

      // search the address book for input
      // and display the results on the page
      function search( input )
      {
         // get the placeholder element and delete its content
         var listBox = document.getElementById( 'Names' );
```

Fig. 15.9 | Ajax-enabled address-book application. (Part 3 of 10.)

```
         listBox.innerHTML = ''; // clear the display box

         // if no search string is specified all the names are displayed
         if ( input == "" ) // if no search value specified
         {
            showAddressBook(); // Load the entire address book
         } // end if
         else
         {
            var params = '[{"param": "input", "value": "' + input + '"}]';
            callWebService( "search",  params , parseData );
         } // end else
      } // end function search

      // Get address data for a specific entry
      function getAddress( entry, name )
      {
         // find the address in the JSON data using the element's id
         // and display it on the page
         var firstLast = name.split(" "); // convert string to array
         var requestUrl = webServiceUrl + "/getAddress?first="
            + firstLast[ 0 ] + "&last=" + firstLast[ 1 ];

         // attempt to send an asynchronous request
         try
         {
            // create request object
            var asyncRequest = new XMLHttpRequest();

            // create a callback function with 2 parameters
            asyncRequest.onreadystatechange = function()
            {
               displayAddress( entry, asyncRequest );
            }; // end anonymous function

            asyncRequest.open( 'GET', requestUrl, true );
            asyncRequest.setRequestHeader("Accept",
            "application/json; charset=utf-8"); // set response datatype
            asyncRequest.send(); // send request
         } // end try
         catch ( exception )
         {
            alert ( 'Request Failed.' );
         } // end catch
      } // end function getAddress

      // clear the entry's data.
      function displayAddress( entry, asyncRequest )
      {
         // if request has completed successfully, process the response
         if ( asyncRequest.readyState == 4 && asyncRequest.status == 200 )
         {
```

Fig. 15.9 | Ajax-enabled address-book application. (Part 4 of 10.)

```
            // convert the JSON string to an object
            var data = asyncRequest.responseText.parseJSON();
            var name = entry.innerHTML // save the name string
            entry.innerHTML = name + '<br/>' + data.Street +
               '<br/>' + data.City + ', ' + data.State
               + ', ' + data.Zip + '<br/>' + data.Telephone;

            // clicking on the entry removes the address
            entry.onclick = function()
            {
               clearField( entry, name );
            }; // end anonymous function

          } // end if
      } // end function displayAddress

      // clear the entry's data
      function clearField( entry, name )
      {
         entry.innerHTML = name; // set the entry to display only the name
         entry.onclick = function() // set onclick event
         {
            getAddress( entry, name ); // retrieve address and display it
         }; // end function
      } // end function clearField

      // display the form that allows the user to enter more data
      function addEntry()
      {
         document.getElementById( 'addressBook' ).style.display = 'none';
         document.getElementById( 'addEntry' ).style.display = 'block';
      } // end function addEntry

      // send the zip code to be validated and to generate city and state
      function validateZip( zip )
      {
         // build parameter array
         var params = '[{"param": "zip", "value": "' + zip + '"}]';
         callWebService ( "validateZip", params, showCityState );
      } // end function validateZip

      // get city and state that were generated using the zip code
      // and display them on the page
      function showCityState( asyncRequest )
      {
         // display message while request is being processed
         document.getElementById( 'validateZip' ).
            innerHTML = "Checking zip...";

         // if request has completed successfully, process the response
         if ( asyncRequest.readyState == 4 )
         {
```

Fig. 15.9 | Ajax-enabled address-book application. (Part 5 of 10.)

```
            if ( asyncRequest.status == 200 )
            {
               // convert the JSON string to an object
               var data = asyncRequest.responseText.parseJSON();

               // update zip code validity tracker and show city and state
               if ( data.Validity == 'Valid' )
               {
                  zipValid = true; // update validity tracker

                  // display city and state
                  document.getElementById( 'validateZip' ).innerHTML = '';
                  document.getElementById( 'city' ).innerHTML = data.City;
                  document.getElementById( 'state' ).
                     innerHTML = data.State;
               } // end if
               else
               {
                  zipValid = false; // update validity tracker
                  document.getElementById( 'validateZip' ).
                     innerHTML = data.ErrorText; // display the error

                  // clear city and state values if they exist
                  document.getElementById( 'city' ).innerHTML = '';
                  document.getElementById( 'state' ).innerHTML = '';
               } // end else
            } // end if
            else if ( asyncRequest.status == 500 )
            {
               document.getElementById( 'validateZip' ).
                  innerHTML = 'Zip validation service not avaliable';
            } // end else if
         } // end if
      } // end function showCityState

      // send the telephone number to the server to validate format
      function validatePhone( phone )
      {
         var params = '[{ "param": "tel", "value": "' + phone + '"}]';
         callWebService( "validateTel", params, showPhoneError );
      } // end function validatePhone

      // show whether the telephone number has correct format
      function showPhoneError( asyncRequest )
      {
         // if request has completed successfully, process the response
         if ( asyncRequest.readyState == 4 && asyncRequest.status == 200 )
         {
            // convert the JSON string to an object
            var data = asyncRequest.responseText.parseJSON();

            if ( data.ErrorText != "Valid Telephone Format" )
            {
```

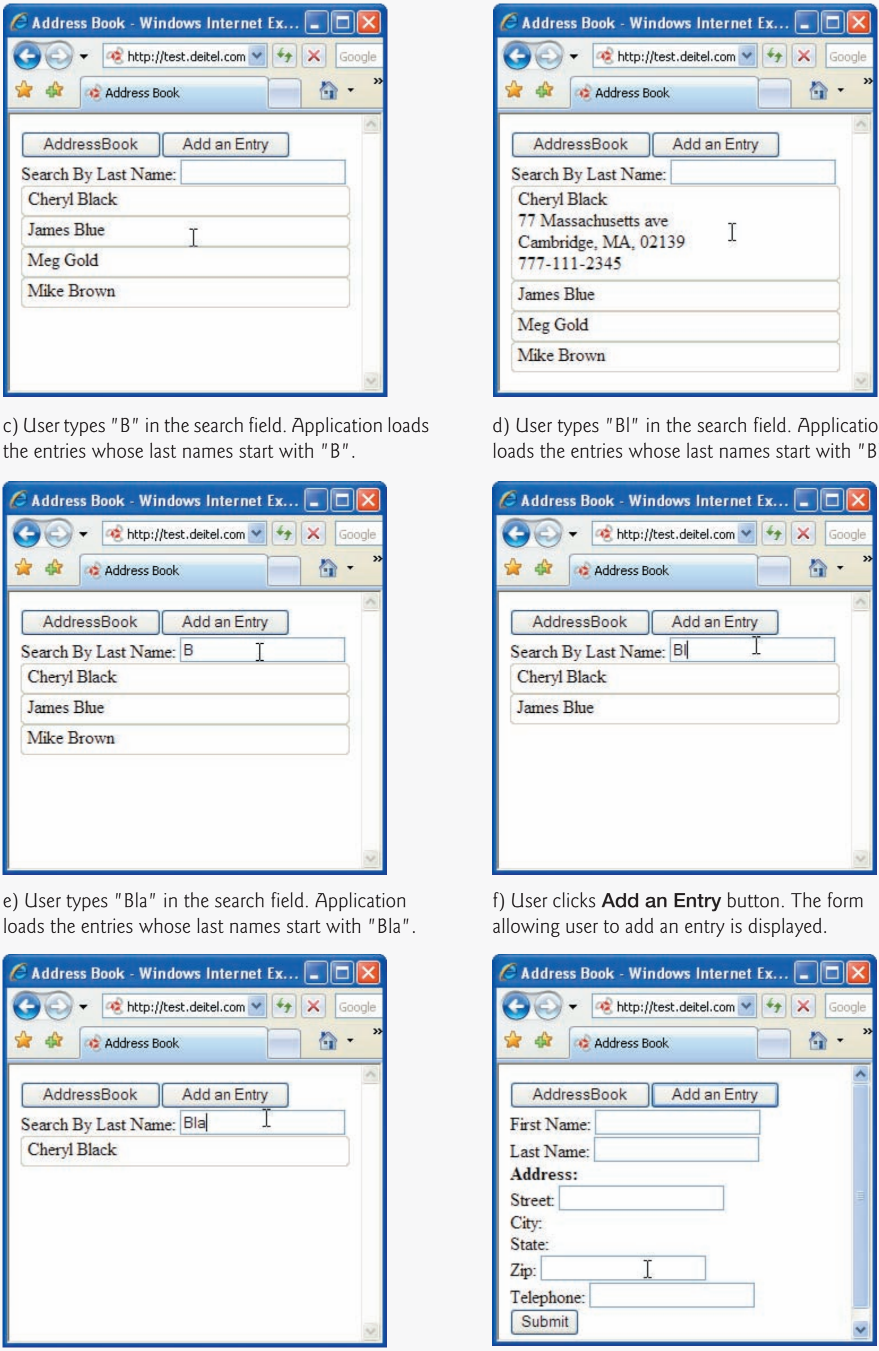
Fig. 15.9 | Ajax-enabled address-book application. (Part 6 of 10.)

```
         phoneValid = false; // update validity tracker
      } // end if
      else
      {
         phoneValid = true; // update validity tracker
      } // end else

      document.getElementById( 'validatePhone' ).
         innerHTML = data.ErrorText; // display the error
    } // end if
} // end function showPhoneError

// enter the user's data into the database
function saveForm()
{
   // retrieve the data from the form
   var first = document.getElementById( 'first' ).value;
   var last = document.getElementById( 'last' ).value;
   var street = document.getElementById( 'street' ).value;
   var city = document.getElementById( 'city' ).innerHTML;
   var state = document.getElementById( 'state' ).innerHTML;
   var zip = document.getElementById( 'zip' ).value;
   var phone = document.getElementById( 'phone' ).value;

   // check if data is valid
   if ( !zipValid || !phoneValid  )
   {
      // display error message
      document.getElementById( 'success' ).innerHTML =
         'Invalid data entered. Check form for more information';
   } // end if
   else if ( ( first == "" ) || ( last == "" ) )
   {
      // display error message
      document.getElementById( 'success').innerHTML =
         'First Name and Last Name must have a value.';
   } // end if
   else
   {
      // hide the form and show the addressbook
      document.getElementById( 'addEntry' )
         .style.display = 'none';
      document.getElementById( 'addressBook' ).
         style.display = 'block';

      // build the parameter to include in the web service URL
      params = '[{"param": "first", "value": "' + first +
         '"}, { "param": "last", "value": "' + last +
         '"}, { "param": "street", "value": "'+ street +
         '"}, { "param": "city", "value": "' +  city +
         '"}, { "param":  "state", "value:": "' + state +
         '"}, { "param": "zip", "value": "' + zip +
         '"}, { "param": "tel", "value": "' + phone + '"}]';
```

Fig. 15.9 | Ajax-enabled address-book application. (Part 7 of 10.)

```

            // call the web service to insert data into the database
            callWebService( "addEntry", params, parseData );
         } // end else
      } // end function saveForm
      //-->
   </script>
</head>
<body onload = "showAddressBook()">
   <div>
      <input type = "button" value = "Address Book"
         onclick = "showAddressBook()"/>
      <input type = "button" value = "Add an Entry"
         onclick = "addEntry()"/>
   </div>
   <div id = "addressBook" style = "display : block;">
       Search By Last Name:
      <input onkeyup = "search( this.value )"/>
      <br/>
      <div id = "Names">
      </div>
   </div>
   <div id = "addEntry" style = "display : none">
      First Name: <input id = 'first'/>
      <br/>
      Last Name: <input id = 'last'/>
      <br/>
      <strong> Address: </strong>
      <br/>
      Street: <input id = 'street'/>
      <br/>
      City: <span id = "city" class = "validator"></span>
      <br/>
      State: <span id = "state" class = "validator"></span>
      <br/>
      Zip: <input id = 'zip' onblur = 'validateZip( this.value )'/>
      <span id = "validateZip" class = "validator">
      </span>
      <br/>
      Telephone:<input id = 'phone'
         onblur = 'validatePhone( this.value )'/>
      <span id = "validatePhone" class = "validator">
      </span>
      <br/>
      <input type = "button" value = "Submit"
         onclick = "saveForm()" />
      <br/>
      <div id = "success" class = "validator">
      </div>
   </div>
</body>
</html>
```

Fig. 15.9 | Ajax-enabled address-book application. (Part 8 of 10.)



a) Page is loaded. All the entries are displayed.

Cheryl Black
James Blue
Meg Gold
Mike Brown

b) User clicks on an entry. The entry expands, showing the address and the telephone.

Cheryl Black
77 Massachusetts ave
Cambridge, MA, 02139
777-111-2345
James Blue
Meg Gold
Mike Brown

c) User types "B" in the search field. Application loads the entries whose last names start with "B".

Search By Last Name: B
Cheryl Black
James Blue
Mike Brown

d) User types "Bl" in the search field. Application loads the entries whose last names start with "Bl".

Search By Last Name: Bl
Cheryl Black
James Blue

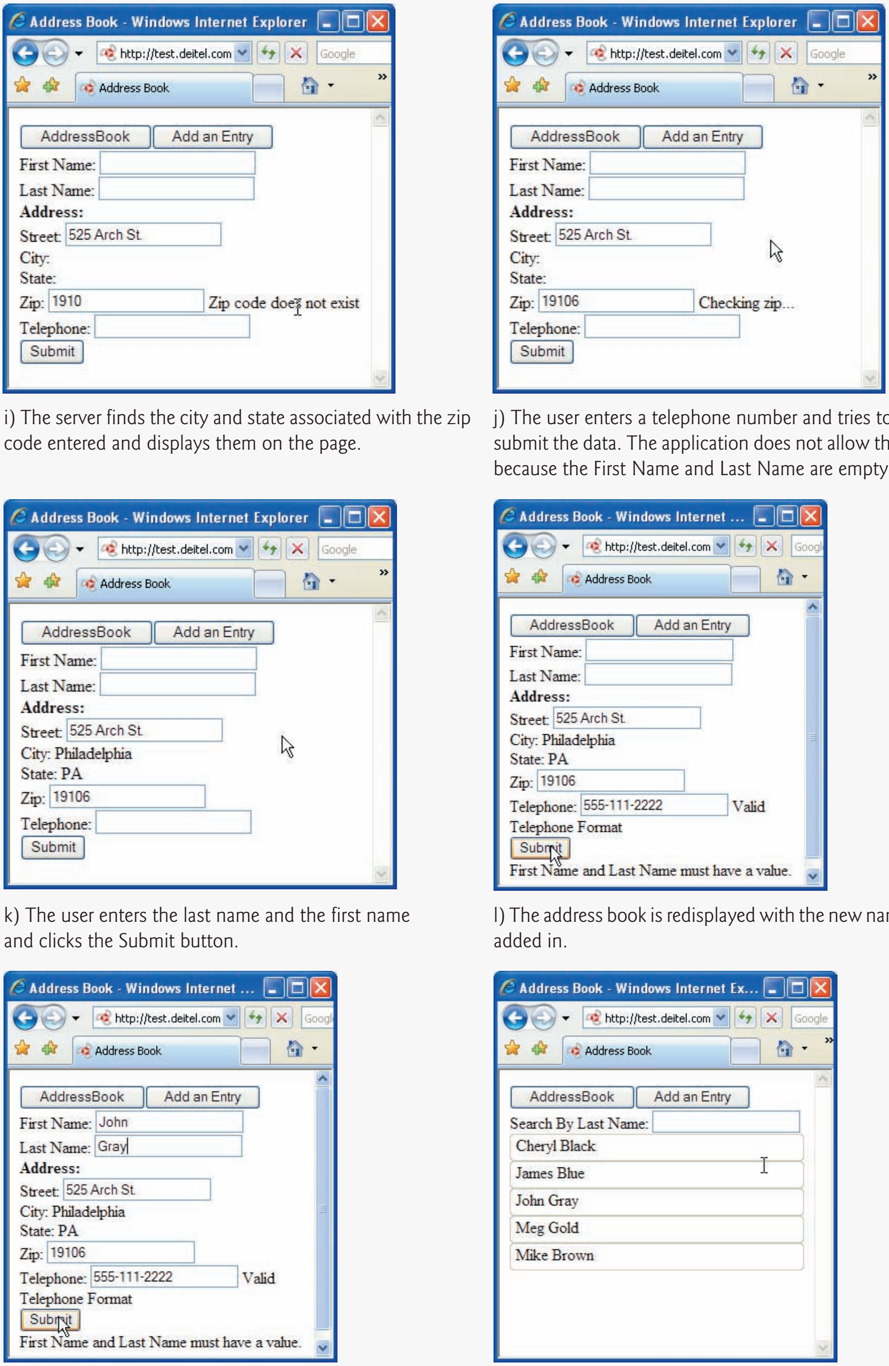
e) User types "Bla" in the search field. Application loads the entries whose last names start with "Bla".

Search By Last Name: Bla
Cheryl Black

f) User clicks **Add an Entry** button. The form allowing user to add an entry is displayed.

First Name:
Last Name:
Address:
Street:
City:
State:
Zip:
Telephone:
Submit

Fig. 15.9 | Ajax-enabled address-book application. (Part 9 of 10.)



g) User types in a nonexistent zip code. An error is displayed.

Address Book - Windows Internet Explorer

http://test.deitel.com

AddressBook | Add an Entry

First Name:

Last Name:

Address:

Street: 525 Arch St

City:

State:

Zip: 1910 Zip code does not exist

Telephone:

Submit

h) User enters a valid zip code. While the server processes it, **Checking Zip...** is displayed on the page.

Address Book - Windows Internet Explorer

http://test.deitel.com

AddressBook | Add an Entry

First Name:

Last Name:

Address:

Street: 525 Arch St

City:

State:

Zip: 19106 Checking zip...

Telephone:

Submit

i) The server finds the city and state associated with the zip code entered and displays them on the page.

Address Book - Windows Internet Explorer

http://test.deitel.com

AddressBook | Add an Entry

First Name:

Last Name:

Address:

Street: 525 Arch St

City: Philadelphia

State: PA

Zip: 19106

Telephone:

Submit

j) The user enters a telephone number and tries to submit the data. The application does not allow this, because the First Name and Last Name are empty.

Address Book - Windows Internet ...

http://test.deitel.com

AddressBook | Add an Entry

First Name:

Last Name:

Address:

Street: 525 Arch St

City: Philadelphia

State: PA

Zip: 19106

Telephone: 555-111-2222 Valid Telephone Format

Submit

First Name and Last Name must have a value.

k) The user enters the last name and the first name and clicks the Submit button.

Address Book - Windows Internet ...

http://test.deitel.com

AddressBook | Add an Entry

First Name: John

Last Name: Gray

Address:

Street: 525 Arch St

City: Philadelphia

State: PA

Zip: 19106

Telephone: 555-111-2222 Valid Telephone Format

Submit

First Name and Last Name must have a value.

l) The address book is redisplayed with the new name added in.

Address Book - Windows Internet Ex...

http://test.deitel.com

AddressBook | Add an Entry

Search By Last Name:

Cheryl Black

James Blue

John Gray

Meg Gold

Mike Brown

Fig. 15.9 | Ajax-enabled address-book application. (Part 10 of 10.)

The application also enables the user to add another entry to the address book by clicking the **addEntry** button (Fig. 15.9(f)). The application displays a form that enables live field validation. As the user fills out the form, the zip-code value is validated and used to generate the city and state (Fig. 15.9(g), Fig. 15.9(h) and Fig. 15.9(i)). The telephone number is validated for correct format (Fig. 15.9(j)). When the Submit button is clicked, the application checks for invalid data and stores the values in a database on the server (Fig. 15.9(k) and Fig. 15.9(l)). You can test-drive this application at test.deitel.com/examples/iw3htp4/ajax/fig15_09_10/AddressBook.html.

Interacting with a Web Service on the Server

When the page loads, the onload event (line 339) calls the showAddressBook function to load the address book onto the page. Function showAddressBook (lines 21–29) shows the addressBook element and hides the addEntry element using the HTML DOM (lines 24–25). Then it calls function callWebService to make an asynchronous request to the server (line 28). Function callWebService requires an array of parameter objects to be sent to the server. In this case, the function we are invoking on the server requires no arguments, so line 27 creates an empty array to be passed to callWebService. Our program uses an ASP.NET web service that we created for this example to do the server-side processing. The web service contains a collection of methods that can be called from a web application.

Function callWebService (lines 32–72) contains the code to call our web service, given a method name, an array of parameter bindings (i.e., the method's parameter names and argument values) and the name of a callback function. The web-service application and the method that is being called are specified in the request URL (line 35). When sending the request using the GET method, the parameters are concatenated URL starting with a ? symbol and followed by a list of *parameter=value* bindings, each separated by an &. Lines 39–49 iterate over the array of parameter bindings that was passed as an argument, and add them to the request URL. In this first call, we do not pass any parameters because the web method that returns all the entries requires none. However, future web method calls will send multiple parameter bindings to the web service. Lines 52–71 prepare and send the request, using similar functionality to the previous two examples. There are many types of user interaction in this application, each requiring a separate asynchronous request. For this reason, we pass the appropriate asyncRequest object as an argument to the function specified by the callBack parameter. However, event handlers cannot receive arguments, so lines 57–60 assign an anonymous function to asyncRequest's onreadystatechange property. When this anonymous function gets called, it calls function callBack and passes the asyncRequest object as an argument. Lines 64–65 set an Accept request header to receive JSON formatted data.

Parsing JSON Data

Each of our web service's methods in this example returns a JSON representation of an object or array of objects. For example, when the web application requests the list of names in the address book, the list is returned as a JSON array, as shown in Fig. 15.10. Each object in Fig. 15.10 has the attributes first and last.

Line 11 links the json.js script to the XHTML file so we can parse JSON data. When the XMLHttpRequest object receives the response, it calls function parseData (lines 75–84). Line 81 calls the string's parseJSON function, which converts the JSON string into a JavaScript object. Then line 82 calls function displayNames (lines 87–106), which

```
[ { "first": "Cheryl", "last": "Black" },
  { "first": "James", "last": "Blue" },
  { "first": "Mike", "last": "Brown" },
  { "first": "Meg", "last": "Gold" } ]
```

Fig. 15.10 | Address-book data formatted in JSON.

displays the first and last name of each address-book entry passed to it. Lines 90–91 use the DOM to store the placeholder `div` element `Names` in the variable `listbox`, and clear its content. Once parsed, the JSON string of address-book entries becomes an array, which this function traverses (lines 94–105).

Creating XHTML Elements and Setting Event Handlers on the Fly

Line 99 uses an XHTML `fieldset` element to create a box in which the entry will be placed. Line 100 registers function `handleOnClick` as the `onclick` event handler for the `div` created in line 98. This enables the user to expand each address-book entry by clicking it. Function `handleOnClick` (lines 109–113) calls the `getAddress` function whenever the user clicks an entry. The parameters are generated dynamically and not evaluated until the `getAddress` function is called. This enables each function to receive arguments that are specific to the entry the user clicked. Line 102 displays the names on the page by accessing the `first` (first name) and `last` (last name) fields of each element of the `data` array.

Function `getAddress` (lines 136–166) is called when the user clicks an entry. This request must keep track of the entry where the address is to be displayed on the page. Lines 151–154 set the `displayAddress` function (lines 168–187) as the callback function, and pass it the entry element as a parameter. Once the request completes successfully, lines 174–178 parse the response and display the addresses. Lines 181–184 update the `div`'s `onclick` event handler to hide the address data when that `div` is clicked again by the user. When the user clicks an expanded entry, function `clearField` (lines 190–197) is called. Lines 192–196 reset the entry's content and its `onclick` event handler to the values they had before the entry was expanded.

Implementing Type-Ahead

The `input` element declared in line 348 enables the user to search the address book by last name. As soon as the user starts typing in the input box, the `onkeyup` event handler calls the `search` function (lines 117–133), passing the `input` element's value as an argument. The `search` function performs an asynchronous request to locate entries with last names that start with its argument value. When the response is received, the application displays the matching list of names. Each time the user changes the text in the input box, function `search` is called again to make another asynchronous request.

The `search` function (lines 117–133) first clears the address-book entries from the page (lines 120–121). If the `input` argument is the empty string, line 126 displays the entire address book by calling function `showAddressBook`. Otherwise lines 130–131 send a request to the server to search the data. Line 130 creates a JSON string to represent the parameter object to be sent as an argument to the `callWebServices` function. Line 131 converts the string to an object and calls the `callWebServices` function. When the server responds, callback function `parseData` is invoked, which calls function `displayNames` to display the results on the page.

Implementing a Form with Asynchronous Validation

When the **Add an Entry** button (lines 343–344) is clicked, the addEntry function (lines 200–204) is called, which hides the addressBook element and shows the addEntry element that allows the user to add a person to the address book. The addEntry element (lines 353–380) contains a set of entry fields, some of which have event handlers that enable validation that occurs asynchronously as the user continues to interact with the page. When a user enters a zip code, the validateZip function (lines 207–212) is called. This function calls an external web service to validate the zip code. If it is valid, that external web service returns the corresponding city and state. Line 210 builds a parameter object containing validateZip's parameter name and argument value in JSON format. Line 211 calls the callWebService function with the appropriate method, the parameter object created in line 210 and showCityState (lines 216–258) as the callback function.

Zip-code validation can take a long time due to network delays. The showCityState function is called every time the request object's readyState property changes. Until the request completes, lines 219–220 display "Checking zip code..." on the page. After the request completes, line 228 converts the JSON response text to an object. The response object has four properties—Validity, ErrorText, City and State. If the request is valid, line 233 updates the zipValid variable that keeps track of zip-code validity (declared at line 18), and lines 237–239 show the city and state that the server generated using the zip code. Otherwise lines 243–245 update the zipValid variable and show the error code. Lines 248–249 clear the city and state elements. If our web service fails to connect to the zip-code validator web service, lines 252–256 display an appropriate error message.

Similarly, when the user enters the telephone number, the function validatePhone (lines 261–265) sends the phone number to the server. Once the server responds, the showPhoneError function (lines 268–288) updates the validatePhone variable (declared at line 17) and shows the message that the web service returned.

When the **Submit** button is clicked, the saveForm function is called (lines 291–335). Lines 294–300 retrieve the data from the form. Lines 303–308 check if the zip code and telephone number are valid, and display the appropriate error message in the Success element on the bottom of the page. Before the data can be entered into a database on the server, both the first-name and last-name fields must have a value. Lines 309–314 check that these fields are not empty and, if they are empty, display the appropriate error message. Once all the data entered is valid, lines 318–321 hide the entry form and show the address book. Lines 324–333 build the parameter object using JSON and send the data to the server using the callWebService function. Once the server saves the data, it queries the database for an updated list of entries and returns them; then function parseData displays the entries on the page.

15.8 Dojo Toolkit

Developing web applications in general, and Ajax applications in particular, involves a certain amount of painstaking and tedious work. Cross-browser compatibility, DOM manipulation and event handling can get cumbersome, particularly as an application's size increases. Dojo is a free, open source JavaScript library that takes care of these issues. Dojo reduces asynchronous request handling to a single function call. Dojo also provides cross-browser DOM functions that simplify partial page updates. It covers many more areas of web development, from simple event handling to fully functional rich GUI controls.

To install Dojo, download the Dojo version 0.4.3 from `www.Dojotoolkit.org/downloads` to your hard drive. Extract the files from the archive file you downloaded to your web development directory or web server. Including the `dojo.js` script file in your web application will give you access to all the Dojo functions. To do this, place the following script in the head element of your XHTML document:

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

where *path* is the relative or complete path to the Dojo toolkit's files. Quick installation instructions for Dojo are provided at `Dojotoolkit.org/book/Dojo-book-0-9/part-1-life-Dojo/quick-installation`.

Figure 15.11 is a calendar application that uses Dojo to create the user interface, communicate with the server asynchronously, handle events and manipulate the DOM. The application contains a calendar control that shows the user six weeks of dates (see the screen captures in Fig. 15.11). Various arrow buttons allow the user to traverse the calendar. When the user selects a date, an asynchronous request obtains from the server a list of the scheduled events for that date. There is an **Edit** button next to each scheduled event. When the **Edit** button is clicked, the item is replaced by a text box with the item's content, a **Save** button and a **Cancel** button. When the user presses **Save**, an asynchronous request saves the new value to the server and displays it on the page. This feature, often referred to as edit-in-place, is common in Ajax applications. You can test-drive this application at `test.deitel.com/examples/iw3htp4/ajax/fig15_11/calendar.html`.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 15.11 Calendar.html -->
<!-- Calendar application built with dojo. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
   <script type = "text/javascript" src = "/dojo043/dojo.js"></script>
   <script type = "text/javascript" src = "json.js"></script>
   <script type = "text/javascript">
      <!--
      // specify all the required dojo scripts
      dojo.require( "dojo.event.*" ); // use scripts from event package
      dojo.require( "dojo.widget.*" ); // use scripts from widget package
      dojo.require( "dojo.dom.*" ); // use scripts from dom package
      dojo.require( "dojo.io.*" ); // use scripts from the io package

      // configure calendar event handler
      function connectEventHandler()
      {
         var calendar = dojo.widget.byId( "calendar" ); // get calendar
         calendar.setDate( "2007-07-04" );
         dojo.event.connect(
            calendar, "onValueChanged", "retrieveItems" );
      } // end function connectEventHandler

```

Fig. 15.11 | Calendar application built with Dojo. (Part 1 of 7.)

```
      // location of CalendarService web service
      var webServiceUrl = "/CalendarService/CalendarService.asmx";

      // obtain scheduled events for the specified date
      function retrieveItems( eventDate )
      {
         // convert date object to string in yyyy-mm-dd format
         var date = dojo.date.toRfc3339( eventDate ).substring( 0, 10 );

          // build parameters and call web service
         var params = '[{ "param":"eventDate", "value":"' +
            date +  "'}]";
         callWebService( 'getItemsByDate', params, displayItems );
      } // end function retrieveItems

       // call a specific web service asynchronously to get server data
      function callWebService( method, params, callback )
      {
         // url for the asynchronous request
         var requestUrl = webServiceUrl + "/" + method;
         var params = paramString.parseJSON();

         // build the parameter string to append to the url
         for ( var i = 0; i < params.length; i++ )
         {
            // check if it is the first parameter and build
            // the parameter string accordingly
            if ( i == 0 )
               requestUrl = requestUrl + "?" + params[ i ].param +
                  "=" + params[ i ].value; // add first parameter to url
            else
               requestUrl = requestUrl + "&" + params[ i ].param +
                "=" + params[ i ].value; // add other parameters to url
         } // end for

         // call asynchronous request using dojo.io.bind
         dojo.io.bind( { url: requestUrl, handler: callback,
            accept: "application/json; charset=utf-8" } );
      } // end function callWebService

      // display the list of scheduled events on the page
      function displayItems( type, data, event )
      {
         if ( type == 'error' ) // if the request has failed
         {
            alert( 'Could not retrieve the event' ); // display error
         } // end if
         else
         {
            var placeholder = dojo.byId( "itemList" ); // get placeholder
            placeholder.innerHTML = ''; // clear placeholder
            var items = data.parseJSON(); // parse server data

```

Fig. 15.11 | Calendar application built with Dojo. (Part 2 of 7.)

```
            // check whether there are events;
            // if none then display message
            if ( items == "" )
            {
               placeholder.innerHTML = 'No events for this date.';
            }

            for ( var  i = 0; i < items.length; i++ )
            {
               // initialize item's container
               var item = document.createElement( "div" );
               item.id = items[ i ].id; // set DOM id to database id

               // obtain and paste the item's description
               var text = document.createElement( "div" );
               text.innerHTML =  items[i].description;
               text.id = 'description' + item.id;
               dojo.dom.insertAtIndex( text, item, 0 );

               // create and insert the placeholder for the edit button
               var buttonPlaceHolder = document.createElement( "div" );
               dojo.dom.insertAtIndex( buttonPlaceHolder, item, 1 );

               // create the edit button and paste it into the container
               var editButton = dojo.widget.
                  createWidget( "Button", {}, buttonPlaceHolder );
               editButton.setCaption( "Edit" );
               dojo.event.connect(
                  editButton, 'buttonClick', handleEdit );

               // insert item container in the list of items container
               dojo.dom.insertAtIndex( item, placeholder, i );
            } // end for
         } // end else
      } // end function displayItems

      // send the asynchronous request to get content for editing and
      // run the edit-in-place UI
      function handleEdit( event )
      {
         var id = event.currentTarget.parentNode.id; // retrieve id
         var params = '[{ "param":"id", "value":"' + id +  '"}]';
         callWebService( 'getItemById', params, displayForEdit );
      } // end function handleEdit

      // set up the interface for editing an item
      function displayForEdit(type, data, event)
      {
         if ( type == 'error' ) // if the request has failed
         {
            alert( 'Could not retrieve the event' ); // display error
         }
```

Fig. 15.11 | Calendar application built with Dojo. (Part 3 of 7.)

```
         else
         {
            var item = data.parseJSON(); // parse the item
            var id = item.id; // set the id

            // create div elements to insert content
            var editElement = document.createElement( 'div' );
            var buttonElement = document.createElement( 'div' );

            // hide the unedited content
            var oldItem = dojo.byId( id ); // get the original element
            oldItem.id = 'old' + oldItem.id; // change element's id
            oldItem.style.display = 'none'; // hide old element
            editElement.id = id; // change the "edit" container's id

            // create a textbox and insert it on the page
            var editArea = document.createElement( 'textarea' );
            editArea.id = 'edit' + id; // set textbox id
            editArea.innerHTML = item.description; // insert description
            dojo.dom.insertAtIndex( editArea, editElement, 0 );

            // create button placeholders and insert on the page
            // these will be transformed into dojo widgets
            var saveElement = document.createElement( 'div' );
            var cancelElement = document.createElement( 'div' );
            dojo.dom.insertAtIndex( saveElement, buttonElement, 0 );
            dojo.dom.insertAtIndex( cancelElement, buttonElement, 1 );
            dojo.dom.insertAtIndex( buttonElement, editElement, 1 );

            // create "save" and "cancel" buttons
            var saveButton =
               dojo.widget.createWidget( "Button", {}, saveElement );
            var cancelButton =
               dojo.widget.createWidget( "Button", {}, cancelElement );
            saveButton.setCaption( "Save" ); // set saveButton label
            cancelButton.setCaption( "Cancel" ); // set cancelButton text

            // set up the event handlers for cancel and save buttons
            dojo.event.connect( saveButton, 'buttonClick', handleSave );
            dojo.event.connect(
               cancelButton, 'buttonClick', handleCancel );

            // paste the edit UI on the page
            dojo.dom.insertAfter( editElement, oldItem );
         } // end else
      } // end function displayForEdit

      // sends the changed content to the server to be saved
      function handleSave( event )
      {
         // grab user entered data
         var id = event.currentTarget.parentNode.parentNode.id;
         var descr = dojo.byId( 'edit' + id ).value;
```

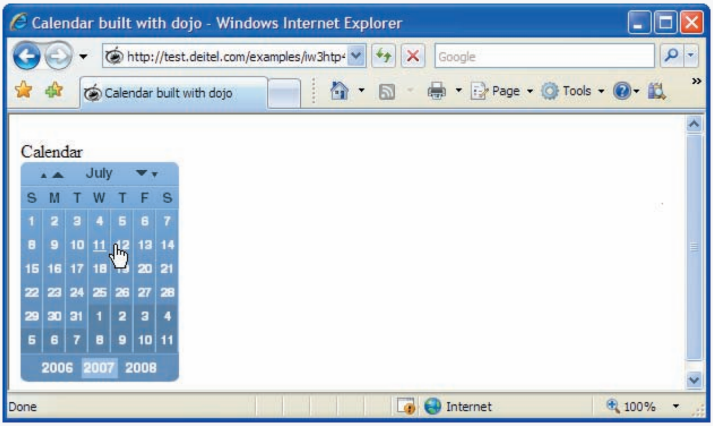
Fig. 15.11 | Calendar application built with Dojo. (Part 4 of 7.)

```

         // build parameter string and call the web service
         var params = '[{ "param":"id", "value":"' + id +
            '"}, {"param": "descr", "value":"' + descr + '"}]';
         callWebService( 'Save', params, displayEdited );
      } // end function handleSave

      // restores the original content of the item
      function handleCancel( event )
      {
         var voidEdit = event.currentTarget.parentNode.parentNode;
         var id = voidEdit.id; // retrieve the id of the item
         dojo.dom.removeNode( voidEdit, true ); // remove the edit UI
         var old = dojo.byId( 'old' + id ); // retrieve pre-edit version
         old.style.display = 'block'; // show pre-edit version
         old.id = id; // reset the id
      } // end function handleCancel

      // displays the updated event information after an edit is saved
      function displayEdited( type, data, event )
      {
         if ( type == 'error' )
         {
            alert( 'Could not retrieve the event' );
         }
         else
         {
            editedItem = data.parseJSON(); // obtain updated description
            var id = editedItem.id; // obtain the id
            var editElement = dojo.byId( id ); // get the edit UI
            dojo.dom.removeNode( editElement, true ); // delete edit UI
            var old = dojo.byId( 'old' + id ); // get item container

            // get pre-edit element and update its description
            var oldText = dojo.byId( 'description' + id );
            oldText.innerHTML = editedItem.description;

            old.id = id; // reset id
            old.style.display = 'block'; // show the updated item
         } // end else
      } // end function displayEdited

      // when the page is loaded, set up the calendar event handler
      dojo.addOnLoad( connectEventHandler );
      // -->
   </script>
   <title> Calendar built with dojo </title>
</head>
<body>
   Calendar
   <div dojoType = "datePicker" style = "float: left"
      widgetID = "calendar"></div>
   <div id = "itemList" style = "float: left"></div>
```

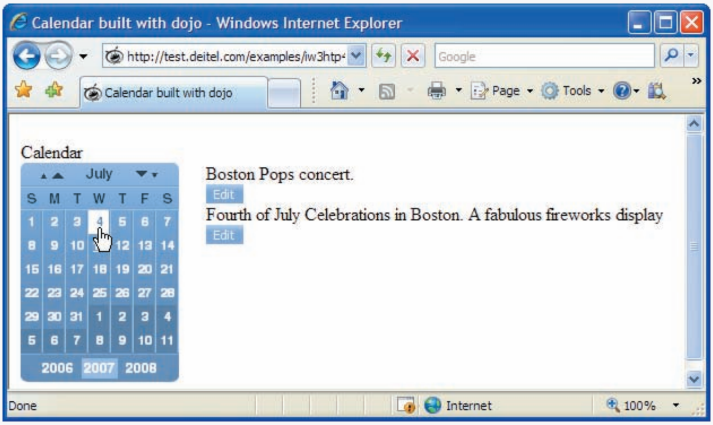
Fig. 15.11 | Calendar application built with Dojo. (Part 5 of 7.)

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

a) `DatePicker` Dojo widget after the web page loads.



b) User selects a date and the application asynchronously requests a list of events for that date and displays the results with a partial page update.



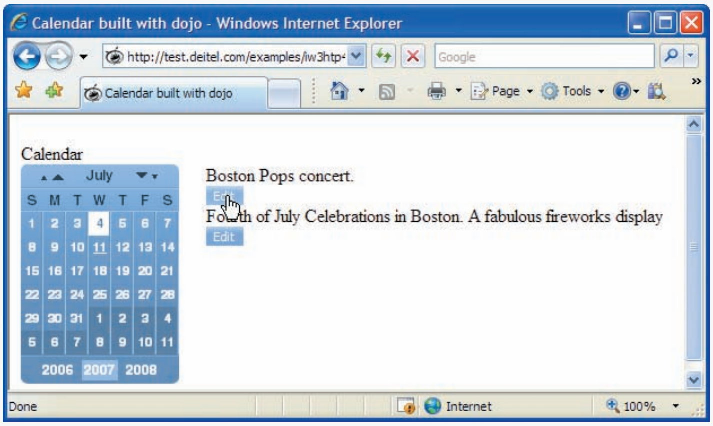
Boston Pops concert.

Edit

Fourth of July Celebrations in Boston. A fabulous fireworks display

Edit

c) User clicks the **Edit** button to modify an event's description.



Boston Pops concert.

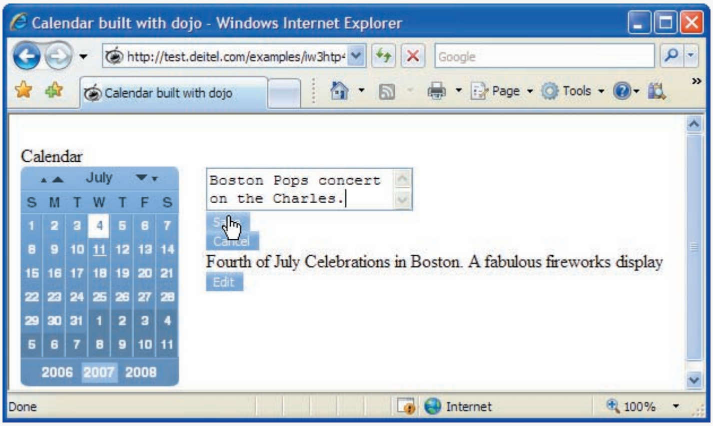
Edit

Fourth of July Celebrations in Boston. A fabulous fireworks display

Edit

Fig. 15.11 | Calendar application built with Dojo. (Part 6 of 7.)

d) Application performs a partial page update, replacing the original description and the **Edit** button with a text box, **Save** button and **Cancel** button. User modifies the event description and clicks the **Save** button.


Calendar built with dojo - Windows Internet Explorer
http://test.deitel.com/examples/iw3htp4
Google
Calendar built with dojo
Page
Tools
Calendar
July
Save
Cancel
Edit
Done
Internet
100%

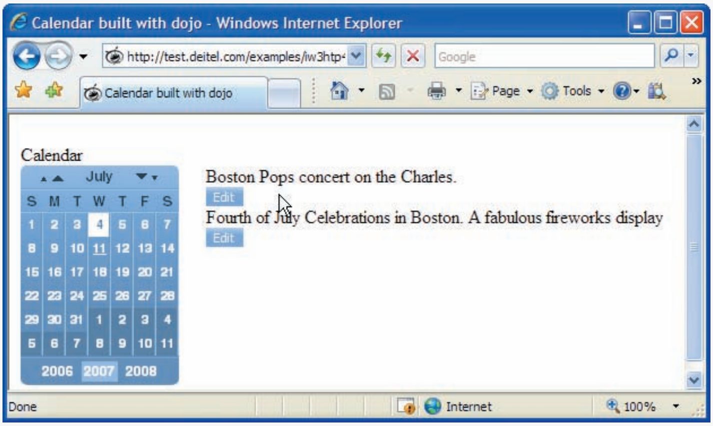
S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4
5	6	7	8	9	10	11

2006 2007 2008

```
Boston Pops concert
on the Charles.
```

Fourth of July Celebrations in Boston. A fabulous fireworks display

d) The **Save** button's event handler uses an asynchronous request to update the server and uses the server's response to perform a partial page update, replacing the editing GUI components with the updated description and an **Edit** button.


Calendar built with dojo - Windows Internet Explorer
http://test.deitel.com/examples/iw3htp4
Google
Calendar built with dojo
Page
Tools
Calendar
July
Edit
Edit
Done
Internet
100%

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4
5	6	7	8	9	10	11

2006 2007 2008

Boston Pops concert on the Charles.

Fourth of July Celebrations in Boston. A fabulous fireworks display

Fig. 15.11 | Calendar application built with Dojo. (Part 7 of 7.)

Loading Dojo Packages

Lines 9–17 load the Dojo framework. Line 9 links the `dojo.js` script file to the page, giving the script access to all the functions in the Dojo toolkit. Dojo is organized in packages of related functionality. Lines 14–17 use the `dojo.require` call, provided by the `dojo.js` script to include the packages we need. The `dojo.io` package functions communicate with the server, the `dojo.event` package simplifies event handling, the `dojo.widget` package provides rich GUI controls, and the `dojo.dom` package contains additional DOM functions that are portable across many different browsers.

The application cannot use any of this functionality until all the packages have been loaded. Line 229 uses the `dojo.addOnLoad` method to set up the event handling after the page loads. Once all the packages have been loaded, the `connectEventHandler` function (lines 20–26) is called.

Using an Existing Dojo Widget

A Dojo widget is any predefined user interface element that is part of the Dojo toolkit. The calendar control on the page is the `DatePicker` widget. To incorporate an existing

Dojo widget onto a page, you must set the `DojoType` attribute of any HTML element to the type of widget that you want it to be (line 236). Dojo widgets also have their own `widgetID` property (line 237). Line 22 uses the `dojo.widget.byId` method, rather than the DOM's `document.getElementById` method, to obtain the calendar widget element. The `dojo.events.connect` method links functions together. Lines 24–25 use it to connect the calendar's `onValueChanged` event handler to the `retrieveItems` function. When the user picks a date, a special `onValueChanged` event that is part of the `DatePicker` widget calls `retrieveItems,` passing the selected date as an argument. The `retrieveItems` function (lines 32–41) builds the parameters for the request to the server, and calls the `callWebService` function. Line 35 uses the `dojo.date.toRfc3339` method to convert the date passed by the calendar control to *yyyy-mm-dd* format.

Asynchronous Requests in Dojo

The `callWebService` function (lines 44–66) sends the asynchronous request to the specified web-service method. Lines 47–61 build the request URL using the same code as Fig. 15.9. Dojo reduces the asynchronous request to a single call to the `dojo.io.bind` method (lines 64–65), which works on all the popular browsers such as Firefox, Internet Explorer, Opera, Mozilla and Safari. The method takes an array of parameters, formatted as a JavaScript object. The `url` parameter specifies the destination of the request, the `handler` parameter specifies the callback function, and the `mimetype` parameter specifies the format of the response. The `handler` parameter can be replaced by the `load` and `error` parameters. The function passed as `load` handles successful requests and the function passed as `error` handles unsuccessful requests.

Response handling is done differently in Dojo. Rather than calling the callback function every time the request's `readyState` property changes, Dojo calls the function passed as the "handler" parameter when the request completes. In addition, in Dojo the script does not have access to the request object. All the response data is sent directly to the callback function The function sent as the `handler` argument must have three parameters—type, data and event.

In the first request, the function `displayItems` (lines 69–115) is set as the callback function. Lines 71–74 check if the request is successful, and display an error message if it isn't. Lines 77–78 obtain the place-holder element (`itemList`), where the items will be displayed, and clear its content. Line 79 converts the JSON response text to a JavaScript object, using the same code as the example in Fig. 15.9.

Partial Page Updates Using Dojo's Cross-Browser DOM Manipulation Capabilities

The Dojo toolkit (like most other Ajax libraries) provides functionality that enables you to manipulate the DOM in a cross-browser portable manner. Lines 83–86 check if the server-side returned any items, and display an appropriate message if it didn't. For each item object returned from the server, lines 91–92 create a `div` element and set its `id` to the item's `id` in the database. Lines 95–97 create a container element for the item's description. Line 98 uses Dojo's `dojo.dom.insertAtIndex` method to insert the description element as the first element in the item's element.

For each entry, the application creates an **Edit** button that enables the user to edit the event's content on the page. Lines 101–109 create a Dojo `Button` widget programmatically. Lines 101–102 create a `buttonPlaceHolder` `div` element for the button and paste it on the page. Lines 105–106 convert the `buttonPlaceHolder` element to a Dojo `Button` widget by calling the `dojo.widget.createWidget` function. This function takes three parameters—the type of widget to be created, a list of additional widget parameters and the element which is to be converted to a Dojo widget. Line 107 uses the button's `setCaption` method to set the text that appears on the button. Line 112 uses the `insertAtIndex` method to insert the items into the `itemList` placeholder, in the order in which they were returned from the server.

Adding Edit-In-Place Functionality

Dojo `Button` widgets use their own `buttonClick` event instead of the DOM `onclick` event to store the event handler. Lines 108–109 use the `dojo.event.connect` method to connect the `buttonClick` event of the Dojo `Button` widget and the `handleEdit` event handler (lines 119–124). When the user clicks the **Edit** button, the `Event` object gets passed to the event handler as an argument. The `Event` object's `currentTarget` property contains the element that initiated the event. Line 121 uses the `currentTarget` property to obtain the `id` of the item. This `id` is the same as the item's `id` in the server database. Line 123 calls the web service's `getItemById` method, using the `callWebService` function to obtain the item that needs to be edited.

Once the server responds, the function `displayForEdit` (lines 127–178) replaces the item on the screen with the user interface used for editing the item's content. The code for this is similar to the code in the `displayItems` function. Lines 129–132 make sure the request was successful and parse the data from the server. Lines 139–140 create the container elements into which we insert the new user-interface elements. Lines 143–146 hide the element that displays the item and change its `id`. Now the `id` of the user-interface element is the same as the `id` of the item that it's editing stored in the database. Lines 149–152 create the text-box element that will be used to edit the item's description, paste it into the text box, and paste the resulting text box on the page. Lines 156–173 use the same syntax that was used to create the **Edit** button widget to create **Save** and **Cancel** button widgets. Line 176 pastes the resulting element, containing the text box and two buttons, on the page.

When the user edits the content and clicks the **Cancel** button, the `handleCancel` function (lines 194–202) restores the item element to what it looked like before the button was clicked. Line 198 deletes the edit UI that was created earlier, using Dojo's `removeNode` function. Lines 200–201 show the item with the original element that was used to display the item, and change its `id` back to the item's `id` on the server database.

When the user clicks the **Save** button, the `handleSave` function (lines 181–191) sends the text entered by the user to the server. Line 185 obtains the text that the user entered in the text box. Lines 188–190 send to the server the `id` of the item that needs to be updated and the new description.

Once the server responds, `displayEdited` (lines 205–226) displays the new item on the page. Lines 214–217 contain the same code that was used in `handleCancel` to remove the user interface used to edit the item and redisplay the element that contains the item. Line 221 changes the item's description to its new value.

15.9 Wrap-Up

In this chapter, we introduced Ajax and showed how to use it to create Rich Internet Applications (RIAs) that approximate the look, feel and usability of desktop applications.

You learned that RIAs have two key attributes—performance and a rich GUI. We discussed that RIA performance comes from Ajax (Asynchronous JavaScript and XML), which uses client-side scripting to make web applications more responsive by separating client-side user interaction and server communication, and running them in parallel.

You learned various ways to develop Ajax applications. We showed how to use "raw" Ajax with its component technologies (XHTML, CSS, JavaScript, dynamic HTML, the DOM, XML and the `XMLHttpRequest` object) to manage asynchronous requests to the server, then process the server responses (via JavaScript event handling) to perform partial page updates with the DOM on the client. You learned how to implement client/server communication using XML, and how to parse server responses using the DOM.

We discussed the impracticality of "raw" Ajax for developing large-scale applications and the hiding of such portability issues by Ajax toolkits, such as Dojo, Prototype, Script.aculo.us and ASP.NET Ajax, and by RIA environments such as Adobe's Flex (Chapter 18), Microsoft's Silverlight (Chapter 19) and JavaServer Faces (Chapters 26–27). You also learned that these Ajax libraries and RIA environments provide powerful ready-to-use GUI controls and functions that enrich web applications. You used two data formats—XML and JSON (JavaScript Object Notation—for communicating between the server and client. We then built an Ajax application with a rich calendar GUI using the Dojo Ajax toolkit. In the applications we presented, you learned techniques including callback functions for handling asynchronous responses, partial page updates to display response data, JavaScript exception handling, type-ahead capabilities for making suggestions to users as they type in a text field and edit-in-place capabilities so users can edit entries directly in a web page.

In subsequent chapters, we use tools such as Adobe Flex, Microsoft Silverlight, Ruby on Rails and JavaServer Faces to build RIAs using Ajax. In Chapter 24, we'll demonstrate features of the Prototype and Script.aculo.us Ajax libraries, which come with the Ruby on Rails framework. Prototype provides capabilities similar to Dojo. Script.aculo.us provides many "eye candy" effects that enable you to beautify your Ajax applications and create rich interfaces. In Chapter 27, we present Ajax-enabled JavaServer Faces (JSF) components. JSF uses Dojo to implement many of its client-side Ajax capabilities.

15.10 Web Resources

`www.deitel.com/ajax`

Our *Ajax Resource Center* contains links to some of the best Ajax resources on the web from which you can learn more about Ajax and its component technologies. Find categorized links to Ajax tools, code, forums, books, libraries, frameworks, conferences, podcasts and more. Check out the tutorials for all skill levels, from introductory to advanced. See our comprehensive list of developer toolkits and libraries. Visit the most popular Ajax community websites and blogs. Explore many popular commercial and free open-source Ajax applications. Download code snippets and complete scripts that you can use on your own website. Also, be sure to visit our Resource Centers with information on Ajax's component technologies, including XHTML (`www.deitel.com/xhtml/`), CSS 2.1 (`www.deitel.com/css21/`), XML (`www.deitel.com/XML/`), and JavaScript (`www.deitel.com/javascript/`). For a complete list of Resource Centers, visit `www.deitel.com/ResourceCenters.html`.

Summary

Section 15.1 Introduction

- Despite the tremendous technological growth of the Internet over the past decade, the usability of web applications has lagged behind compared to desktop applications.
- Rich Internet Applications (RIAs) are web applications that approximate the look, feel and usability of desktop applications. RIAs have two key attributes—performance and rich GUI.
- RIA performance comes from Ajax (Asynchronous JavaScript and XML), which uses client-side scripting to make web applications more responsive.
- Ajax applications separate client-side user interaction and server communication, and run them in parallel, making the delays of server-side processing more transparent to the user.
- “Raw” Ajax uses JavaScript to send asynchronous requests to the server, then updates the page using the DOM.
- When writing “raw” Ajax you need to deal directly with cross-browser portability issues, making it impractical for developing large-scale applications.
- Portability issues are hidden by Ajax toolkits, such as Dojo, Prototype and Script.aculo.us, which provide powerful ready-to-use controls and functions that enrich web applications and simplify JavaScript coding by making it cross-browser compatible.
- We achieve rich GUI in RIAs with Ajax toolkits and with RIA environments such as Adobe’s Flex, Microsoft’s Silverlight and JavaServer Faces. Such toolkits and environments provide powerful ready-to-use controls and functions that enrich web applications.
- The client-side of Ajax applications is written in XHTML and CSS, and uses JavaScript to add functionality to the user interface.
- XML and JSON are used to structure the data passed between the server and the client.
- The Ajax component that manages interaction with the server is usually implemented with JavaScript’s `XMLHttpRequest` object—commonly abbreviated as XHR.

Section 15.2 Traditional Web Applications vs. Ajax Applications

- In traditional web applications, the user fills in the form’s fields, then submits the form. The browser generates a request to the server, which receives the request and processes it. The server generates and sends a response containing the exact page that the browser will render, which causes the browser to load the new page and temporarily makes the browser window blank. The client *waits* for the server to respond and *reloads the entire page* with the data from the response.
- While a synchronous request is being processed on the server, the user cannot interact with the client web browser.
- The synchronous model was originally designed for a web of hypertext documents—what some people call the “brochure web.” This model yielded “choppy” application performance.
- In an Ajax application, when the user interacts with a page, the client creates an `XMLHttpRequest` object to manage a request. The `XMLHttpRequest` object sends the request to and awaits the response from the server. The requests are asynchronous, allowing the user to continue interacting with the application while the server processes the request concurrently. When the server responds, the `XMLHttpRequest` object that issued the request invokes a callback function, which typically uses partial page updates to display the returned data in the existing web page *without reloading the entire page*.
- The callback function updates only a designated part of the page. Such partial page updates help make web applications more responsive, making them feel more like desktop applications.

Section 15.3 Rich Internet Applications (RIAs) with Ajax

- A classic XHTML registration form sends all of the data to be validated to the server when the user clicks the **Register** button. While the server is validating the data, the user cannot interact with the page. The server finds invalid data, generates a new page identifying the errors in the form and sends it back to the client—which renders the page in the browser. Once the user fixes the errors and clicks the **Register** button, the cycle repeats until no errors are found, then the data is stored on the server. The entire page reloads every time the user submits invalid data.
- Ajax-enabled forms are more interactive. Entries are validated dynamically as the user enters data into the fields. If a problem is found, the server sends an error message that is asynchronously displayed to inform the user of the problem. Sending each entry asynchronously allows the user to address invalid entries quickly, rather than making edits and resubmitting the entire form repeatedly until all entries are valid. Asynchronous requests could also be used to fill some fields based on previous fields' values.

Section 15.4 History of Ajax

- The term Ajax was coined by Jesse James Garrett of Adaptive Path in February 2005, when he was presenting the previously unnamed technology to a client.
- All of the technologies involved in Ajax (XHTML, JavaScript, CSS, dynamic HTML, the DOM and XML) have existed for many years.
- In 1998, Microsoft introduced the `XMLHttpRequest` object to create and manage asynchronous requests and responses.
- Popular applications like Flickr, Google's Gmail and Google Maps use the `XMLHttpRequest` object to update pages dynamically.
- The name Ajax immediately caught on and brought attention to its component technologies. Ajax has quickly become one of the hottest technologies in web development, as it enables webtop applications to challenge the dominance of established desktop applications.

Section 15.5 "Raw" Ajax Example using the `XMLHttpRequest` Object

- The `XMLHttpRequest` object (which resides on the client) is the layer between the client and the server that manages asynchronous requests in Ajax applications. This object is supported on most browsers, though they may implement it differently.
- To initiate an asynchronous request, you create an instance of the `XMLHttpRequest` object, then use its `open` method to set up the request, and its `send` method to initiate the request.
- When an Ajax application requests a file from a server, the browser typically caches that file. Subsequent requests for the same file can load it from the browser's cache.
- For security purposes, the `XMLHttpRequest` object does not allow a web application to request resources from servers other than the one that served the web application.
- Making a request to a different server is known as cross-site scripting (also known as XSS). You can implement a server-side proxy—an application on the web application's web server—that can make requests to other servers on the web application's behalf.
- When the third argument to `XMLHttpRequest` method `open` is `true`, the request is asynchronous.
- An exception is an indication of a problem that occurs during a program's execution.
- Exception handling enables you to create applications that can resolve (or handle) exceptions—in some cases allowing a program to continue executing as if no problem had been encountered.
- A `try` block encloses code that might cause an exception and code that should not execute if an exception occurs. A `try` block consists of the keyword `try` followed by a block of code enclosed in curly braces (`{}`).

- When an exception occurs, a `try` block terminates immediately and a `catch` block (also called a `catch` clause or exception handler) catches (i.e., receives) and handles an exception.
- The `catch` block begins with the keyword `catch` and is followed by an exception parameter in parentheses and a block of code enclosed in curly braces.
- The exception parameter's name enables the `catch` block to interact with a caught exception object, which contains `name` and `message` properties.
- A callback function is registered as the event handler for the `XMLHttpRequest` object's `onreadystatechange` event. Whenever the request makes progress, the `XMLHttpRequest` calls the `onreadystatechange` event handler.
- Progress is monitored by the `readyState` property, which has a value from 0 to 4. The value 0 indicates that the request is not initialized and the value 4 indicates that the request is complete.

Section 15.6 Using XML and the DOM

- When passing structured data between the server and the client, Ajax applications often use XML because it consumes little bandwidth and is easy to parse.
- When the `XMLHttpRequest` object receives XML data, the `XMLHttpRequest` object parses and stores the data as a DOM object in the `responseXML` property.
- The `XMLHttpRequest` object's `responseXML` property contains the XML returned by the server.
- DOM method `createElement` creates an XHTML element of the specified type.
- DOM method `setAttribute` adds or changes an attribute of an XHTML element.
- DOM method `appendChild` inserts one XHTML element into another.
- The `innerHTML` property of a DOM element can be used to obtain or change the XHTML that is displayed in a particular element.

Section 15.7 Creating a Full-Scale Ajax-Enabled Application

- JSON (JavaScript Object Notation)—a simple way to represent JavaScript objects as strings—is an alternative way (to XML) for passing data between the client and the server.
- Each JSON object is represented as a list of property names and values contained in curly braces.
- An array is represented in JSON with square brackets containing a comma-separated list of values.
- Each value in a JSON array can be a string, a number, a JSON representation of an object, `true`, `false` or `null`.
- JavaScript's `eval` function can convert JSON strings into JavaScript objects. To evaluate a JSON string properly, a left parenthesis should be placed at the beginning of the string and a right parenthesis at the end of the string before the string is passed to the `eval` function.
- The `eval` function creates a potential security risk—it executes any embedded JavaScript code in its string argument, possibly allowing a harmful script to be injected into JSON. A more secure way to process JSON is to use a JSON parser
- JSON strings are easier to create and parse than XML and require fewer bytes. For these reasons, JSON is commonly used to communicate in client/server interaction.
- When a request is sent using the `GET` method, the parameters are concatenated to the URL. URL parameter strings start with a `?` symbol and have a list of *parameter-value* bindings, each separated by an `&`.
- To implement type-ahead, you can use an element's `onkeyup` event handler to make asynchronous requests.

Section 15.8 Dojo Toolkit

- Developing web applications in general, and Ajax applications in particular, involves a certain amount of painstaking and tedious work. Cross-browser compatibility, DOM manipulation and event handling can get cumbersome, particularly as an application's size increases. Dojo is a free, open source JavaScript library that takes care of these issues.
- Dojo reduces asynchronous request handling to a single function call.
- Dojo provides cross-browser DOM functions that simplify partial page updates. It also provides event handling and rich GUI controls.
- To install Dojo, download the latest release from `www.Dojotoolkit.org/downloads` to your hard drive. Extract the files from the archive file you downloaded to your web development directory or web server. To include the `Dojo.js` script file in your web application, place the following script in the head element of your XHTML document:

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

  where *path* is the relative or complete path to the Dojo toolkit's files.
- Edit-in-place enables a user to modify data directly in the web page, a common feature in Ajax applications.
- Dojo is organized in packages of related functionality.
- The `dojo.require` method is used to include specific Dojo packages.
- The `dojo.io` package functions communicate with the server, the `dojo.event` package simplifies event handling, the `dojo.widget` package provides rich GUI controls, and the `dojo.dom` package contains additional DOM functions that are portable across many different browsers.
- A Dojo widget is any predefined user interface element that is part of the Dojo toolkit.
- To incorporate an existing Dojo widget onto a page, you must set the `dojoType` attribute of any HTML element to the type of widget that you want it to be.
- The `dojo.widget.byId` method can be used to obtain a Dojo widget.
- The `dojo.events.connect` method links functions together.
- The `dojo.date.toRfc3339` method converts a date to *yyyy-mm-dd* format.
- The `dojo.io.bind` method configures and sends asynchronous requests. The method takes an array of parameters, formatted as a JavaScript object. The `url` parameter specifies the destination of the request, the `handler` parameter specifies the callback function, and the `mimetype` parameter specifies the format of the response. The `handler` parameter can be replaced by the `load` and `error` parameters. The function passed as the `load` handler processes successful requests and the function passed as the `error` handler processes unsuccessful requests.
- Dojo calls the function passed as the `handler` parameter only when the request completes.
- In Dojo, the script does not have access to the request object. All the response data is sent directly to the callback function.
- The function sent as the `handler` argument must have three parameters—`type`, `data` and `event`.
- The Dojo toolkit (like most other Ajax libraries) provides functionality that enables you to manipulate the DOM in a cross-browser manner.
- Dojo's `dojo.dom.insertAtIndex` method inserts an element at the specified index in the DOM.
- Dojo's `removeNode` function removes an element from the DOM.
- Dojo `Button` widgets use their own `buttonClick` event instead of the DOM `onclick` event to store the event handler.
- The `Event` object's `currentTarget` property contains the element that initiated the event.

Terminology

Ajax
Ajax toolkit
asynchronous request
callback function
`catch` block
`catch` clause
`catch` keyword
cross-browser compatibility
cross-site scripting (XSS)
Dojo Ajax library
edit-in-place
exception
exception handler
exception handling
`GET` method of `XMLHttpRequest` object
`getResponseHeader` method of `XMLHttpRequest` object
JavaScript Object Notation (JSON)
`onReadyStateChange` property of the `XMLHttpRequest` object
`open` method of `XMLHttpRequest`
partial page update
Prototype Ajax library
"raw" Ajax
`readyState` property of `XMLHttpRequest` object
`responseText` property of `XMLHttpRequest` object
`responseXML` property of `XMLHttpRequest` object
same origin policy (SOP)
Script.aculo.us Ajax library
`send` method of `XMLHttpRequest`
`setRequestHeader` method of `XMLHttpRequest` object
`status` property of `XMLHttpRequest` object
`statusText` property of `XMLHttpRequest` object
synchronous request
`try` block
`try` keyword
type ahead
XHR (abbreviation for `XMLHttpRequest`)
`XMLHttpRequest` object

Self-Review Exercises

15.1 Fill in the blanks in each of the following statements:

a) Ajax applications use ________ requests to create Rich Internet Applications.
b) In Ajax applications, the ________ object manages asynchronous interaction with the server.
c) The event handler called when the server responds is known as a(n) ________ function.
d) The ________ attribute can be accessed through the DOM to update an XHTML element's content without reloading the page.
e) JavaScript's `XMLHttpRequest` object is commonly abbreviated as ________.
f) ________ is a simple way to represent JavaScript objects as strings.
g) Making a request to a different server is known as ________.
h) JavaScript's ________ function can convert JSON strings into JavaScript objects.
i) A(n) ________ encloses code that might cause an exception and code that should not execute if an exception occurs.
j) The `XMLHttpRequest` object's ________ contains the XML returned by the server.

15.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Ajax applications must use XML for server responses.
b) The technologies that are used to develop Ajax applications have existed since the 1990s.
c) The event handler that processes the response is stored in the `readyState` property of `XMLHttpRequest`.
d) An Ajax application can be implemented so that it never needs to reload the page on which it runs.
e) The `responseXML` property of the `XMLHttpRequest` object stores the server's response as a raw XML string.

f) The Dojo toolkit (like most other Ajax libraries) provides functionality that enables you to manipulate the DOM in a cross-browser manner.
g) An exception indicates successful completion of a program's execution.
h) When the third argument to `XMLHttpRequest` method `open` is `false`, the request is asynchronous.
i) For security purposes, the `XMLHttpRequest` object does not allow a web application to request resources from servers other than the one that served the web application.
j) The `innerHTML` property of a DOM element can be used to obtain or change the XHTML that is displayed in a particular element.

Answers to Self-Review Exercises

15.1 a) asynchronous. b) `XMLHttpRequest`. c) callback. d) `innerHTML`. e) XHR. f) JSON. g) cross-site scripting (or XSS). h) `eval`. i) `try` block. j) `responseXML` property.

15.2 a) False. Ajax applications can use any type of textual data as a response. For example, we used JSON in this chapter.
b) True.
c) False. `readyState` is the property that keeps track of the request's progress. The event handler is stored in the `onReadyStateChange` property.
d) True.
e) False. If the response data has XML format, the `XMLHttpRequest` object parses it and stores it in a document object.
f) True.
g) False. An exception is an indication of a problem that occurs during a program's execution.
h) False. The third argument to `XMLHttpRequest` method `open` must be `true` to make an asynchronous request.
i) True.
j) True.

Exercises

15.3 Describe the differences between client/server interactions in traditional web applications and client/server interactions in Ajax web applications.

15.4 Consider the `AddressBook` application in Fig. 15.9. Describe how you could reimplement the type-ahead capability so that it could perform the search using data previously downloaded rather than making an asynchronous request to the server after every keystroke.

15.5 Describe each of the following terms in the context of Ajax:
a) type-ahead
b) edit-in-place
c) partial page update
d) asynchronous request
e) `XMLHttpRequest`
f) "raw" Ajax
g) callback function
h) same origin policy
i) Ajax libraries
j) RIA

[*Note to Instructors and Students:* Due to security restrictions on using `XMLHttpRequest`, Ajax applications must be placed on a web server (even one on your local computer) to enable the applications to work correctly, and when they need to access other resources, those must reside on the same web server. ***Students:*** You'll need to work closely with your instructors to understand your lab setup so you can run your solutions to the exercises (the examples are already posted on our web server) and to run many of the other server-side applications that you'll learn later in the book.]

15.6 The XML files used in the book-cover catalog example (Fig. 15.8) also store the titles of the books in a `title` attribute of each `cover` node. Modify the example so that every time the mouse hovers over an image, the book's title is displayed below the image.

15.7 Create an Ajax-enabled version of the feedback form from Fig. 4.13. As the user moves between form fields, ensure that each field is non-empty. For the e-mail field, ensure that the e-mail address has valid format. In addition, create an XML file that contains a list of email addresses that are not allowed to post feedback. Each time the user enters an e-mail address check whether it is on that list; if so, display an appropriate message.

15.8 Create an Ajax-based product catalog that obtains its data from JSON files located on the server. The data should be separated into four JSON files. The first file should be a summary file, containing a list of products. Each product should have a title, an image filename for a thumbnail image and a price. The second file should contain a list of descriptions for each product. The third file should contain a list of filenames for the full-size product images. The last file should contain a list of the thumbnail image file names. Each item in a catalogue should have a unique ID that should be included with the entries for that product in every file. Next, create an Ajax-enabled web page that displays the product information in a table. The catalog should initially display a list of product names with their associated thumbnail images and prices. When the mouse hovers over a thumbnail image, the larger product image should be displayed. When the user moves the mouse away from that image, the original thumbnail should be redisplayed. You should provide a button that the user can click to display the product description.

15.9 Create a version of Exercise 15.8 that uses Dojo's capabilities and widgets to display the product catalog. Modify the asynchronous request's to use `dojo.io.bind` functions rather than raw Ajax. Use Dojo's DOM functionality to place elements on the page. Improve the look of the page by using Dojo's button widgets rather than XHTML button elements..